VISTA: Vpo Interactive System for Tuning Applications*

Baosheng Cai[1], Wankang Zhao[1], David Whalley[1], Robert van Engelen[1], Xin Yuan[1], Jack Davidson[2,3], and Kyle Gallivan[1]

[1] Dept. of Computer Science, Florida State University, Tallahassee, FL 32306-4530
{cai,wankzhao,whalley,engelen,xyuan,gallivan}@cs.fsu.edu

[2] Microsoft Research, Redmond, WA 98052
[3] Dept. of Computer Science, University of Virginia, Charlottesville, VA 22903
{jwd}@virginia.edu

Abstract. Application developers face many challenges when developing applications for embedded systems. A major challenge is meeting the conflicting constraints of speed, code density, and power consumption. Traditional optimizing compiler technology is usually of little help in meeting such constraints. To meet speed, power, and size constraints application developers typically resort to hand-coded assembly language. The results are software systems that are not portable, less robust, and more costly to develop and maintain. This paper describes a new interactive compilation paradigm implemented in a system called *vista* that we believe can help achieve the cost/performance tradeoffs demanded by embedded applications. 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, the *vista* system opens the optimization process and gives the application programmer, when necessary, the ability to finely control it. In particular, *vista* allows 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) view the low-level program representation in a graphical display. *Vista* can be used by embedded systems developers to produce applications, by compiler writers to debug errors and/or prototype new low-level code-improving transformations, and by a instructor/educator to illustrate code-improving transformations (e.g. in a compilers class).

1 Introduction

The problem of automatically generating acceptable code for embedded microprocessors is much more complicated than for general-purpose processors. First,

* This work was supported in part by NSF grants EIA-9806525, CCR-9904943, and EIA-0072043.

embedded applications are optimized for a number of conflicting constraints. In addition to speed, other common constraints are code size and power consumption. For many embedded applications, code density and power consumption are often more critical than speed. 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.

Automatic compilation for embedded microprocessors is further complicated because 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.

In this paper we describe 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 programmer 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 code sequence that the *vista* inserts and integrates into the program. We are not aware of any compiler that allows a programmer 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. Once a code-improving transformation is applied in a typical compiler, the effects of that transformation cannot be reversed. 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 programmer when interacting with the optimization engine, *vista* should provide the ability for the programmer to view the current program representation and any relevant compilation state information (i.e., live registers, available registers, def-use information, etc.).

Fig. 1 depicts a high-level overview of the flow of information in the interactive compilation system. The programmer would initially specify a source file to be compiled. The programmer retains control over the code-improvement

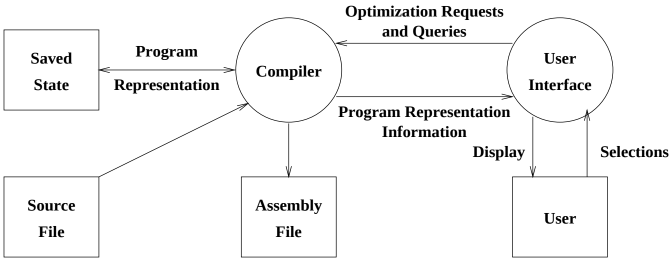

Fig. 1. Overview of the Interactive Compilation Process

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 programmer may also wish to save multiple optimized versions to contrast their performance. Eventually, the programmer may be satisfied with the generated code and can produce the assembly. In addition, the programmer 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.

Beyond *vista*'s primary purpose of supporting development of embedded systems, it has several other uses. First, *vista* can assist a compiler writer in debugging errors and/or prototyping new low-level code-improving transformations by specifying them by hand. Second, it can help compiler writers in understanding the interaction and interplay of different optimizations. Finally, an instructor or educator teaching compilation techniques can use the system to illustrate code-improving transformations to students.

The remainder of this paper is structured as follows. First, we review related work regarding alternative compilation paradigms and user interfaces for compilers. Second, we describe the underlying structure of *vista*'s optimization engine. Third, we present the functionality of the types of compilation requests that a programmer may make and show how this functionality is achieved in the user interface. Fourth, we discuss implementation issues that were involved in developing an interactive compilation system. Fifth, we describe the current status of the implementation. Finally, we give the conclusions of the paper.

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.

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 parallelizing 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.

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].

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 call RTLs (register transfer lists). RTL is a low-level, machine- and language-independent representation that encodes machine-specific instructions. The comprehensive use of RTL in *vpo* has several important consequences. Because there is a single representation, *vpo* offers the possibility of applying analyses and optimizations in an arbitrary order and repeatedly. In addition, the use of RTL 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.

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.

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 its analysis phases (e.g. dataflow analysis, controlflow 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.

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. 2 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, we provide options to display additional information about the program representation that a compiler writer may find useful.

The left side of Fig. 2 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 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. 2 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

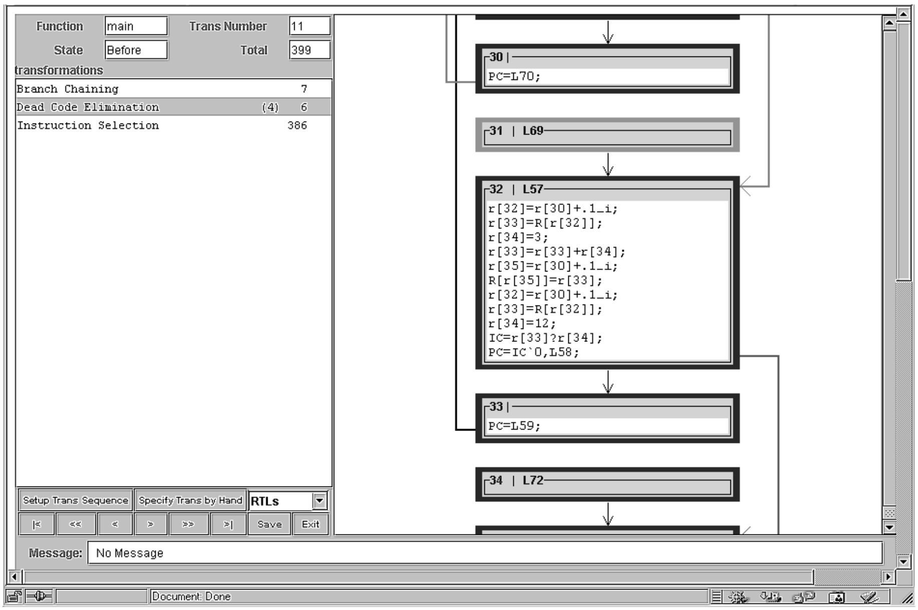

Fig. 2. User Interface Depicting a History of the Compilation Phases Performed

associated with the |<, <<, < buttons are described in Section 4.5. 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. 2 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.

4.2 Directing the Order 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. 3 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 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 subexpression 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.

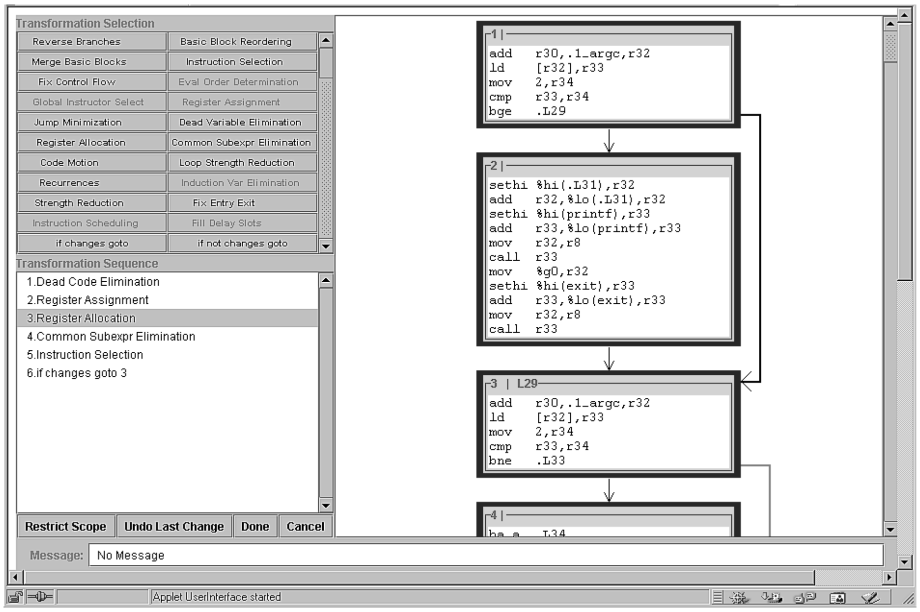

Fig. 3. User Interface for Specifying a Sequence of Optimization Phases

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 manner in which the scope can be restricted depends on the type of optimization phase applied. For instance, the loop optimization phases that include *minimize loop jumps*, *loop invariant code motion*, *loop strength reduction*, *recurrence elimination*, and *induction variable elimination*, are restricted by specifying the loops in which these phases are to be applied. The *register allocation* phase is restricted by specifying the set of variables that could be allocated to registers. Several phases are restricted by selecting a set of basic blocks (e.g. *branch chaining*) and other optimizations are restricted by specifying a set of instructions (e.g. *local instruction selection*). 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 shows the interface used to support specifying code-improving transformations by hand. The user selects an instruction with the 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

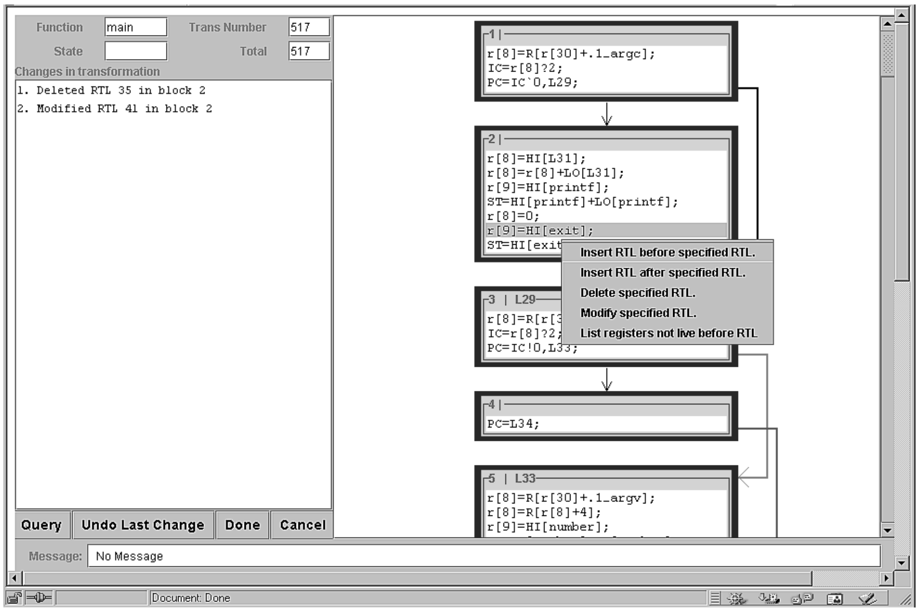

Fig. 4. User Interface for Specifying a Transformation by Hand

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 [18].

Fig. 5 shows an example of the improvement that can be obtained by specifying a transformation by hand. Fig. 5(a) depicts a loop in the *whetstone* benchmark and Fig. 5(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. Fig. 5(c) shows that the call and four 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 only if necessary) and sends it back to user interface to be displayed. Thus, the compiler can be used to help ensure that the changes associated with hand-

```
bc2:   for (i = 1; i <= n11; i++)
          x = sqrt(exp(log(x)/t1));
```

(a) Loop in Whetstone

```
       ...
       st     %f0,[%sp+68]
       st     %f1,[%sp+72]
       ld     [%sp+68],%o0
       ld     [%sp+72],%o1
       call   sqrt
       ...
```

(b) Generated Code for the sqrt Call

```
       ...
       fsqrtd %f0,%f0
       ...
```

(c) After Hand-Specifing Changes

Fig. 5. Example of a Hand-Specified Transformation

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 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 systems 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 backtracking of transformations is accomplished using the (|<, <<, <) buttons, as previously shown in Fig. 2. 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.

5 Implementation Issues

The user interface was implemented using Java to enhance its portability. We used Java 1.2, which includes the Java Swing user interface toolkit that is used to create graphical user interfaces. The aspects of the interface that limits its speed are the displaying of information and the communication with the compiler. Thus, we have found that 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.

There were numerous modifications to *vpo* that were required to support interactive compilation. First, the high-level function 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 shows the revised logic used for responding to user requests. A user can request to apply a sequence of optimization phases, can specify a transformation by hand, can undo previously applied transformations, or can query the compiler for some information about the current program representation.

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 *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

```
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
      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.
         ...
         END SWITCH
         Mark analysis invalidated by current phase.
         pc += 1.
      END WHILE
   ELSE IF user selected a transformation change THEN
      Perform transformation 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
```

Fig. 6. Algorithm for Performing User Requests on a Function

performing the optimization phase, *vpo* marks which analysis could possibly be invalidated by the current phase. Identifying the analysis needed for each optimization and 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. *fix entry, exit*), 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).

Each change associated with a hand-specified transformation has to be reflected in the program representation in the compiler. If an instruction was inserted or modified, the RTL or assembly instruction specified had to be converted to the encoded RTL representation used in the compiler. We developed two translators for this purpose. The first translator converts a human generated RTL into an encoded RTL. The second translator transforms an assembly instruction into an encoded RTL. After obtaining the encoded RTL, we used the machine description in the compiler to check if the instruction specified was legal for the machine. Before performing the change, the compiler has to check if the change requested is valid. 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 user requested change does not cause the program representation to be in an invalid state, then the change is performed by the compiler.

A user can also request to undo one or more previously applied transformations. In order to accomplish such requests, both the compiler and the user interface keep a history of changes to the program representation. The left portion of Fig. 7 depicts a linked list of a history of changes. All changes (additions and deletions) to the linked list occur at the tail. The user interface allows a user to apply or undo transformations using this linked list. Enough information regarding each change must be saved so its effect can be undone. For instance, if a change reflects a modification to an instruction, then the compiler and user interface must save the old version of the instruction before the modification so its effect can be reversed if requested by the user. The user interface updates its representation of the program shown in the right portion of Fig. 7 to reflect applying or undoing transformations. The current point in the left portion of Fig. 7 indicates the point in the history of transformations that is kept in the user interface where the user is viewing the representation. If a user selects a different phase or transformation to perform when he/she has not currently applied all of the transformations received, then the compiler is instructed to undo the appropriate number of transformations to reflect the current representation shown in the user interface.

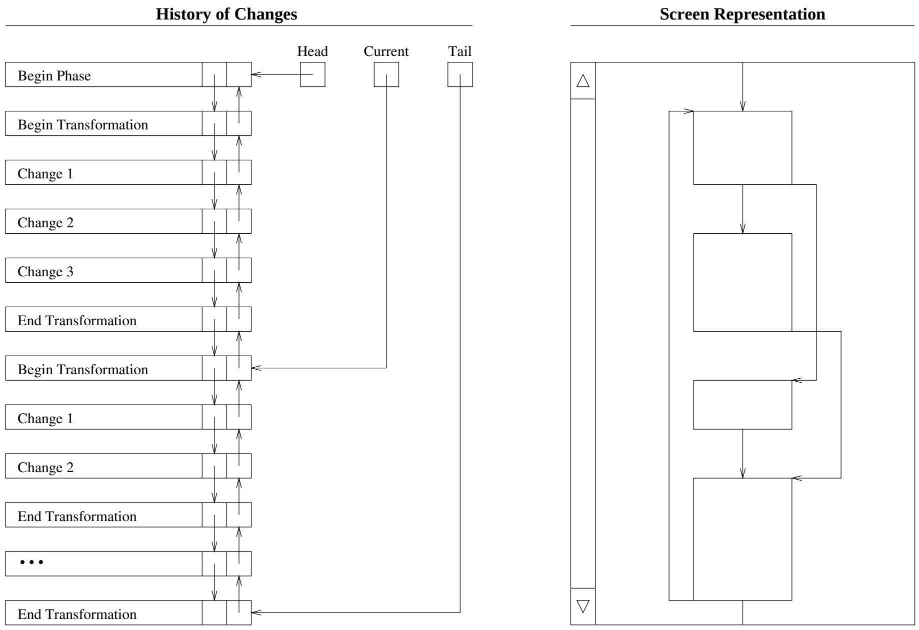

Fig. 7. Data Structures Used for Undoing Transformations

6 Current Status of the Implementation

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, and undo previously applied transformations in the compiler. There are still some features that have yet to be implemented in the user interface.

The user cannot currently select to limit the scope in which an optimization phase will be applied. However, we have implemented the ability to limit the scope of optimization phases in the compiler. We still need to modify the compiler to provide the user interface with the appropriate information (e.g. list of loops for a loop optimization phase, list of variables for a register allocation phase, etc.) and design and implement the interface in which this information is presented to the user.

A user cannot currently request queries about the state of the program representation. We envision a number of useful queries, such as live variable information and symbolic expansion of expressions at given points in the control flow, information about loops, etc. Such requests will be useful when specifying transformations by hand or deciding which optimization phases to perform next. 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.

Finally, a user currently has to perform a compilation of a file in a single session. We plan to eventually allow a user to end a session, have the sequence of changes applied to the program representation saved in a configuration file, and allow a user to resume a session by reading the configuration file back into the compiler at some future time.

[Reviewers: We anticipate completing the implementation of the remaining capabilities by the time the paper is presented and to provide a demonstration of the user interface at the conference.]

7 Conclusions

We have described a new interactive compilation paradigm that can help achieve the cost/performance tradeoffs that are needed for embedded applications. We have implemented this system 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 of the optimization phases, support the hand specification of code-improving transformations, and undo previously applied transformations. 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.

References

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

[2] B. Appelbe, K. Smith, and C. McDowell. Start/pat: a Parallel-Programming Toolkit. *IEEE Software*, 6:29–40, 1988.

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

[4] M. E. Benitez and J. W. Davidson. A Portable Global Optimizer and Linker. In *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pages 329–338, June 1988.

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

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

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

[8] J. Browne, K. Sridharan, J. Kiall, C. Denton, and W. Eventoff. Parallel Structuring of Real-Time Simulation Programs. In *COMPCON Spring '90: Thirty-Fifth IEEE Computer Society International Conference*, pages 580–584, 1990.

[9] K. Cooper, P. Schielke, and D. Subramanian. Optimizing for Reduced Code Space Using Genetic Algorithms. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embedded Systems*, pages 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. In *Proceedings of Supercomputing*, pages 194–203, 1992.

[12] Christine Eisenbeis and Sylvain Lelait. LoRA: a Package for Loop Optimal Register Allocation. In *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. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 45–58, June 1991.

[16] D. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. Parafrase-2: an Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 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] R. van Engelen, D. Whalley, and X. Yuan. Automatic Validation of Code-Improving Transformations. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embedded Systems*, June 2000.