VISTA: VPO Interactive System for Tuning Applications

PRASAD KULKARNI, WANKANG ZHAO, STEPHEN HINES, DAVID WHALLEY,
XIN YUAN, ROBERT VAN ENGELEN and KYLE GALLIVAN
Florida State University
JASON HISER and JACK DAVIDSON
University of Virginia
BAOSHENG CAI
Oracle Corporation
MARK BAILEY
Hamilton College
HWASHIN MOON and KYUNGHWAN CHO
Korea Advanced Institute of Science & Technology
and
YUNHEUNG PAEK
Seoul National University

Software designers face many challenges when developing applications for embedded systems. One major challenge is meeting the conflicting constraints of speed, code size, and power consumption. Embedded application developers often resort to hand-coded assembly language to meet these

Authors' addresses: P. Kulkarni, W. Zhao, S. Hines, D. Whalley, X. Yuan, R. van Engelen, and K. Gallivan, Computer Science Department, Florida State University, Tallahassee, FL 32306-4530; email: {kulkarni,wankzhao,hines,whalley,xyuan,engelen,gallivan}@cs.fsu.edu; J. Hiser and J. Davidson, Computer Science Department, University of Virginia, Charlottesville, VA 22904; email: {hiser,jwd}@cs.virginia.edu; B. Cai, Oracle Corporation, 4OP 955 Oracle Parkway, Redwood City, CA 94065; email: baosheng.cai@oracle.com; M. Bailey, Computer Science Department, Hamilton College, 198 College Hill Road, Clinton, NY 13323; email: mbailey@hamilton.edu; H. Moon, and K. Cho, Electrical Engineering Department, Korea Advanced Institute of Science & Technology, 373-1 Guseong-dong, Yuseong-gu, Daejeon 305-701, Korea; Y. Paek, School of Electrical Engineering, Seoul National University, Kwanak-gu, Seoul 151-744, Korea; email: ypaek@snu.ac.kr.

constraints since traditional optimizing compiler technology is usually of little help in addressing this challenge. The results are software systems that are not portable, less robust, and more costly to develop and maintain. Another limitation is that compilers traditionally apply the optimizations to a program in a fixed order. However, it has long been known that a single ordering of optimization phases will not produce the best code for every application. In fact, the smallest unit of compilation in most compilers is typically a function and the programmer has no control over the code improvement process other than setting flags to enable or disable certain optimization phases. This paper describes a new code improvement paradigm implemented in a system called VISTA that can help achieve the cost/performance trade-offs that embedded applications demand. The VISTA system opens the code improvement process and gives the application programmer, when necessary, the ability to finely control it. VISTA also provides support for finding effective sequences of optimization phases. This support includes the ability to interactively get static and dynamic performance information, which can be used by the developer to steer the code improvement process. This performance information is also internally used by VISTA for automatically selecting the best optimization sequence from several attempted. One such feature is the use of a genetic algorithm to search for the most efficient sequence based on specified fitness criteria. We include a number of experimental results that evaluate the effectiveness of using a genetic algorithm in VISTA to find effective optimization phase sequences.

Categories and Subject Descriptors: H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Graphical user interfaces*; D.2.6 [**Software Engineering**]: Programming Environments—*Interactive environments*; D.3.4 [**Programming Languages**]: Processors—*Compilers, optimization*; D.4.7 [**Operating Systems**]: Organization and Design—*Realtime systems and embedded systems, interactive systems*

General Terms: Performance, Measurement, Experimentation, Algorithms

Additional Key Words and Phrases: User-directed code improvement, interactive compilation, phase ordering, genetic algorithms

---

1. INTRODUCTION

The problem of automatically generating acceptable code for embedded microprocessors is often much more complicated than for general-purpose processors. First, 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 size and power consumption are often more critical than speed. Often, the conflicting constraints of speed, code size, and power consumption are managed by the software designer writing and tuning assembly code. Unfortunately, the resulting software is less portable, less robust (more prone to errors), 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 code improvement and code generation difficult [Liao et al. 1999; Marwedel and Goossens 1995]. While some progress has been made in developing compilers and embedded software development tools, many 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 domain.

One issue that limits the effectiveness of traditional optimizing compilers is that they apply the code-improving transformations to a program in a fixed

order. However, it is well-known that a single fixed sequence of optimization phases cannot produce the best code for all applications on a given machine [Vegdahl 1982]. Whether or not a particular optimization enables or disables opportunities for subsequent optimizations is difficult to predict, since it depends on the application being compiled, the previously applied optimizations, and the target architecture [Whitfield and Soffa 1997]. Current compiler technology also does not offer the user much flexibility in tuning the application. The only control they often provide is in the form of compilation flags, which could be used to turn certain optimization phases on and off.

In this paper we describe a new code-improvement paradigm, implemented in a system called VISTA, which allows the user to interactively tune the application and steer the optimization process in an attempt to achieve the cost/performance trade-offs (size, power, speed, cost, etc.) demanded for embedded applications. VISTA includes many features which allow users to:

1. view the low-level program representation in a graphical display throughout the compilation process,
2. direct the order and scope of optimization phases,
3. manually specify code transformations,
4. browse through and undo previously applied transformations,
5. automatically obtain performance information,
6. obtain other static analysis information on demand, and
7. automatically search for effective optimization phase sequences.

These features, along with several other convenient facilities, make VISTA a very robust compilation environment for tuning embedded applications.

This paper is organized as follows. In Section 2, we point the reader to related work in the area of alternative compiler paradigms, compiler user interfaces, and aggressive compilation techniques. In Section 3, we outline the VISTA framework and the information flow in VISTA. We describe in Section 4 the functionality of VISTA along with the implementation. Later, in Section 5, we show the results of a set of experiments that illustrate the effectiveness of using the genetic algorithm for automatically finding effective optimization sequences. In Section 6, we provide a case study, which shows how manual and automatic tuning can be combined in VISTA to achieve better overall performance. We devote Section 7 to discuss some interesting implementation issues. In Section 8 we discuss directions for future work. Finally, in Section 9, we state our conclusions.

2. RELATED WORK

There exist systems that are used for simple visualization of the compilation process. The *UW Illustrated Compiler* [Andrews et al. 1988], also known as *icomp*, has been used in undergraduate compiler classes to illustrate the compilation process. The *xvpodb* system [Boyd and Whalley 1993, 1995] has been used to illustrate low-level code transformations in the VPO compiler system [Benitez and Davidson 1988]. 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 transformations.

Other researchers have developed systems that provide interactive compilation support. These systems include the *pat* toolkit [Appelbe et al. 1989], the *parafrase-2* environment [Polychronopoulos et al. 1989], the *e/sp* system [Browne et al. 1990], a visualization system developed at the University of Pittsburgh [Dow et al. 1992], and *SUIF explorer* [Liao et al. 1999]. These systems provide support by illustrating the possible dependencies that may prevent parallelizing transformations. A user can inspect these dependencies and assist the compilation system by indicating if a dependency can be removed. In contrast, VISTA does not specifically deal with parallelizing transformations, but instead supports low-level transformations and user-specified changes, which are needed for tuning embedded applications in general.

A few low-level interactive compilation systems have also been developed. One system, which is coincidentally also called *VISTA* (visual interface for scheduling transformations and analysis), allows a user to verify dependencies during instruction scheduling that may prevent the exploitation of instruction level parallelism in a processor [Novack and Nicolau 1993]. Selective ordering of different optimization phases does not appear to be an option in their system. The system that most resembles our work is called *VSSC* (visual simple-SUIF compiler) [Harvey and Tyson 1996]. It allows optimization phases to be selected at various points during the compilation process. It also allows optimizations to be undone, but unlike our compiler, only at the level of complete optimization phases as opposed to individual transformations within each phase. Other features in our system, such as supporting user-specified changes and performance feedback information, do not appear to be available in these systems.

There has been prior work that used aggressive compilation techniques to improve performance. Superoptimizers have been developed that use an exhaustive search for instruction selection [Massalin 1987] or to eliminate branches [Granlund and Kenner 1992]. Iterative techniques using performance feedback information after each compilation have been applied to determine good optimization parameters (e.g., blocking sizes) for specific programs or library routines [Kisuki et al. 2000; Whaley et al. 2001; Knijnenburg et al. 2000]. A system using genetic algorithms to better parallelize loop nests has been developed and evaluated [Nisbet 1998]. These systems perform source-to-source transformations and are limited in the set of optimizations they apply. Selecting the best combination of optimizations by turning on or off optimization flags, as opposed to varying the order of optimizations, has been investigated [Chow and Wu 1999]. A low-level compilation system developed at Rice University uses a genetic algorithm to reduce code size by finding efficient optimization phase sequences [Cooper et al. 1999]. However, this system is batch oriented instead of interactive and concentrates primarily on reducing code size and not execution time and is designed to use the same optimization phase order for all of the functions within a file.
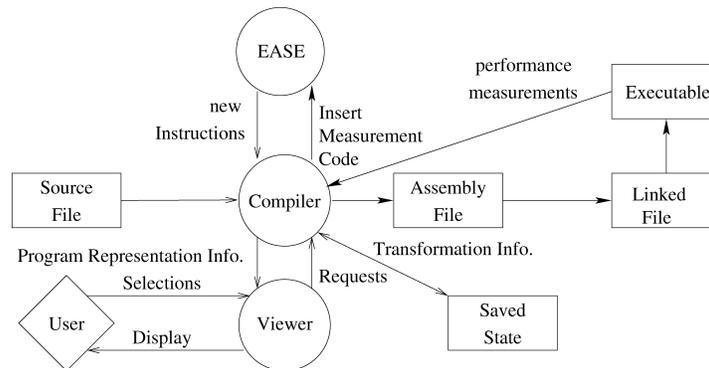
Fig. 1.   Interactive code improvement process.

3. THE VISTA FRAMEWORK

In this section we summarize the VISTA framework. This includes an overview of the information flow in VISTA along with a brief introduction of the optimization engine and visualization environment used.

3.1 Dataflow in VISTA

Figure 1 illustrates the flow of information in VISTA, which consists of a compiler and a viewer. The programmer initially indicates a file to be compiled and then specifies requests through the viewer, which include sequences of optimization phases, user-defined transformations, queries, and performance measures. The compiler performs the actions associated with the specified requests and sends the program representation information back to the viewer. In response to a request to obtain performance measures, the compiler, in turn, requests EASE to instrument the assembly with additional instructions to obtain the dynamic instruction counts. When executed, this instrumented code returns the performance measures. Similarly, it is possible to obtain execution cycles via simulation. VISTA can also provide static measures. While code size can be easily obtained, we have also provided WCET (worst-case execution time) information by invoking a timing analyzer, instead of invoking EASE [Zhao et al. 2004]. When the user chooses to terminate the session, VISTA writes the sequence of transformations to a file so they can be reapplied at a later time, enabling future updates to the program representation.

The viewer and the back-end compiler are implemented as separate processes in VISTA. The communication between the compiler and the viewer takes place via sockets. Separating the compiler and viewer as distinct processes provides additional flexibility. Thus, in theory, by modifying a compiler to send the appropriate information to the viewer, it should be possible to connect any compiler as the back-end for the viewer. It is also possible, for instance, to save the sequence of change messages sent from the compiler to the viewer and use a simple simulator 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 viewer, which can be used to support batch mode experimentation with different phase orderings. We were also concerned that the amount of memory used by the compiler and the viewer may be excessive for a single process. Separating the compiler and the viewer into separate processes allows users to access the interactive code improvement system on a different machine from which the compiler executes. By sending short messages and using low-overhead sockets for the actual communication, we have observed that communication is rarely the bottleneck, even when the two processes are located hundreds of miles apart and communication is over the Internet.

The VISTA system is designed to optimize code at a low level in a compiler back-end. The compiler front-end currently only supports the translation of the high-level language "C" to intermediate code, which is recognized by VISTA. The compiler back-end can, in turn, produce code for many different machines, like the SPARC, ARM, x86, Java, etc. If we are able to translate a different high-level language, such as C++ or Java, to our intermediate language that can be input to our back-end, VPO, then we can tune such applications as well. However, applications in many object-oriented languages have smaller functions and VISTA is currently designed to tune a single function at a time. Thus other optimizations, such as inlining, may be required to be performed at a higher level before a user can effectively tune such applications with VISTA.

3.2 VISTA's Optimization Engine

VISTA's optimization engine is based on VPO, the very portable optimizer [Benitez and Davidson 1988, 1994]. 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 intermediate representation called RTLs (register transfer lists). An RTL is a machine- and language-independent representation of a machine-specific operation. The comprehensive use of RTLs in VPO has several important consequences. Because there is a single representation, VPO offers the possibility of applying analyses and code transformations repeatedly and in an arbitrary order. 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 code improvement is important for embedded systems, because it is a viable approach for realizing compilers that produce code that effectively balances target-specific constraints, such as code size, 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 compilers targeting embedded microprocessors where chip manufacturers provide many different variants of the same base architecture and some chips have application-specific designs.

A third property of VPO is that it is easily extended to handle new architectural features. Extensibility is also important for compilers targeting 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 (data-flow analysis, control-flow analysis, etc.) are designed so that information is easily extracted and updated. This property makes writing new code improvement phases easier and it allows the information collected by the analyzers to be obtained for display.

3.3 EASE

The EASE (environment for architectural study and experimentation) [Davidson and Whalley 1991] environment is used in VISTA to collect static and dynamic performance measures to evaluate the improvements made in reducing the code size and the number of instructions executed. EASE collects these measures by instrumenting the assembly code generated by the compiler by additional instructions, which increment a counter each time a basic block is executed. These counts are multiplied by the number of instructions in each basic block to determine the total number of instructions executed.

Such performance feedback is important, as it gives the user a better perspective as to whether the sequence of optimizations applied is giving the expected benefits or if the user should roll back the changes and try some different sequence. The measures also indicate the portions of the code, which are more frequently executed, so the user could focus his/her attention on improving that portion of the program. It also helps VISTA automatically determine the best sequence of optimizations for a function.

4. FUNCTIONALITY OF VISTA

The VISTA framework supports the following features. First, it allows a user to view a low-level graphical representation of the function being compiled, which is much more convenient than extracting this information from a source-level debugger. Second, a user can select the order and scope of optimization phases. Selecting the order of optimization phases may allow a user to find a sequence of phases that is more effective for a specific function than the default optimization phase order. Limiting the scope of the optimization phases allows a user to allocate resources, such as registers, for the critical regions of the code. Third, a user can manually specify transformations. This feature is useful when exploiting specialized architectural features that cannot be exploited automatically by the compiler. Fourth, a user can undo previously applied transformations or optimization phases. This feature eases experimentation with alternative phase orderings or user-defined transformations. Fifth, VISTA supports the ability to obtain program performance measures at any time during the code improvement process. Such feedback would be very useful to a user interactively tuning the code. Sixth, VISTA provides the ability to find effective optimization sequences automatically. Finally, the user can query the compiler for other static information, like dominator or live register information, which may aid the user in better tuning the application. In this section, we describe the functionality of VISTA and provide details on how these features were actually implemented.
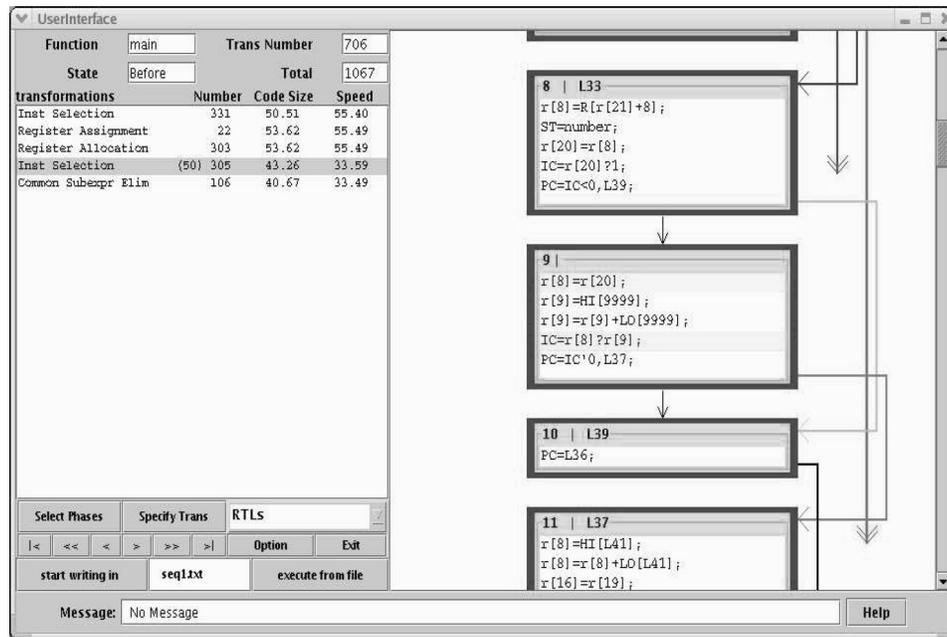
Fig. 2.   Main user interface window in VISTA.

4.1 Viewing the Low-Level Representation

Figure 2 shows a snapshot of the VISTA viewer that supports interactive code improvement. The program representation appears in the right window of the viewer and is shown as basic blocks in a control-flow graph. Within the basic blocks are machine instructions. The programmer may view these instructions as either RTLs or assembly code. In addition, VISTA provides options to display additional information about the program that a programmer may find useful.

The left window varies depending on the mode selected by the user. Figure 2 shows the default display mode. The top left of the viewer screen shows the name of the current function and the number of the transformation currently being displayed. A transformation consists of a sequence of changes that preserve the semantics of the program. The viewer can display a transformation in either the before or after state. In the before state, the transformation has not yet been applied. However, the instructions that are 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 and easily understand the effects of each transformation. Figures 2 and 3 show the before and after states during a transformation in *instruction selection*. Before and after states are used in other graphical compilation viewers [Boyd and Whalley 1993, 1995]

The bottom left window contains the viewer control panel. The ">," ">>," and "> |" buttons allow a user to advance through the transformations that were performed. The ">" button allows a user to display the next transformation. A user can display an entire transformation (before and after states) with two
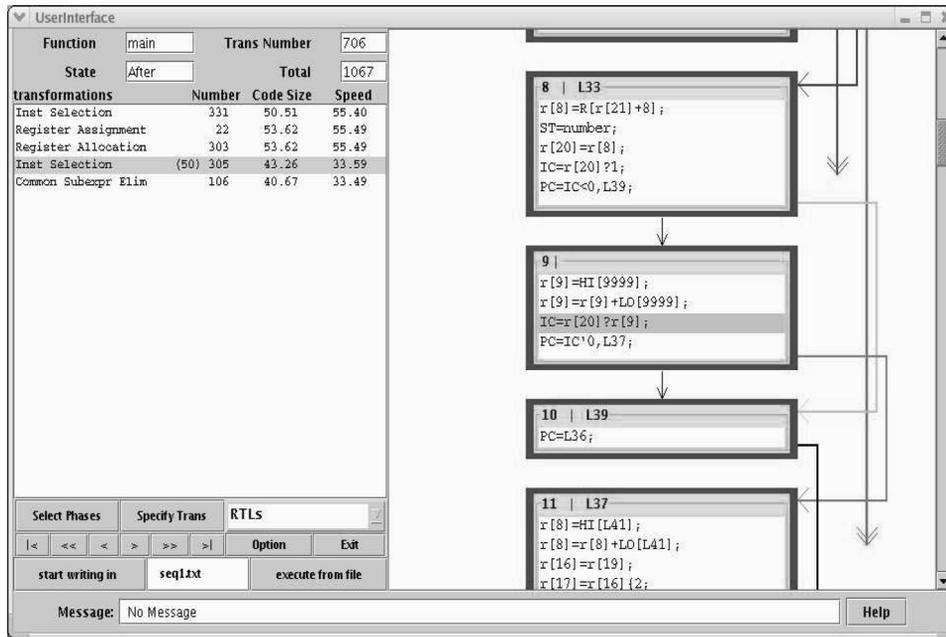
Fig. 3. After state in the main user interface window in VISTA.

clicks of this button. The ">>" button allows a user to advance to the next optimization phase. A phase is a sequence of transformations applying the same type of transformation. The "> |" button allows the user to advance beyond all transformations and phases. The "| <," "<<," and "<" buttons allow the user to back through the transformations with corresponding functionality. These buttons provide the ability to navigate through the changes made to the program by the compiler. These navigation facilities are also useful if the user needs to undo some transformations or phases, as described in Section 4.5. To enable the viewing of previous transformations, the viewer maintains a linked list of all messages sent to it by the compiler. Browsing through the transformations in the viewer only involves traversal of this linked list. Thus, after the compiler performs the transformations and sends the information to the viewer, viewing the transformations in the viewer is done locally with no messages exchanged between the compiler and the viewer.

Also shown in the left window is the history of code improvement phases that the compiler has performed, which includes phases that have been applied in the viewer and the phases yet to be applied by the viewer. We have found that displaying the list of code improvement phases in this manner provides a user some context to the current state of the program.

When viewing a program, the user may also change the way the program is displayed and may query the compiler for additional information. As mentioned earlier, the program representation may be in RTLs or assembly. Displaying the representation in RTLs may be preferred by compiler writers, while assembly may be preferred by embedded systems application developers who are familiar

(a) RTLs Displayed   (b) Assembly Displayed   (c) Control-flow-Only Displayed
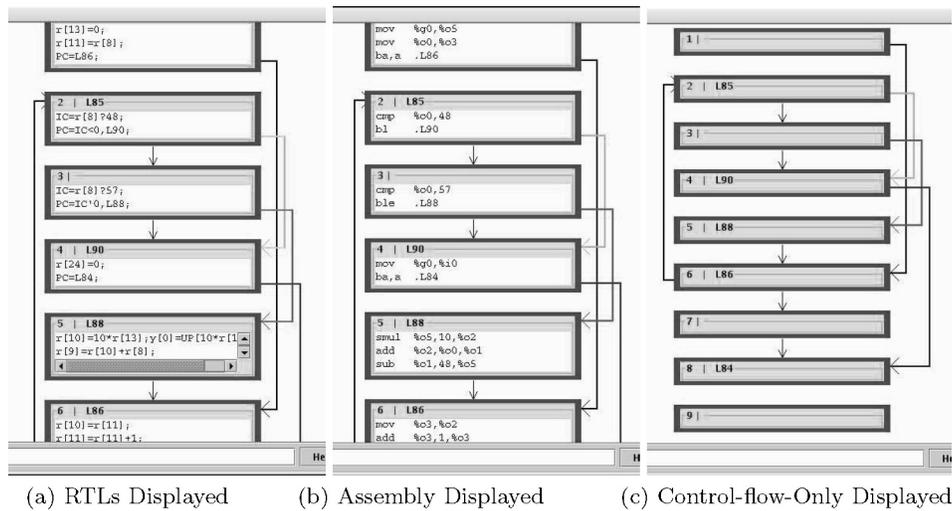
Fig. 4.   Three different program representations in VISTA.

with the assembly language for a particular machine. When control flow is of more interest than the specific machine instructions, the user can choose to display only the basic block structure without the machine instructions. This makes it possible to display more basic blocks on the screen and gives the user a higher-level view of the function. Figure 4 shows the RTL, assembly, and control flow-only representation of a function in VISTA.

We implemented the viewer using Java to enhance its portability. We used Java 2, which includes the Java Swing toolkit that is used to create graphical user interfaces. The aspects of the interface that limit 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.

4.2 Selecting the Order and Scope of Optimization Phases

Generally, a programmer has little control over the order in which a typical compiler applies code improvement phases. Usually the programmer may only turn a compiler code improvement phase on or off for the entire compilation of a file. For some functions, one phase ordering may produce the most suitable code, while a different phase ordering may be best for other functions. VISTA provides the programmer with the flexibility to specify what code improvements to apply to a program region and the order in which to apply them. A knowledgeable embedded systems application developer can use this capability for critical program regions to experiment with different orderings in an attempt to find the most effective sequence.

We also find it useful to conditionally invoke an optimization phase based on whether a previous optimization phase caused any changes to the program representation. The application of one optimization phase often provides opportunities for another optimization phase. Such a feature allows a sequence
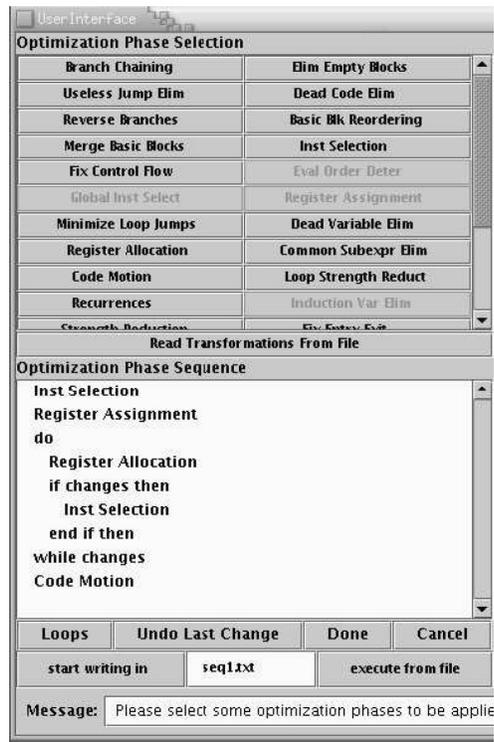
Fig. 5.   Interactively selecting optimization phases.

of optimization phases to be applied until no further improvements are found. Likewise, an optimization phase that is unlikely to result in code-improving transformations unless a prior phase has changed the program representation, can be invoked only if changes occurred, which may save compilation time. VISTA supports such conditional compilation by providing four structured control statements (*if-changes-then, if-changes-then-else, do-while-changes, while-changes-do*). These structured statements can also be nested.

Consider the interaction between register allocation and instruction selection optimization phases. Register allocation replaces load and store instructions with register-to-register move instructions, which provides opportunities for instruction selection. Instruction selection combines instructions together and reduces register pressure, which may allow additional opportunities for register allocation. Figure 5 illustrates how to exploit the interaction between these two phases with a simple example. The user has selected two constructs, which are a *do-while-changes* statement and a *if-changes-then* statement. For each iteration, the compiler performs register allocation. Instruction selection is only performed if register allocation allocates one or more live ranges of a variable to a register. These phases will be iteratively applied until no additional live ranges are allocated to registers.

In order to communicate to VPO the sequence of optimizations phases to apply, the viewer translates the structured statements into a low-level sequence
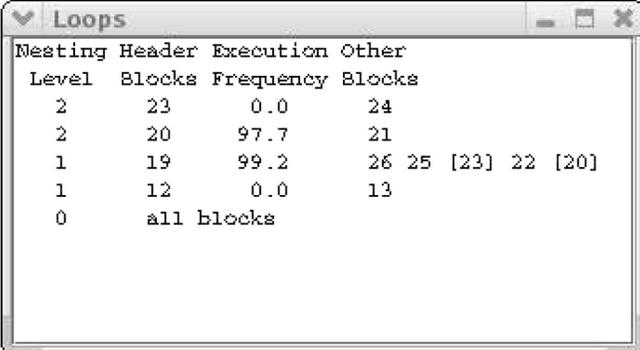
Fig. 6.   Loop report indicating member basic blocks.

of requests. This sequence is interpreted by VPO and each resulting change to the program representation is sent to the viewer. This process continues until a stop operation has been encountered. The operations to be performed by the selection shown in Figure 5 are as follows:

1. Perform instruction selection
2. Perform register assignment
3. Enter loop
4. Perform register allocation
5. If no changes in last phase goto 7
6. Perform instruction selection
7. If changes during loop iteration then goto 4
8. Exit loop
9. Perform loop-invariant code motion
10. Stop

As shown in the upper window of Figure 5, the user is prevented from selecting some of the phases at particular points in the compilation. This is because of compiler restrictions on the order in which it can perform phases. Although we have tried to keep these restrictions to a minimum, some restrictions are unavoidable. For instance, the compiler does not allow the register allocation phase (allocating variables to hardware registers) to be selected until the register assignment phase (assigning pseudoregisters to hardware registers) has been completed. The user may also only request that the compiler perform register assignment once.

In addition to specifying the order of the code improvement phases, a user can also restrict the scope in which a phase is applied to a region of code. This feature allows the allocation of resources, such as registers, based on what the user considers to be the critical portions of the program. The user can restrict the region to a set of basic blocks by either clicking on blocks in the right window or clicking on loops in a loop report similar to the one shown in Figure 6. The viewer marks the basic blocks in which the scope is to be restricted and sends
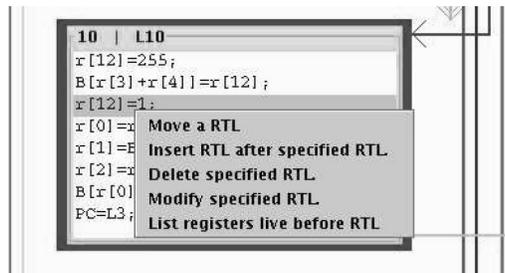
Fig. 7.   Manually specifying a transformation on the ARM.

this information to the compiler. Thus, before applying any optimization phase, the compiler first checks if the basic block is in scope. If not, then the block is skipped. This is safe to do for optimizations local to a block. When applying loop optimizations, if the header block is not included in the scope, then the loop is skipped. A few phases cannot have their scope restricted because of the method in which they are implemented in the compiler or how they interact with other phases (e.g., filling delay slots). Note that, by default, the scope in which a phase is applied is the entire function.

VPO also required several other modifications to support interactive code improvement. The original batch compiler had to be modified to listen to user requests and perform corresponding actions, as well as to send messages to the viewer. Each optimization phase in a compiler needs certain data and control-flow information about the function to do its work. For interactive compilation, we had to identify the analysis each phase needs and the analysis that each optimization phase invalidates. We also had to identify which phases were required during the compilation (e.g., *fix entry/exit*), which code improvement phases could only be performed once (e.g., *fill delay slots*), and the restrictions on the order in which code improvement phases could be applied.

4.3 User-Specified Code Transformations

Despite advances in code generation for embedded systems, knowledgeable assembly programmers can always improve code generated by current compiler technology. This is likely to be the case, because the programmer has access to information the compiler does not. In addition, many embedded architectures have special features (e.g., zero overhead loop buffers, modulo address arithmetic) not commonly available on general-purpose processors. Automatically exploiting these features is difficult because of 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 is not an attractive alternative because of the longer development time, higher likelihood of errors, and loss of portability. It is desirable to have a system that not only supports traditional compiler code improvement phases, but also supports the ability to manually specify transformations.

Figure 7 shows how VISTA supports the application of user-specified code transformations. When the user clicks on an instruction, the viewer displays

the list of possible user-specified changes for that instruction. As the user selects each change, the change is sent to the compiler. Transformations to basic blocks (insert, delete, label) are also possible. In response to each user-specified transformation, the compiler first verifies the syntax of that particular transformation. A number of semantic checks are also performed. 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. This ensures that the transformations do not violate some basic semantic constraints of the language. This is easy and fast to do, but, at the same time, it does not guarantee that all the semantics of the program are preserved. After performing this check, if the transformation is valid, then the compiler sends the appropriate change messages to the viewer so it can update the presented program representation.

Each change associated with a user-specified transformation must be reflected in the program representation in the compiler. If an instruction is inserted or modified, the RTL or assembly instruction specified must 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 use the machine description in the compiler to check if the instruction specified was legal for the machine. Before performing the change, the compiler checks if the change requested is valid. This check includes validating the syntax of the instruction, as well some semantics checks mentioned previously. The compiler only performs the change if it does not detect the change to be invalid.

The user can also query the compiler for information that may be helpful when specifying a transformation. For example, 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 viewer. Thus, the compiler can be used to help ensure that the changes associated with user-specified transformations are properly made and to guide the user in generating valid and more efficient code.

Providing user-specified transformations has an additional benefit. After the user has identified and performed a transformation, the optimization engine can be called upon to further improve the code. Such user-specified transformations may reveal additional opportunities for the optimization engine that were not available without user knowledge. In this way, the optimizer and user can, jointly, generate better code than either the user or the optimizer could have generated separately. In Section 6, we provide a case study which illustrates how user knowledge about the application and the machine can be used along with automatic program tuning to generate better overall code with less effort.

4.4 Automatic Pattern Recognition and Replacement

The ability to perform hand-specified transformations in VISTA is important as it allows a knowledgeable user to make changes, which could not be directly

exploited by any predefined compiler optimization. This feature is even more useful at the current time, where compiler writers are finding it increasingly difficult to keep pace with the rapidly evolving microprocessor industry. Similar program inefficiencies can occur multiple times in the same function. It would be extremely tiring for the user to find each occurrence of a particular type of program fragment and replace it with more efficient code at every place. What is needed here is a tool that will find all occurrences of some code fragment and replace it with the desired efficient code, as specified by the user. VISTA provides exactly this functionality, which is, in essence, an interactively pattern-driven peephole optimizer [Davidson and Whalley 1989].

We have included in VISTA a pattern-based peephole optimizer. The user can specify the search pattern as multiple strings of the form:

```
<search pattern>:<deads>:<list of registers>
```

The <search pattern> must be of the form [x=y;]+, which matches the form of an RTL in VISTA. "*x*" and "*y*" are, in turn, sequences of characters and arguments. The characters in the pattern must exactly match the characters in an RTL for the search to be successful. The arguments can match any sequence of characters in the RTL until the following character in the search pattern matches some literal in the RTL being compared. Each argument number in the search pattern is associated with the RTL characters it matches, and can be used later to substitute the same characters in the replacement pattern. In the <deads> field, we can specify the registers that must be dead in this RTL. The <list of registers> specifies which arguments must be registers. The last two fields in the search pattern are optional.

The replacement pattern is similarly constituted and must use the same arguments as the search pattern or other constant characters. The pattern-matching arguments in the searched RTL are used to construct the replacement RTL. The viewer scans every instruction in the function and attempts to match it with the search pattern. The matched sequence of instructions is highlighted in the viewer. Each argument in the search pattern in associated with an appropriate sequence of characters. The viewer then constructs the replacement RTLs by replacing all the arguments in the replacement pattern with their associated character sequence. These replacement instructions are then sent to the compiler to verify their syntax and semantics. If correct, the matched and replacement instructions are shown in the dialog boxes in the viewer and the user has the option of making the change. This process is illustrated in Figure 8. In cases where the syntax or semantics are detected as invalid by the compiler, an appropriate message is displayed in the dialog box and the user is not allowed to make any change (Figure 9).

Finally, we also provide an option to replace all matched sequences automatically. Here the viewer finds each instruction sequence matching the search pattern, constructs the replacement instructions, verifies their syntax and semantics, and then, if valid, automatically makes the replacement. The dialog box indicates the number of successfully replaced patterns in the function, as illustrated in Figure 10.
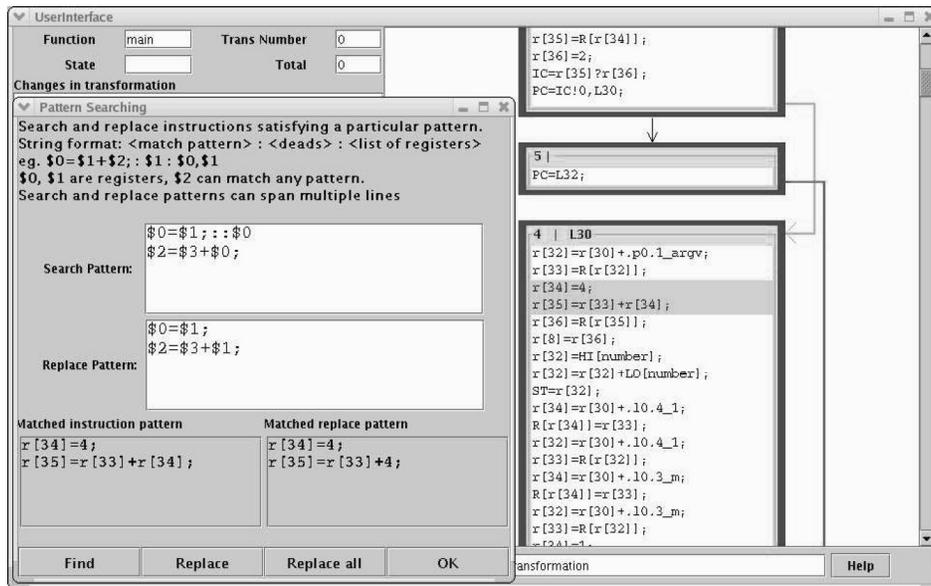
Fig. 8.   Valid pattern during pattern based search.
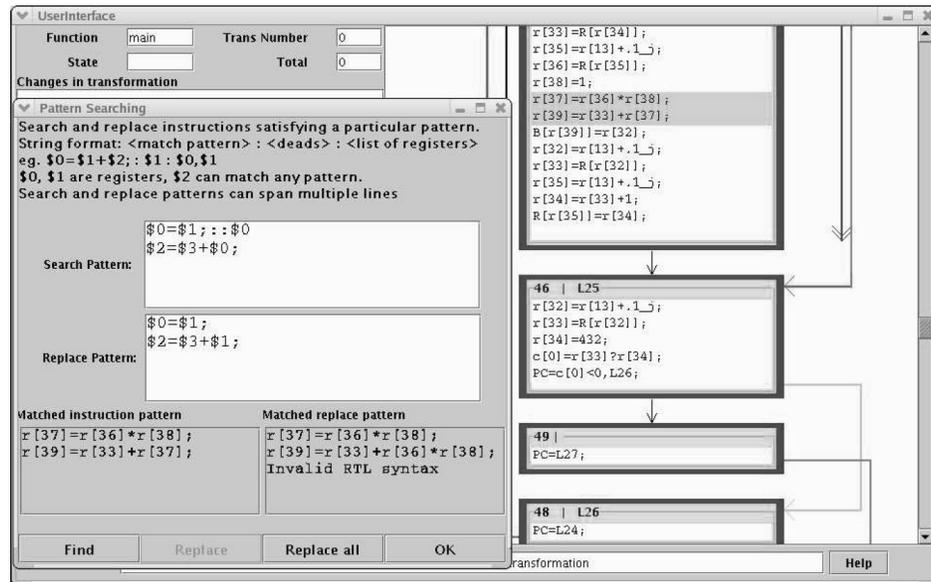
Fig. 9.   Invalid pattern during pattern based search.

4.4.1 *Prototyping New Code Improvements.*   The ability to manually specify low-level code transformations has another important, more general, application of user-specified transformations. Unlike high-level code transformations, it is difficult to prototype the effectiveness of low-level code transformations.
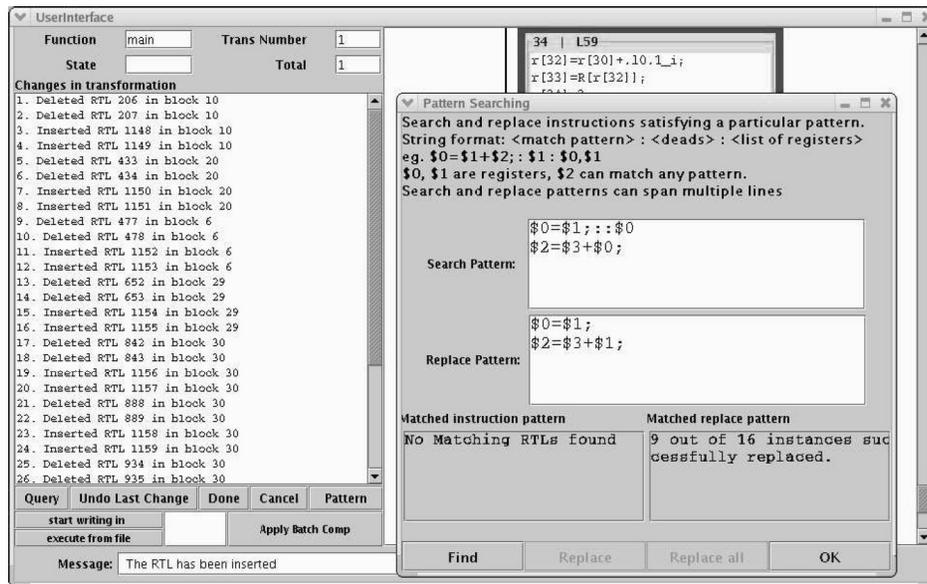
Fig. 10.   Replace all valid patterns during pattern based search.

There are two factors that make prototyping low-level code transformations difficult. First, many low-level code improvements exploit architectural features that are not visible in a high-level representation. For example, machine-dependent code improvements, such as allocating a variable to a register, do not have equivalent source code transformations. In such cases, the source code cannot be hand-modified to measure the effectiveness of the transformation. Second, low-level code transformations are often only fully effective when performed after other, specific transformations have been applied. For example, branch chaining may reveal additional opportunities for unreachable code elimination. For such cases, it may be possible to perform the transformation in source code, but it is not possible to prototype its effectiveness accurately at the source level, since opportunities will be missed.

One prototyping strategy is to generate low-level code, write it to a file, manually perform the code improvement, read the low-level code back in, and perform any additional code improvements. This process can only work if the code improver accepts the same representation it generates. Although VPO uses a single low-level representation, the RTLs it accepts use pseudoregisters, while the RTLs it generates use hardware registers. Often, there are other phase-order limitations that prevent the use of this strategy. By opening the code improvement process to user-specified changes, VISTA provides a general solution to prototyping and measuring the effectiveness of new low-level code transformations.

4.5 Undoing Transformations

An important design issue for an interactive code improvement system is the manner in which support is provided to an embedded systems application

developer to experiment with different orderings of phases and user-specified transformations in an attempt to improve the generated code. In order to support such experimentation, VISTA provides the ability for the user to reverse previously made decisions regarding phases and transformations that have been specified.

This reversing of transformations is accomplished using the ("| <," "<<," "<") buttons as shown in Figures 2 and 3. These buttons allow a user to view 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 code improvement phase. Likewise, the "| <" button allows the user to view the state of the program 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.

As mentioned in Section 4.1, the viewer maintains a linked list of all messages sent to it by the compiler. This list is distinct from a similar history of all previous program changes maintained by the compiler. At the end of one compilation session, the compiler dumps this list to a file. The next time the compiler is invoked for the same program, the compiler first reads from this file and applies all the transformations performed during past compilation sessions. A separate *transformation file* has to be maintained for each source file, containing only the changes made to all the functions in that file.

Before the user can invoke a code improvement phase or user-specified transformation while viewing a prior state of the program, the subsequent transformations must be removed. Thus, the user has the ability to permanently undo previously applied phases and transformations. When the compiler receives a request to undo previous transformations, it dumps the transformation list it maintains to a file. The current function with all its data structures are discarded and reinitialized. A fresh instance of the function is then read from the input file. This function has no optimizations applied to it at this time. The compiler then reads the saved transformation file and reapplies the previous changes, but only up to the point where the user had indicated that the transformations be discarded. The remaining changes are thus automatically removed.

The other approach tried was to keep enough state information regarding each change so that it could be rolled back. For instance, if a change reflects a modification to an instruction, then the compiler and viewer must save the old version of the instruction before modification so that its effect can be reversed if requested by the user. This approach was found to be difficult to maintain as it required complicated data structures and an unacceptably large amount of space to store the information needed to reverse each change. This made the code difficult to understand and less robust. As we already maintained a list of all program changes, we felt it unnecessary to maintain another list of all things to undo to reverse each change. In an interactive compilation environment, where the majority of the time spent is in between the user clicking buttons, the additional effort in reading the unoptimized function back in was found to be worth the reduced complexity in the compiler.

The ability to undo previously applied transformations is used extensively by VISTA when measuring program performance. 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 revoke changes to the program will facilitate the development of such transformations.

4.6 Visualizing Performance

An important requirement of interactive code improvement is the ability to measure and visualize the performance of the program being improved. It is only by examining the performance of the program that the user can understand the effectiveness of their user-specified code improvements and their choices in phase ordering. VISTA provides the mechanisms necessary to easily gather and view this information. VISTA supports obtaining both static and dynamic measurements. The static measure we used in our experiments is a count of the instructions in the function (i.e., the code size). For the dynamic measure, we most often use a crude approximation of the number of CPU cycles by counting the number of instructions executed, since this measure can be obtained with little overhead. We measure the dynamic instruction count by instrumenting the assembly code with instructions to increment counters using the EASE system [Davidson and Whalley 1991]. We have also used VISTA to obtain other performance measures like WCET [Zhao et al. 2004] and software simulation cycles [Burger and Austin 1997].

VISTA provides the user two options to determine program performance:

1. The first option indicates the current program performance, at the point when this option is selected. The compiler instruments, links and executes the program only once and sends the dynamic performance counts to the viewer. The viewer displays the relative frequency of execution of each basic block in the block's header, as seen in the right portion of Figure 11. This information indicates the frequently executed parts of the code, so that the user can concentrate resources on optimizing the critical portions of the code.
2. The second option displays the relative static and dynamic improvements made after each optimizing phase. When this option is selected, the compiler executes the program once and gets the baseline measures. After each phase, the program is again executed and static and dynamic counts obtained are compared with the baseline measures. The relative improvements made by each phase are displayed in the viewer, which are shown in the left portion of Figure 11. This information allows a user to quickly gauge the progress that has been made in improving the function. It is also very useful when it is important for the function to reach a particular performance goal.

To get the dynamic instruction counts, the compiler must link and execute the program. The commands to accomplish this can be specified by the user in a file. If this file is not present, then the user is prompted for this information before getting the measurements for the first time. The window that is opened to collect this information is shown in Figure 12. The *Actual Output File* is compared with the *Desired Output File* to verify that correct output is produced.
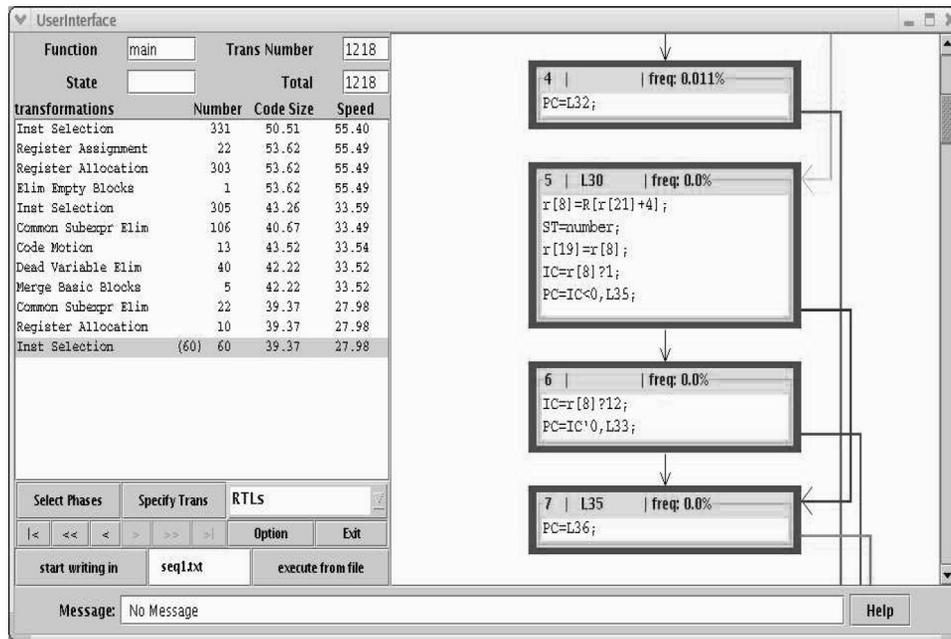
Fig. 11.   Getting performance measures in VISTA.
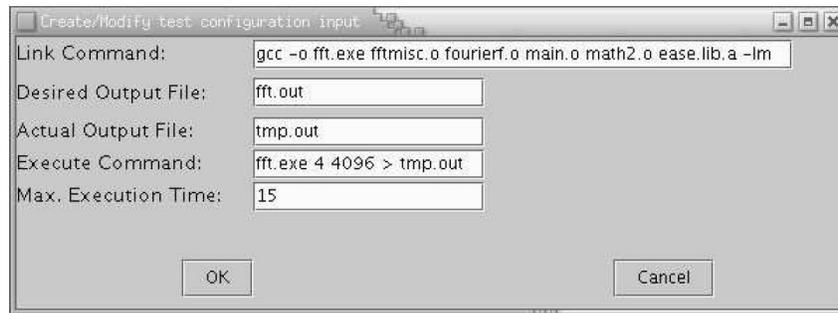
Fig. 12.   Test configuration window.

If the executable produced by the compiler goes into an infinite loop, then the execution is terminated after *Max. Execution Time*.

Figure 13 shows the steps the compiler performs to obtain the performance counts. At the start of the measurement process, the compiler needs to store some information needed to get back to the original state. This includes saving the assembly for the previous functions and global variables to a file, and the transformations applied to the current function, thus far. A pointer to the start of the current function in the input file is also stored. There are some required transformations that must always be performed before the assembly code for a function can be generated. These are *register assignment* (assigning pseudo registers to hardware registers) and *fix entry-exit* (fixing the entry and exit of the function to manage the run-time stack). Before generating assembly for
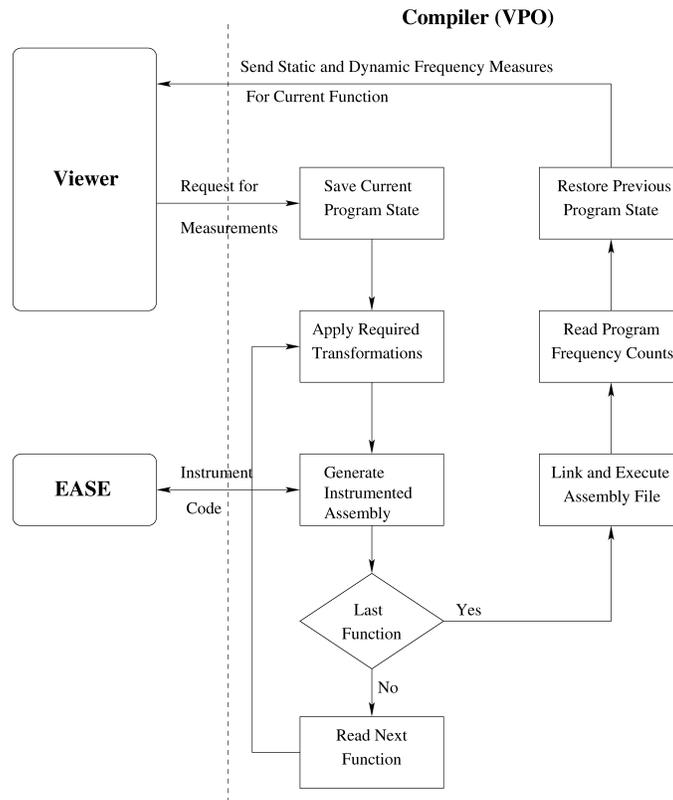
Fig. 13.   The measurement process.

measurements, VISTA checks if these phases have been applied and, if not, performs them.

The instrumentation of code by EASE is done as the next stage. The EASE environment instruments the code with additional instructions, including increments to counters associated with each basic block. The instrumented assembly is output to a file. After code for the function is generated, all the data structures holding the state of this function are cleared and reinitialized. VISTA then reads in the remaining functions in the file, applies the required transformations, and instruments each with EASE code to produce the final assembly file.

This instrumented assembly file is linked and executed. Upon execution, an information file that contains the number of instructions and the frequency of execution of each basic block in the function is generated. These counts are sent to the viewer for display. At this point, the compiler is holding the last function in the input file with EASE instrumentation applied to it. To get back to the point when measurements were initiated, the pointer to the start of the current function in the input file is used to reread the function. The compiler then reapplies the sequence of previously applied transformations to this function to reproduce the program representation at the point where measurements were taken.
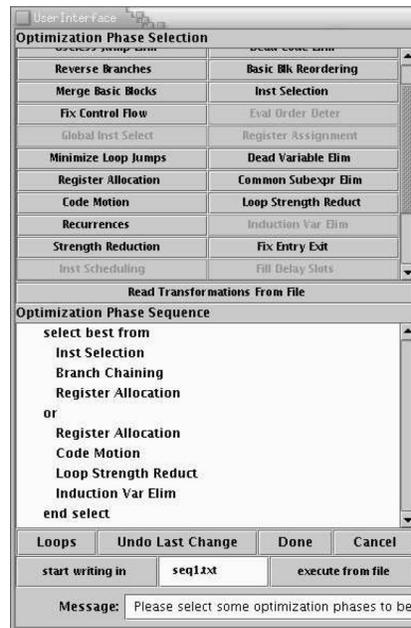
Fig. 14.   Selecting the best sequence from a specified set.

VISTA compiles functions within a file one at a time and each file is compiled separately. In order to reduce compilation overhead, the position in the assembly file at the beginning of the function is saved and the assembly is regenerated at that point. Thus, obtaining new measurements requires producing instrumented assembly for only the remaining functions in the file and assembling only the current file. VISTA also saves the position in the intermediate code file that is input to VPO and the position in the transformation file to further reduce I/O.

4.7 Performance-Driven Selection of Optimization Sequences

In addition to allowing a user to specify an optimization sequence, it is desirable for the compiler to automatically compare two or more sequences and determine which is most beneficial. VISTA provides two structured constructs that support automatic selection of optimization sequences. The first construct is the *select-best-from* statement and is illustrated in Figure 14. This statement evaluates two or more specified sequences of optimization phases and selects the sequence that best improves performance according to the selected criteria. For each sequence, the compiler applies the specified optimization phases, determines the program performance (instruments the code for obtaining performance measurements, produces the assembly, executes the program, and obtains the measurements), and reapplies the transformations to reestablish the program representation at the point where the *select-best-from* statement was specified. After the best sequence is found, the compiler reapplies that sequence.
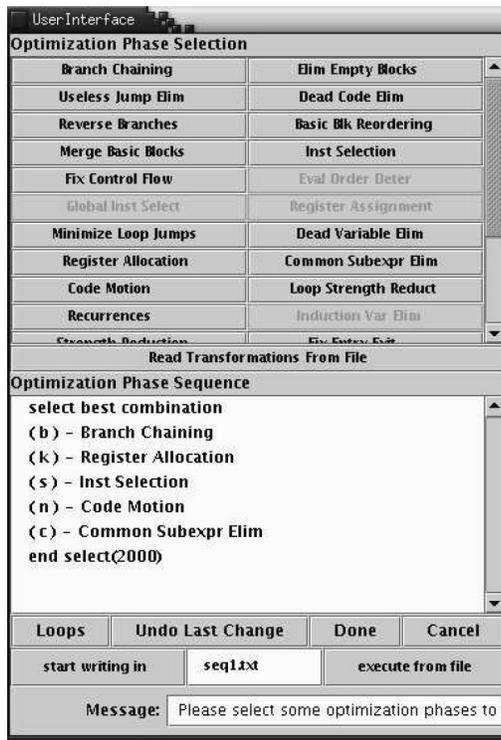
Fig. 15.   Selecting the best sequence from a set of optimization phases.
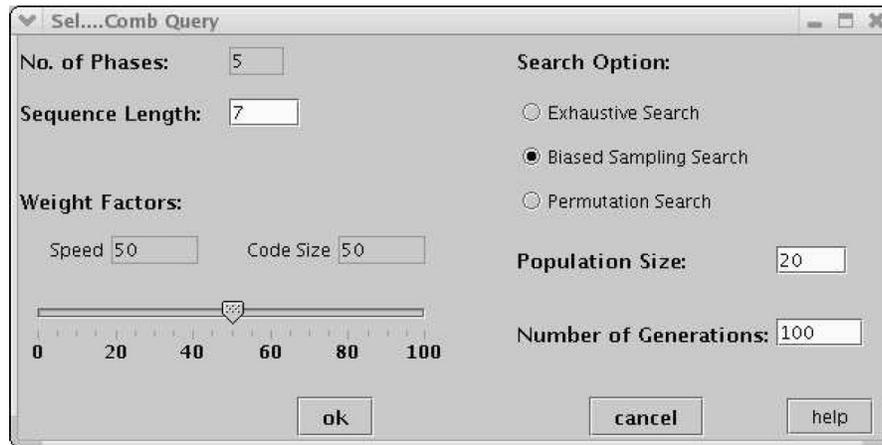
Fig. 16.   Selecting options to search the space of possible sequences.

The other construct, the *select-best-combination* statement, is depicted in Figure 15. This statement accepts a set of m distinct optimization phases and attempts to discover the best sequence for applying these phases. Figure 16 shows the different options that VISTA provides the user to control the search. The user specifies the *sequence length*, n, which is the total number of phases
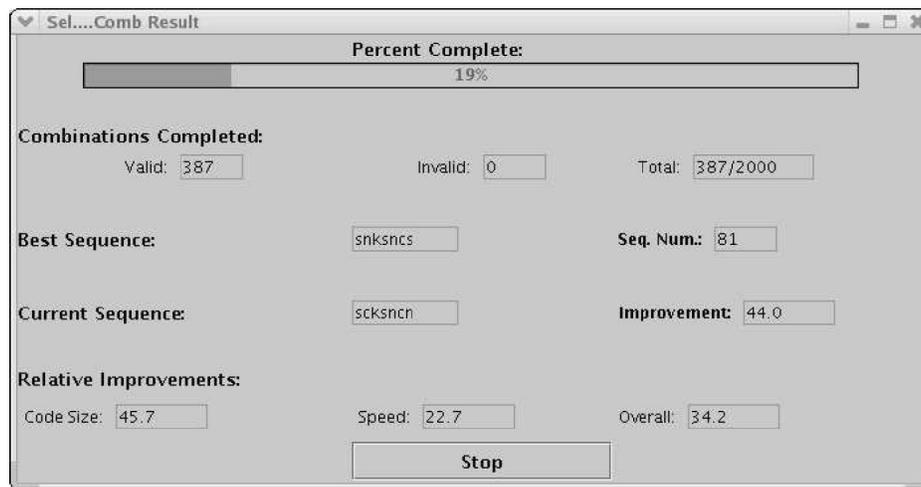
Fig. 17. Window showing the status of searching for an effective sequence.

applied in each sequence. An *exhaustive search* attempts all possible m^n sequences, which may be appropriate when the total number of possible sequences can be evaluated in a reasonable period of time. The *biased sampling search* applies a genetic algorithm in an attempt to find the most effective sequence within a limited amount of time. This type of search is appropriate when the search space is too large to evaluate all possible sequences. For this search, the number of different sequences in the population and the number of generations must be specified, which limits the total number of sequences evaluated. These terms are described in more detail later in the paper. The *permutation search* attempts to evaluate all permutations of a specified length. Unlike the other two searches, a permutation by definition cannot have any of its optimization phases repeated. Thus, the sequence length, n, must be less than or equal to the number of distinct phases, m. The total number of sequences attempted will be $m!/(m-n)!$ A permutation search may be an appropriate option when the user is sure that each phase should be attempted, at most, once. VISTA also allows the user to choose weight factors for instructions executed and code size, where the relative improvement of each is used to determine the overall improvement. When using the *select-best-from* statement, the user is also prompted to select a weight factor.

Performing these searches is often time-consuming. Thus, VISTA provides a window showing the current status of the search. Figure 17 shows a snapshot of the status of the search that was selected in Figures 15 and 16. The percentage of sequences completed along with the best sequence and its effect on performance is given. The user can terminate the search at any point and accept the best sequence found so far.

4.8 Presenting Static Program Information

The viewer in VISTA also can display a variety of static program analysis information to the user. Such information can prove to be very useful to a
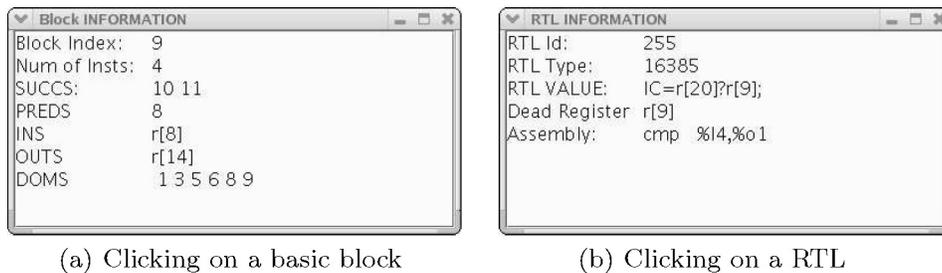
(a) Clicking on a basic block   (b) Clicking on a RTL

Fig. 18.   Getting static program information in VISTA.
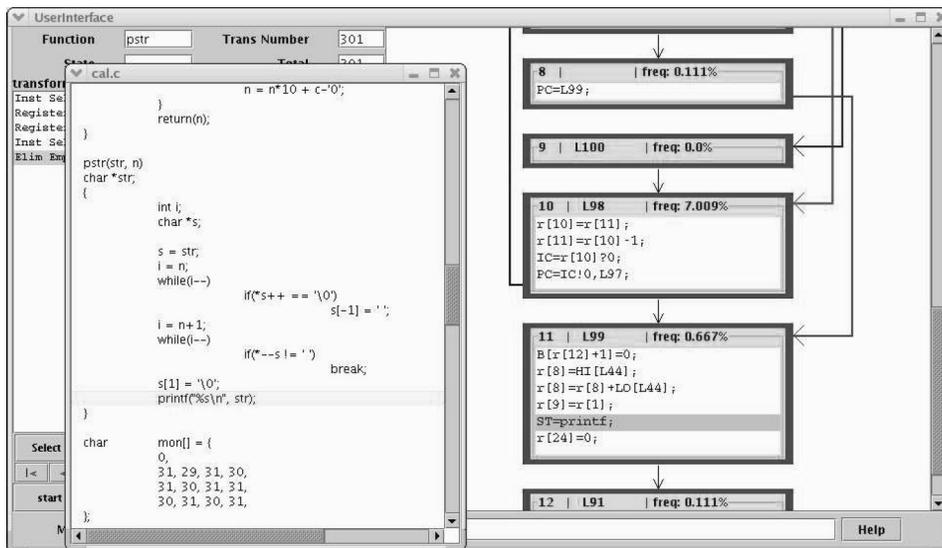
Fig. 19.   Clicking on a basic block highlights corresponding source code lines.

programmer interactively tuning the program. Clicking on a basic block in VISTA provides information, such as the live registers coming into and going out of the basic block, the dominators of that block, and the successors and predecessors of that block. Clicking on block 9 in Figure 3 displays the information shown in Figure 18a. Clicking on a RTL displays the assembly representation of the RTL and the dead registers at that point. Figure 18b shows the information displayed on clicking the highlighted RTL in Figure 3. As mentioned earlier, VISTA also has the ability to show the registers live at any point in the function.

In addition to such static information, double-clicking on a basic block or RTL in the viewer, opens up another window showing the "C" source code with the source lines corresponding to that block or RTL highlighted. This is illustrated in Figure 19. VISTA also has the reverse ability, in the sense that clicking on any line in this source-code window highlights the corresponding RTLs generated for those lines in the main VISTA user interface window. This functionality is accomplished by maintaining source-line information in the intermediate code

Table I. MiBench Benchmarks Used in the Experiments

Category	Program	Description
auto/industrial	bitcount	Test bit manipulation abilities of a processor
network	dijkstra	Calculates shortest path between nodes using Dijkstra's Algorithm
telecomm	fft	Performs a fast fourier transform on an array of data
consumer	jpeg	Image compression and decompression
security	sha	Secure hash algorithm
office	stringsearch	Searches for given words and phrases

generated by the front-end. Note that this information is not always accurate, as certain optimizations, like code motion, move instructions and basic blocks around. In spite of this, such information along with the various static analysis results prove to be very handy when optimizing and debugging any program.

5. EXPERIMENTAL RESULTS

This section describes the results of a set of experiments to illustrate the effectiveness of using VISTA's biased sampling search, which uses a genetic algorithm to find effective sequences of optimization phases. We used a set of *MiBench* programs, which are C benchmarks targeting specific areas of the embedded market [Guthaus et al. 2001]. We used one benchmark from each of the six categories of applications. Descriptions of the programs we used are given in Table I.

We perform the following experiments on the ARM architecture.[1] Our experiments have many similarities to the Rice study, which used a genetic algorithm to reduce code size [Cooper et al. 1999]. We believe the Rice study was the first to demonstrate that genetic algorithms could be effective for finding efficient optimization phase sequences. However, there are several significant differences between their study and our experiments, and we will contrast some of the differences in this section.

The Rice experiments used a genetic algorithm to find effective sequences consisting of twelve phases from ten candidate optimizations. They compared these sequences to the performance obtained from a fixed sequence of twelve optimization phases. In contrast, VPO does not utilize a fixed sequence of phases. Instead, VPO repeatedly applies phases until no other improvements can be obtained. Figure 20 shows the algorithm used to determine the order in which optimization phases are applied to VPO. This algorithm has evolved over the years and the primary goal has always been to reduce execution time. The smallest unit of compilation in the Rice work was a file, which may contain many individual functions. In this study, we perform our searches for effective optimization sequences on a per-function basis.

A complication when assessing VISTA's ability to find effective optimization sequences as compared to the batch VPO compiler is that the *register assignment* (assigning pseudoregisters to hardware registers) and *fixed entry exit* (fixing the entry and exit of the function to manage the run-time stack) phases are

---

[1]This is in contrast to a similar set of experiments performed on the general-purpose SPARC processor in a conference version of the paper [Kulkarni et al. 2003].

```
branch  chaining
remove useless basic blocks
remove useless jumps
remove unreachable code
reverse jumps
remove jumps by block reordering
merge basic blocks
instruction selection
fix control flow
evaluation order determination
global instruction selection
register assignment
instruction selection
minimize loop jumps
if(changes in last phase)
    merge basic blocks
do
    do
        do
            dead assignment elimination
        while changes
        register allocation
        if(changes in last two phases)
            instruction selection
    while changes
    do
        common subexpression elimination
        dead assignment elimination
        loop transformations
        remove useless jumps
        branch chaining
        remove unreachable code
        remove useless basic blocks
        reverse jumps
        remove jumps by block reordering
        remove useless jumps
        if(changes in last 7 phases)
            minimize loop jumps
            if(changes in last phase)
                merge basic blocks
        dead assignment elimination
        strength reduction
        instruction selection
    while changes
while changes
branch chaining
remove unreachable code
remove useless basic blocks
reverse jumps
remove jumps by block reordering
fix entry exit
instruction scheduling
fill delay slots
if(changes in last phase)
    remove useless jumps
    remove branch chains
```

Fig. 20.   VPO's order of optimizations applied in the *batch* mode.

Table II. Candidate Optimization Phases in the Genetic Algorithm Along with Their
Designations

Optimization Phase	Gene	Description
branch chaining	b	Replaces a branch or jump target with the target of the last jump in the jump chain
eliminate empty block	e	Removes empty blocks from the control flow graph
useless jump elimination	u	Remove useless transfers of control like a jump to the next block in the control flow
dead code elimination	d	Remove block unreachable from the top block
reverse branches	r	Reverses a conditional branch when it branches over a jump to eliminate the jump
block reordering	i	Removes a jump by reordering basic blocks when the target of the jump has only a single predecessor
merge basic blocks	m	Merges two consecutive basic blocks when the predecessor has no transfer of control and the successor has only one predecessor
instruction selection	s	Combine instructions together when the combined effect is a legal instruction
evaluation order determination	o	Reorder instructions within a basic block to calc. expressions that require the most registers first
minimize loop jumps	j	Remove an unconditional jump at the end of a loop or one that jumps into a loop, by replicating a portion of the loop
dead assignment elimination	h	Removes assignments where the assignment value is never used
register allocation	k	Replaces references to a variable within a specific live range with a register
common subexpression elimination	c	Eliminates fully redundant calculations
loop transformations	l	Performs loop-invariant code motion, recurrence elimination, loop strength reduction, and induction variable elimination on each loop ordered by loop nesting level. Each of these transformations can also be individually selected by the user.
strength reduction	q	Replaces an expensive instruction with one or more cheaper ones

required, which means that they must be applied only once. Many of the other phases shown in Figure 20 must be applied after *register assignment* and before *fix entry exit*. Thus, we first set up the compiler to perform *register assignment* on demand, i.e. before the first phase which needs it. We then apply the genetic algorithm to find the best sequence of improving phases among fifteen candidate phases attempted by the batch compiler before *fix entry exit*. These fifteen phases, along with a short description of each phase, are listed in Table II.

Another complication was the number of optimization phases to apply, since it may be beneficial to perform a specific optimization phase multiple times. When applying the genetic algorithm, one must specify the number of optimization phases (genes) in each sequence (chromosome). It was not clear how to determine an appropriate uniform limit since the number of attempted optimization phases by the batch compiler could vary with each function. Therefore, we first determine the number of successfully applied optimization phases (those

which affected one or more instructions in the compiled function) during batch compilation. We then multiply the number of successful phases by 1.25 to obtain the number of optimization phases in each sequence for that function. This sequence length is a reasonable limit for each function and still gives an opportunity to successfully apply more optimization phases than the batch compiler is able to accomplish. Note that this number was much less that the total phases attempted by the batch compiler for each function in all of the experiments.

5.1 Batch Compilation Measures

Table III shows batch compilation information for each function in each of the benchmark programs. The first column identifies the program and the number of static instructions that is produced for the application after batch compilation. The second column lists the functions in the corresponding benchmark program. In four of the benchmarks, some functions were not executed, even though we used the input data that was supplied with the benchmark. Since such functions did not affect the dynamic measures, we have designated such functions together as *unexecuted funcs*. The third and fourth columns show the percentage of the program that each function represents for the dynamic and static instruction count after applying the optimization sequence. Although the batch compiler applies the same sequence of optimizations in the same order, many optimizations may not produce any modifications in the program. Iteration also causes some transformations to be repeatedly applied. Thus the sequence and number of optimizations successfully applied often differs between functions. The fifth column shows the sequence and number of optimization phases successfully applied by the batch compiler before *fix entry exit*. One can see that the sequences of successful optimization phases can vary greatly between functions in the same application. The final column shows the total number of optimization phases attempted. The number applied can vary greatly depending upon the size and loop structure of the function. The number of attempted phases is also always significantly larger than the number of successfully applied phases.

We also evaluated the impact that iteratively applying optimization phases had on dynamic and static instruction counts. We obtained this measurement by comparing the results of the default batch compilation to results obtained without iteration, which uses the algorithm in Figure 20 with all the `do-while`'s iterated only once. On average, we found that iteratively applying optimization phases reduced dynamic instruction count by 9.51% over results obtained without iteration. The impact on static counts was much less, with iteration only improving results by 0.54%. This was expected, since the compiler has been originally tuned for execution time alone. In fact, we were initially unsure if any additional dynamic improvements could be obtained using a genetic algorithm given that iteration may mitigate many phase-ordering problems.

5.2 Genetic Algorithm Optimization Measures

There are a number of parameters in a genetic algorithm that can be varied to affect the performance of the generated code. The best algorithm for a particular

Table III. Batch Optimization Measurements

Program and Size	Function	% of Dyna.	% of Stat.	Applied Sequence and Length	Attempted Phases
bitcount (496)	AR_btbl_bitcount	3.22	3.86	emsoskschs (10)	153
	BW_btbl_bitcount	3.05	3.66	emsoshsks (9)	82
	bit_count	13.29	3.25	beurmsohsksc (12)	152
	bit_shifter	37.41	3.86	bedrimsheksc (12)	169
	bitcount	8.47	10.16	emsosksc (8)	81
	main	13.05	19.51	bedrimsosjmhskshcls (19)	177
	ntbl_bitcnt	14.40	3.66	ermssksc (8)	150
	ntbl_bitcount	7.12	8.54	emsoskskskscsc (14)	183
	unexecuted funcs	0.00	43.49	(10.3)	149.8
	average			(10.83)	146.94
dijkstra (327)	dequeue	0.85	10.40	bemsoksch (9)	148
	dijkstra	83.15	44.04	beudrimsojmsbkschls (19)	193
	enqueue	15.81	12.84	beudrimsoksc (12)	152
	main	0.06	22.94	bedrimsojmskshcls (17)	174
	print_path	0.01	8.26	ermsosksc (9)	81
	qcount	0.12	1.53	emsks (5)	76
	average			(11.83)	137.33
fft (728)	CheckPointer	0.00	2.34	bemsks (6)	76
	IsPowerOfTwo	0.00	2.61	bermsohks (9)	79
	NumberOfBits...	0.00	3.98	beurmsokslc (11)	154
	ReverseBits	14.13	2.61	bedimsojmhsksclrlrlr (20)	173
	fft_float	55.88	38.87	beudrimsojmhskskschllhrsc lrlrksclrlrlrs (39)	301
	main	29.98	39.56	bedrimsogsjmhskschlhqscsc (25)	189
	unexecuted funcs	0.00	10.03	(10)	151.00
	average			(17.14)	160.42
jpeg (5171)	finish_input_ppm	0.01	0.04	emsh (4)	66
	get_raw_row	48.35	0.48	eurmsohksc (10)	150
	jinit_read_ppm	0.10	0.35	emsoksc (7)	148
	main	43.41	3.96	beudrimsojmhshksclsc (20)	158
	parse_switches	0.51	11.26	beudrimsojmhsbekschsc (21)	192
	pbm_getc	5.12	0.81	beurmsohkschc (13)	151
	read_pbm_integer	1.41	1.26	beudrimsoeksc (13)	167
	select_file_type	0.27	2.07	beudrimsoekschc (15)	157
	start_input_ppm	0.79	5.96	beudrimsosjmsekschlrlrlrsc (26)	196
	write_stdout	0.03	0.12	emsks (5)	79
	unexecuted funcs	0.00	73.69	(14.04)	148.36
	average			(13.94)	148.04
sha (372)	main	0.00	13.71	bedrimsksclsc (13)	153
	sha_final	0.00	10.75	ermsohsksc (10)	129
	sha_init	0.00	5.11	bedsojmhshsksclrlrlrs (21)	152
	sha_print	0.00	3.76	emshsksc (8)	80
	sha_stream	0.00	11.02	bedimsjmkslsc (13)	151
	sha_transform	99.51	44.62	bedimsojmshskshcllhsclhscscsc (29)	258
	sha_update	0.49	11.02	bedrimsojmhsksclc (17)	181
	average			(15.86)	159.87
string-search (760)	init_search	92.32	6.18	bedimsojmskshc (14)	101
	main	3.02	14.08	bedrimsogjmksclschsc (20)	179
	strsearch	4.66	7.37	bedrimsoseksclsc (16)	173
	unexecuted funcs	0.00	71.44	(16.29)	169.14
	average			(16.40)	163.70
average				(13.87)	151.69

application only comes from experience and continuous fine tuning based on empirical results. In our experiments for this paper, we have used a very basic genetic algorithm loosely based on the one used by Cooper et al. [1999] in their study to effectively reduce the code size for embedded applications. They demonstrated that this algorithm gives much improved performance as compared to random probing. In the future, we plan to experiment with different parameters of the genetic algorithm and study their effects on program performance. The population size (fixed number of sequences or chromosomes) is set to 20 and each of these initial sequences is randomly initialized. Since some phase orderings are not valid in the compiler, checks are made to only generate legal optimization phase sequences. The sequences in the population are sorted by fitness values (using the dynamic and static counts according to the weight factors). At each generation (time step), we remove the worst sequence and three others from the lower (poorer performing) one-half of the population chosen at random. Each of the removed sequences are replaced by randomly selecting a pair of sequences from the upper one-half of the population and then performing a crossover operation on that pair to generate two new sequences. The crossover operation combines the lower one-half of one sequence with the upper one-half of the other sequence and vice versa to create the new pair of sequences. Fifteen chromosomes are then subjected to mutation (the best performing sequence and the newly generated four sequences are not mutated). During mutation, each gene (optimization phase) is replaced with a randomly chosen one with a low probability. For this study, mutation occurs with a probability of 5% for a chromosome in the upper one-half of the population and a probability of 10% in the lower one-half. All changes made by crossover and mutation were checked for validity of the resulting sequences. This was done for a set of 100 generations. Note that many of these parameters can be varied interactively by the user during compilation, as shown in Figure 16.

Table IV shows the results that were obtained for each function by applying the genetic algorithm. For these experiments, we obtained the results for three different criteria. For each function, the genetic algorithm was used to perform a search for the best sequence of optimization phases, based on static instruction count only, dynamic instruction count only, and 50% of each factor. As in Table III, *unexecuted funcs* indicate those functions in the benchmark that were never executed using the benchmark's input data. We also indicate that the effect on the dynamic instruction count was not applicable (N/A) for these functions. The last six columns show the effect on static and dynamic instruction counts, as compared to the batch compilation measures for each of the three fitness criteria. The results that were expected to improve according to the fitness criteria used are shown in boldface. The results indicate that in spite of the inherent randomness of the genetic algorithm, the improvements obtained do tend to lean more toward the particular fitness criteria being tuned. For most functions, the genetic algorithm was able to find sequences that either achieved the same result or improved the result as compared to the batch compilation results.

Figures 21 and 22 show the overall effect of using the genetic algorithm for each test program on the dynamic and static results, respectively. The results

Table IV. Effect on Dynamic Instruction Counts and Space Using the Three Fitness Criteria

Program	Functions	Optimize for Dy. Cnt.		Optimize for Space		Optimizing for Both	
		% Dyna. Improv.	% Stat. Improv.	% Dyna. Improv.	% Stat. Improv.	% Dyna. Improv.	% Stat. Improv.
bitcount	AR_btbl_bitcount	0.00	0.00	0.00	0.00	0.00	0.00
	BW_btbl_bitcount	0.00	0.00	0.00	0.00	0.00	0.00
	bit_count	−1.40	−11.11	−1.40	−11.11	−1.40	−11.11
	bit_shifter	0.40	7.14	0.40	7.14	0.40	7.14
	bitcount	0.00	0.00	0.00	0.00	0.00	0.00
	main	0.001	3.33	0.001	3.33	0.001	3.33
	ntbl_bitcnt	14.68	14.29	14.68	14.27	14.70	14.27
	ntbl_bitcount	0.00	0.00	0.00	0.00	0.00	0.00
	unexecuted funcs	N/A	N/A	N/A	1.60	N/A	0.54
	total	2.72	2.00	2.72	1.81	2.72	1.81
dijkstra	dequeue	0.00	3.85	0.00	3.85	0.00	3.85
	dijkstra	4.20	6.73	−0.01	6.73	4.20	6.73
	enqueue	0.00	0.00	0.00	0.00	0.00	0.00
	main	49.89	5.09	38.53	10.17	49.88	5.09
	print_path	5.12	5.56	5.12	5.56	5.12	5.56
	qcount	0.00	0.00	0.00	0.00	0.00	0.00
	total	3.70	4.84	0.22	6.05	3.70	4.84
fft	CheckPointer	33.33	7.69	16.67	7.69	33.33	7.69
	IsPowerOfTwo	12.5	8.33	0.00	8.33	0.00	0.00
	NumberOfBits...	−23.92	0.00	−26.09	5.00	0.00	0.00
	ReverseBits	−1.27	7.69	−1.27	7.69	−1.27	7.69
	fft_float	11.17	2.29	2.45	3.27	10.71	4.90
	main	3.63	5.10	−4.03	5.10	3.63	5.10
	unexecuted funcs	N/A	N/A	N/A	2.86	N/A	0.00
	total	8.26	3.80	0.51	4.33	7.96	4.62
jpeg	finish_input_ppm	0.00	0.00	0.00	0.00	0.00	0.00
	get_raw_row	0.00	0.00	0.00	0.00	0.00	0.00
	jinit_read_ppm	15.39	15.39	15.36	15.36	15.39	15.39
	main	5.58	−11.00	−0.05	1.50	0.12	6.50
	parse_switches	6.29	1.78	0.70	0.79	−0.70	0.59
	pbm_getc	0.00	0.00	0.00	0.00	0.00	0.00
	read_pbm_integer	5.00	−2.00	0.00	0.00	0.00	0.00
	select_file_type	0.00	0.00	0.00	−1.61	0.00	0.00
	start_input_ppm	2.68	3.57	2.70	3.57	2.70	3.57
	write_stdout	0.00	0.00	0.00	0.00	0.00	0.0
	unexecuted funcs	N/A	N/A	N/A	5.02	N/A	5.48
	total	3.18	0.27	0.05	4.20	0.11	4.70
sha	main	9.68	4.65	9.68	4.65	9.68	4.65
	sha_final	4.17	5.13	4.17	5.13	4.17	5.13
	sha_init	0.00	0.00	0.00	0.00	0.00	0.00
	sha_print	8.33	8.33	8.33	8.33	8.33	8.33
	sha_stream	0.61	8.83	−31.22	26.47	8.57	32.35
	sha_transform	0.70	10.18	−2.42	13.92	0.35	12.66
	sha_update	0.06	8.82	0.00	5.71	0.00	5.71
	byte_reverse	1.75	−41.67	0.52	10.42	0.52	10.42
	total	0.90	1.30	−1.83	11.20	0.38	11.20
string search	init_search	0.15	12.9	0.15	12.90	0.15	12.90
	main	0.002	6.15	−3.21	9.23	−3.21	9.23
	strsearch	3.80	5.00	−0.62	5.00	−0.62	5.00
	unexecuted funcs	N/A	N/A	N/A	8.38	N/A	8.82
	total	0.29	7.35	0.05	9.85	0.05	9.85
average		3.18	3.26	0.29	6.24	2.49	6.17
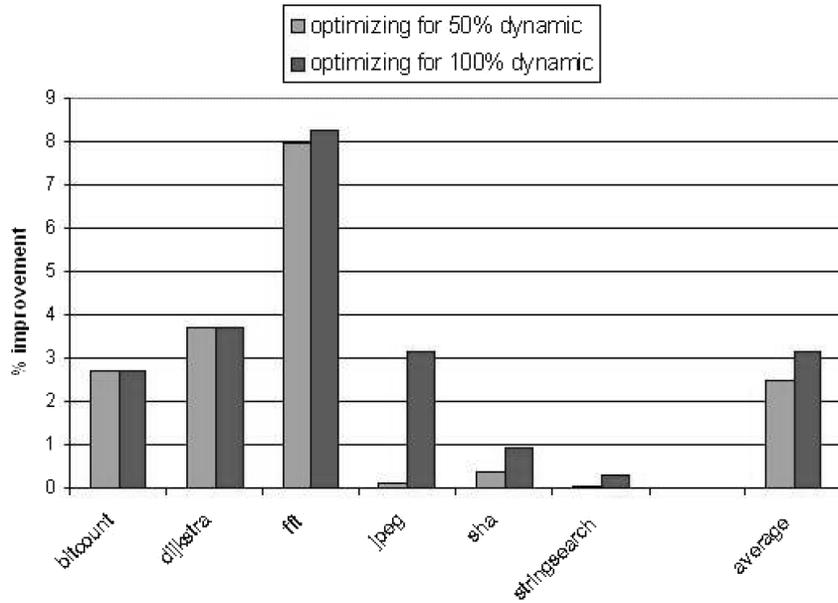
Fig. 21. Overall effect on dynamic instruction count.
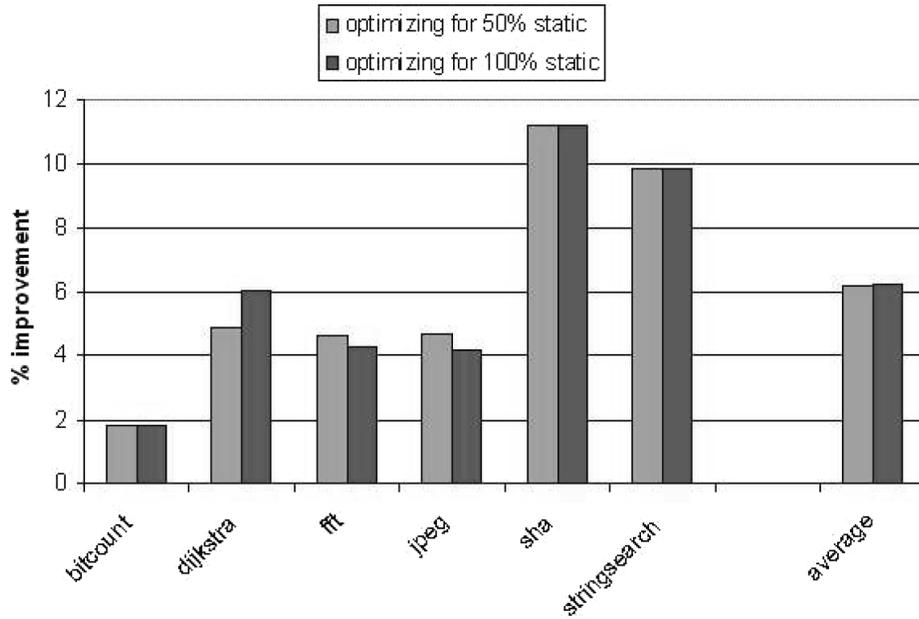
Fig. 22. Overall effect on static instruction count.

show that the genetic algorithm was more effective at reducing the static instruction count than dynamic instruction count, which is not surprising since the batch compiler was developed with the primary goal of improving the execution time of the generated code and not reducing code size. However, respectable

dynamic improvements were still obtained despite having a baseline with a batch compiler that iteratively applies optimization phases until no more improvements could be made. Note that many batch compilers do not iteratively apply optimization phases and the use of a genetic algorithm to select optimization phase sequences will have greater benefits as compared to such non-iterative batch compilations. The results when optimizing for both dynamic instruction counts and space showed that we were able to achieve close to the same dynamic benefits when optimizing for dynamic instruction counts and close to the same static benefits when optimizing for space. A user can set the fitness criteria for a function to best improve the overall result. For instance, small functions with high dynamic instruction counts can be optimized for speed, functions with low dynamic instruction counts can be optimized primarily for space (since these measurements are typically much faster as we do not need to execute the code), and large functions with high dynamic counts can be optimized for both space and speed. The optimization phase sequences selected by the genetic algorithm for each function are shown in Table V. The sequences shown are the ones that produced the best results for the specified fitness criteria. Similar to the results in Table III, these sequences represent the optimization phases successfully applied, as opposed to all optimization phases attempted.

From these results it appears that *strength reduction* was rarely applied since we used dynamic instruction counts instead of taking the latencies of more expensive instructions, like integer multiplies, into account. It appears that certain optimization phases enable other specific phases. For instance, *instruction selection* (s) often follows *register allocation* (k), since instructions can often be combined after memory references are replaced by registers. Likewise, *dead assignment elimination* (h) often follows *common subexpression elimination* (c), since a sequence of assignments often become useless when the use of its result is replaced with a different register.

The results in Table V also show that functions within the same program produce the best results with different optimization sequences. The functions with fewer instructions typically had not only fewer successfully applied optimization phases but also less variance in the sequences selected between the different fitness criteria. Note that many sequences may produce the same result for a given function and the one shown is just the first sequence found that produces the best result.

Finding the best sequence using the genetic algorithm for 100 generations with a population size of twenty required a few hours for each function on an ARM. The compilation time was less when optimizing for size only, since we would only get dynamic instruction counts when the static instruction count was less than or equal to the count found so far for the best sequence. In this case we would use the dynamic instruction count as a secondary fitness value to break ties. In general, we found that for most functions the search time was dominated not by the compiler, but instead by assembling, linking, and executing the program. If we use size without obtaining a dynamic instruction count, then we typically obtain results for each function in far less time. However, for some of the bigger functions, with long sequence lengths, compilation time is

Table V. Optimization Phase Sequences Selected Using the Three Fitness Criteria

Program	Functions	Optimizing for Dy. Cnt.	Optimizing for Space	Optimizing for Both
bitcount	AR_btbl_bitcount	skes	sksech	chsksmd
	BW_btbl_bitcount	skes	skes	dskms
	bit_count	crhdesikscmsr	mshkurcdks	humsrkhces
	bit_shifter	bsekhdsricb	erskrshcb	skcrbsdhr
	bitcount	sdokms	sodksm	sdoks
	main	dsokhrslrsmkcs	srohskescmlcr	soeksmcslhrsksr
	ntbl_bitcnt	crhsks	csckrehs	srchks
	ntbl_bitcount	smkcse	dmsks	smkcse
dijkstra	dequeue	soerkscb	sdksmcr	smkrsce
	dijkstra	skerslbemucr	rskiebsdcre	sikscrbdjce
	enqueue	esbdimks	idskebsd	mbsudmiksmc
	main	idskelrsblmcd	sksecbidemr	skmdrlrslc
	print_path	ersmkcs	csksmcrc	skcres
	qcount	smkc	smkc	chsm
fft	CheckPointer	scrkcdh	skrms	rcskhc
	IsPowerOfTwo	sboerhks	seckimh	bcrskhm
	NumberOfBits...	sbchekrumcl	ibisrdklcs	iskbmrsljd
	ReverseBits	dskmciebhsd	skehrsrmc	ecrskchsmr
	fft_float	brdsuckschclrslhlscks	eshkmrlslhursclrshks	durshbcdkhksclrshlhsc
	main	bskdlerihkms	sekhrlbrmsch	dskbhdlmrirsh
jpeg	finish_input_ppm	ech	ech	ech
	get_raw_row	shekci	shkurcm	esuhkce
	jinit_read_ppm	cskdsc	mskcs	skcsm
	main	bdljmiskesbehr	mesobhldksrcu	sbodrmkshlcur
	parse_switches	ishskmbdscrejs	rshsurbksdcsre	srhskusdrcbmsd
	pbm_getc	dsokricbhe	sokeurech	soucrkmcdeh
	read_pbm_integer	ecbsksdchmrir	smbkcrsedi	soucrkcshr
	select_file_type	dsrkscimhbr	sdkscriheb	bsrkdmschi
	start_input_ppm	useokrbdrsircem	rsikcedbsrdr	rsikmcsedrrbm
	write_stdout	cdmh	cdmh	smkc
sha	main	dsrbelrmkcs	ibmrskdscs	ibrskldcs
	sha_final	dskhicrs	sdkmcicsrh	skhmcrecs
	sha_init	sksmce	esksc	msksec
	sha_print	schdkmc	smeckch	sdckhc
	sha_stream	rslrkjs	dsrksr	clchesksbdu
	sha_transform	seoblkcmdjclhslsmbic	somrlkcdlhslcrsc	recmqsklcshlcsr
	sha_update	sdiorkcbhdcmjs	sokecmrhcs	sobkcrsirdcmh
string-search	init_search	sebkdsmcim	birsksdcm	esbksdmcie
	main	slkrsdlcmh	rmeslsksrc	dmsbdklsrumc
	strsearch	bsksedurc	skbserimced	dbedskrisce

also significant on the ARM. We have developed techniques to avoid evaluating redundant sequences to further speed up the genetic algorithm [Kulkarni et al. 2004]. In the future, we plan to enhance these techniques to further, reduce the search time.

5.3 Simulation Results

In the previous section, we presented *dynamic instruction counts* as an indicator of the run-time execution speed of the application. In an ideal world, where each instruction takes the same number of cycles to execute, operating system policies like context switches, processor, and machine characteristics, like

pipelines and memory hierarchies, and the delay caused by I/O do not have any noticeable effect on program execution time, this would indeed be true. However, on most machines these features have a significant impact on the execution time of any application. In fact, for an application dominated by I/O, or for an application dominated by system calls and calls to other functions, reducing only the number of dynamic instructions in the current function only may have no noticeable effect on execution time. When combined with pipeline, memory hierarchy, and OS effects, this might even result in an actual increase in the execution time for some program executions.

As a result many people do not consider dynamic instruction counts to provide a reasonable indication of program performance. However, getting accurate and reproducible execution times is nontrivial in the presence of OS and complicated architectural features. To offset such effects, many users perform multiple runs of the application and show the average execution time along with the standard deviation. This method provides reasonably reproducible measures for compute-intensive kernels. However, such a method also has a tendency to break down for applications doing I/O. The small performance improvements we generally observe can also be easily dwarfed by the variations in the execution times because of such external factors. While such individual variations for each function may be fairly small, their cumulative effect may be more substantial. Finally, the development environment is often different from the target environment for embedded applications and actual executions may not be possible. Such considerations lead us to use a simulator to get cycle counts for the function being optimized, since these measures are repeatable. Improvement in simulation cycle counts, while taking pipeline and memory hierarchy effects into consideration, can serve as a reasonable estimation of execution time improvement [Burger and Austin 1997].

As part of our experiments, we used the SimpleScalar toolset [Burger and Austin 1997], which is widely used and acknowledged. The cycle-accurate version of the ARM simulator computes cycle counts for the entire application. This would include the cycle counts of all procedure and system calls made from the function being optimized. Since that part of the code was not impacted by the current optimization sequence, including cycle counts for those functions was inappropriate. Thus, we modified the simulator to disregard cycle counts when it leaves the current function and again start the counts upon reentering the current function. However, the simulation of these other functions still includes the effect on various stores, like branch prediction buffers and caches. Thus, we believe that other simulator idiosyncrasies like branch mispredictions and cache misses have been reasonably addressed during this modification. Obviously, simulation takes orders of magnitude more time than direct execution (for the benchmark *fft*, over several executions, we observed that simulation takes at least 5000 times as much time as direct execution). We also wanted to show that the improvements using cycle counts are fairly consistent with those using dynamic instruction counts. Thus, we have only evaluated the performance for one benchmark, *fft*. We chose *fft* as it had shown the best overall dynamic improvement (when optimizing for only dynamic instruction counts) and because this benchmark has a good mix of both small and large functions.

Table VI. Effect of Iterative Compilation on Dynamic Cycle Counts

Functions	Batch Total Cycles	Iterative Total Cycles	% Iterative Improvement
CheckPointer	15	11	26.67
IsPoverOfTwo	76	43	43.42
NumberOfBits...	78	74	5.13
ReverseBits	147523	147496	0.02
fft_float	1444030	1234553	14.51
main	537587	421972	21.51
Total	2129309	1804149	15.27

Each function in this benchmark is only processed for 50 generations, as opposed to the 100 generations used for the above experiments. This was done as we had observed during our prior experiments that most functions reach their best sequence in the first 50 generations. All other experiment parameters have been maintained the same. The simulation environment was configured for an inorder ARM processor with issue, decode, and commit widths of 4, 16 KB L1 data cache, 16 KB L1 instruction cache, and 256 KB shared L2 cache. Table VI shows the improvements in cycle counts given by the sequence found by the genetic algorithm over the batch compiler measures.

The results in Table VI show that the improvements in dynamic cycle counts are correlated to the improvements in dynamic instruction counts shown in Table IV. The functions which actually degraded in Table IV show only marginal improvements in Table VI. Overall, for the function studied, the improvements in actual cycle counts are almost twice the improvements in dynamic instruction counts, as compared to the batch compiler measures. The results may vary for other functions. Note that these results only show the suitability of iterative compilation to improve cycle counts and, in no way, suggests that our approach is more beneficial to cycle counts than to dynamic instruction counts. Also, in most cases, the actual sequences giving the best performance in Table VI are different than the sequences achieving best performance for corresponding functions in Table IV. The use of simulation versus dynamic instruction counts is effectively a tradeoff between accuracy and search time.

6. COMBINING INTERACTIVE AND AUTOMATIC COMPILATION

A major advantage of interactive compilation is that the user can utilize his/her knowledge about the application and machine domains to produce better code than that produced automatically by the compiler. Automatic compilation is still important as it greatly eases the task of performing manual changes later. In this section, we illustrate the ability of VISTA to combine the benefits of both interactive and automatic tuning on some section of code by providing a case study that involves tuning a critical function extracted from a software that controls a compact disc player (see Figure 23).

For this particular function, the application programmer applies his knowledge of the problem domain and uses 8-bit data variables, when possible, to reduce the memory footprint of the application. For example, the programmer knows that the loop-induction variables will not exceed 8 bits and declares the

856 • P. Kulkarni et al.

```
1   char uskb[1000];
2   void function() {
3         char cnt, i, data;
4         if(uskb[0]) {
5               for( cnt=uskb[0]; cnt > 1; cnt-- ) {
6                     data=uskb[cnt];
7                     for( i = 1; i <= 21; i++ ) {
8                           if( data == sk[i] ) {
9                                 sk[i] = 0xff;
10                                sk[0]--;
11                                break;
12                          }
13                    }
14              }
15              for( cnt = 1 ; cnt <= 21; cnt++ )
16                    uskb[cnt] = 0xff;
17              uskb[0] = 0;
18        }
19  }
```

Fig. 23. C source.

Table VII. Summary of Case Study

Compilation Method	Code Size	Instructions in Critical Loop
batch compiler	45	5
batch plus hand edits	42	4
VISTA's GA	42	5
GA plus hand edits	39	4
interactive GA and hand edits	39	4

induction variables to be 8-bit chars. If the optimizer determines it is best to leave these variables on the stack, memory is conserved. However, if the compiler promotes these variables to 32-bit registers, it must preserve the program semantics, which do not allow the value to become larger than 8 bits. Consequently, the compiler inserts truncation instructions (and reg, reg, #255) when a register holding an 8-bit value is modified. In the example, the three increments of the three induction variables incur this truncation overhead. One might claim that the compiler could remove the extra operations, and this is partially true. However, the truncation operation for the induction variable cnt at line 5 cannot be removed, since the loop bounds are unknown at compile time.

To illustrate the effectiveness of VISTA's compilation techniques, the sample program was compiled using five different approaches (see Table VII). First, the program was compiled using the batch compiler. The batch compiler generated 45 instructions for the function and five instructions for the critical loop (line 15). If the resulting code does not meet the specified size and performance constraints, one option the programmer has is to hand edit the output of the batch compiler. Removing the truncation instructions manually, the programmer can save three instructions for the function and one in the critical loop (see row two of Table VII).

Using VISTA's genetic algorithm to apply all optimizations (200 generations, 50% size, and 50% dynamic instruction counts) yields better code than the

ACM Transactions on Embedded Computing Systems, Vol. 5, No. 4, November 2006.

```
1    .L17: cmp     r2, #21          # compare r2 with 21
2           strleb  r12,[r5,r2]      # store least significant byte in r12 if (r2 <= 21)
3           addle   r2,r2,#1         # increment r2 if (r2 <= 21)
4           ble     .L17             # loop back to .L17 if (r2 <= 21)
```

Fig. 24.   Loop produced by GA plus hand edits.

```
1    .L17: and     r2,r2,#255       # only keep the least significant byte
2           cmp     r2, #21          # compare r2 with 21
3           strleb  r12,[r6,r2]      # store least significant byte in r12 if (r2 <= 21)
4           addle   r2,r2,#1         # increment r2 if (r2 <= 21)
5           ble     .L17             # loop back to .L17 if (r2 <= 21)
```

Fig. 25.   Loop produced from the batch compiler.

batch compiler, 42 instructions overall with 5 instructions in the critical loop. However, applying the same hand edits that were applied to batch compiler's output yields a function with 39 instructions and 4 instructions in the critical loop. This result is a significant savings over the initial batch version.

It may be possible to do even better. By examining the sequence of optimizations VISTA's genetic algorithm applies and their effect on the resulting code, the application programmer can determine that register allocation introduces the truncation-related inefficiencies in the program. By discarding optimizations after register allocation, applying the hand edits, and rerunning the GA, VISTA may be able to produce even better assembly than doing a final edit of the output of the genetic algorithm. In this particular case study, no further improvements are gained, but it is easy to believe that other functions being tuned may see further improvement. Table VII summarizes the results of the case study. Figures 24 and 25 show the best and worst assembly for our critical loop (line 15 in the source) obtained in Table VII.

One might claim that hand edits would be unnecessary if the programmer had declared the induction variables to be type int. Although true for our example, modifying the source program is not always possible. In a more complex function, induction variables cannot always be assigned to a register for the entire execution of the function. If an induction variable needs to reside on the stack for a portion of the program, either hand edits are necessary or the compiler will waste stack memory.

7. IMPLEMENTATION ISSUES

In this section, we discuss the implementation of certain features in VISTA.

7.1 Correct Button Status in the Viewer

In the VISTA user interface the only buttons active at any point during the compilation process are those that can be legally selected by the user. The rest of the buttons are grayed out. As the user selects optimization phases to be applied, the sets of selectable and disabled buttons should change. For example, selecting *register assignment* enables many other optimizations dependent on *register assignment*, like *register allocation* and *code motion*. Clicking *fill delay slots* grays out most other optimizations, which are not legal after this

phase. The control statements like the *if*, *while*, *select best from*, and *select best combination* constructs complicate the act of determining which buttons are active during and after applying each construct. This problem is further complicated by the feature of undoing previously applied transformations, supported in VISTA, since this requires storing detailed button status information at many different points with the ability of getting back to a previous button state when changes are undone. The interaction of all these factors made the task of determining the correct button status a nontrivial task requiring meticulous and careful handling.

7.2 Proper Handling of Interactive Compilation

The original VPO compiler was batch-oriented and had a fixed order in which optimizations and the various static program analysis required for each optimization were done. To modify the compiler for interactive compilation, we carefully isolated the analysis required for each optimization. At places, the analysis required was done as part of the optimization itself. In such cases, we modularized the code to separate the analysis part from the actual optimization part. It was also impractical to do all the required analysis before each optimization phase, as some of those might have been performed for some previous phase and would still be legal at the current compilation point. Thus, we also identified the analysis invalidated by each optimization phase. Only the static program analysis not previously performed, or invalidated by some previous optimization must be applied before each optimization phase.

In VISTA, there are two separate program states maintained, one at the compiler and another at the viewer. It is important that the two remain consistent at every point during the compilation session. The user only views the program order and information in the viewer. If this information does not reflect the correct status in the compiler, then the user would be misled and the whole point of interactively tuning the program would be lost. Thus, we made sure that each change made in the compiler is also sent to the viewer. We also implemented a *sanity check*, that compares the two program states for consistency and informs the user of inconsistencies.

7.3 Maintaining the Transformation List

When transformations are undone, the current program state in the compiler is discarded and a fresh program state with no optimizations is reread. Transformations are then applied to this program state from a transformation list, which is a linked list of all the changes previously applied. The changes are only applied up to a certain point so that all the changes not applied are lost or *undone*. Note that the original changes to the program are made by calling the actual optimization routines, while after *undoing*, the changes are reapplied by reading from a list. Also, at the end of the current compilation session, the transformation list is saved to a file, which is read back in and reapplied at the start of the next session.

We maintain a list of all previous transformations in VISTA, rather than just the optimization phases applied, for three reasons. First, maintaining all the

changes in a separate list makes it possible to undo a part of some optimization. Second, the implementation of optimization routines may be changed by compiler writers in between two sessions. As the changes are read from a file, before the next session, the user is able to see the program in the same state as it was at the end of the previous session, irrespective of whether the implementation of the actual optimization routine has been changed. Third, this was needed to correctly reapply *hand-specified* transformations, since each hand-specified transform is an individual customized change by the user, and so does not make the same change each time. Thus, we need the added granularity of saving each individual change, instead of only saving the list of optimization phases. Saving all the changes in the transformation list and reapplying them later to produce the same effect took much time and careful design and implementation.

7.4 Batch Mode for Experiments

In an interactive environment, the user is expected to perform all tasks using keyboard and mouse clicks. While this is reasonable during normal compilation sessions, it is impractical while debugging and conducting experiments. To simplify and expedite the task of performing experiments, a batch execution mode is included in VISTA. It is possible in the viewer to save all the button clicks to a file. This file can be called, either from the command line or from the user interface to reexecute the same set of commands. The format of the file is simple enough to even write manually. The experiments involving genetic algorithms take a lot of time and usually run overnight. The ability to use batch compilation gave the authors the option to automate the entire process by writing scripts. This saved a lot of time and unnecessary manual labor.

7.5 Interrupting/Restarting Iterative Measurements

Program improvement using iterative compilation is a time-consuming process. Many functions typically require several hours to evaluate all the sequences during the genetic algorithm. In such a scenario, if the compiler breaks down or the machine crashes in the middle of the algorithm, then valuable time is lost, if we need to restart the algorithm from the beginning. To save time, it would be nice if one could resolve the problem and then restart the algorithm from the point it had crashed. This feature is indispensable when obtaining simulator cycle counts since the compilation time soars from hours to days. This feature is now built in to VISTA. An important consideration was maintaining the states of the hashtables, so that the same number of sequences would still be detected as redundant.

8. FUTURE WORK

There is much future work to be considered on the topic of selecting effective optimization sequences. We currently use a very simple genetic algorithm to search for effective sequences. Changing the genetic algorithm can give vastly different results. The crossover and mutation operations can be changed. Presently, even with this algorithm, we only obtained measurements for 100 generations and a optimization sequence that is 1.25 times the length

of successfully applied batch optimization sequence. It would be interesting to see how performance improves as the number of generations and the sequence length varies. In addition, the set of candidate optimization phases could be extended. Finally, the set of benchmarks could be increased.

Finding effective optimization sequences using the genetic algorithm is a time consuming process. This problem is even more severe on embedded processors, because most of these systems only provide execution time measurements via simulation on a host processor. The overhead of simulating programs to obtain speed performance information may be problematic when performing large searches using a genetic algorithm, which would likely require thousands of simulations. Embedded processors usually also have slower clock speeds than general-purpose processors. We are currently exploring ways to reduce the search time by evolving approaches to find more redundant sequences, which would not need execution or simulation. Also worth researching are ways to find the best sequence earlier in the search process, so that it would be possible to run the search for fewer generations.

All of the experiments in our study involved selecting optimization-phase sequences for entire functions. Since we also have the ability in VISTA to limit the scope of an optimization phase to a set of basic blocks, it would be interesting to perform genetic algorithm searches for different regions of code within a function. For frequently executed regions, we could attempt to improve speed and for infrequently executed regions we could attempt to improve space. Selecting sequences for regions of code may result in the best measures when both speed and size are considered. It would be even more interesting if we could automate the process of limiting the scope, based on program profile measurements.

In the future we also plan to work on reducing the time required for the simulations. In addition to the cycle accurate simulator, the SimpleScalar toolset also includes a fast functional simulator, which only measures the dynamic instruction count. Since we only measure cycles in the current function, we could evaluate that part of the application with the slow-cycle simulator. The rest of the program can be simulated using the fast-functional simulator. We anticipate this would result in only a small inaccuracy compared to using the cycle simulator for the entire simulation. We are in the process of integrating the two simulators to allow switching back and forth between the slow and the fast modes.

9. CONCLUSIONS

We have described a new code improvement paradigm that changes the role of low-level code improvers. This new approach can help achieve the cost/performance tradeoffs that are needed for tuning embedded applications. By adding interaction to the code improvement process, the user can gain an understanding of code improvement tradeoffs by examining the low-level program representation, directing the order of code improvement phases, applying user-specified code transformations, and visualizing the impact on performance.

The ability to automatically provide feedback information after each successfully applied optimization phase allows the user to gauge the progress when

tuning an application. The structured constructs allow the conditional or iterative application of optimization phases and, in essence, provides an optimization phase programming language. We have also provided constructs that automatically select optimization phase sequences based on specified fitness criteria. A user can enter specific sequences and the compiler chooses the sequence that produces the best result. A user can also specify a set of phases, along with options for exploring the search space of possible sequences. The user is provided with feedback describing the progress of the search and may abort the search and accept the best sequence found at that point.

We have also performed a number of experiments to illustrate the effectiveness of using a genetic algorithm to search for efficient sequences of optimization phases. We found that significantly different sequences are often best for each function even within the same program or module. However, we also found that certain optimization phases appear to enable other specific phases. We showed that the benefits can differ, depending on the fitness criteria, and that it is possible to use fitness criteria that takes both speed and size into account. While we demonstrated that iteratively applying optimization phases until no additional improvements are found in a batch compilation can mitigate many phase-ordering problems with regard to dynamic instruction count, we found that dynamic improvements could still be obtained from this aggressive baseline using a genetic algorithm to search for effective optimization phase sequences.

An environment that allows a user to easily tune the sequence of optimization phases for each function in an embedded application can be very beneficial. This system can be used by embedded systems developers to tune application code, by compiler writers to prototype, debug, and evaluate proposed code transformations, and by instructors to illustrate code transformations.

REFERENCES

ANDREWS, K., HENRY, R., AND YAMAMOTO, W. 1988. Design and implementation of the UW illustrated compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 105–114.

APPELBE, B., SMITH, K., AND MCDOWELL, C. 1989. Start/pat: A parallel—programming toolkit. In *IEEE Software . 4*, 6, 29–40.

BENITEZ, M. E. AND DAVIDSON, J. W. 1988. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 conference on Programming Language Design and Implementation*. ACM Press, New York. 329–338.

BENITEZ, M. E. AND DAVIDSON, J. W. 1994. The advantages of machine-dependent global optimization. In *Proceedings of the 1994 International Conference on Programming Languages and Architectures*. 105–124.

BOYD, M. AND WHALLEY, D. 1993. Isolation and analysis of optimization errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 26–35.

BOYD, M. AND WHALLEY, D. 1995. Graphical visualization of compiler optimizations. *Programming Languages 3*, 69–94.

BROWNE, J., SRIDHARAN, K., KIALL, J., DENTON, C., AND EVENTOFF, W. 1990. Parallel structuring of real-time simulation programs. In *COMPCON Spring '90: Thirty-Fifth IEEE Computer Society International Conference*. 580–584.

BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News 25*, 3, 13–25.

CHOW, K. AND WU, Y. 1999. Feedback-directed selection and characterization of compiler optimizations. In *Workshop on Feedback-Directed Optimization*.

COOPER, K. D., SCHIELKE, P. J., AND SUBRAMANIAN, D. 1999. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*. ACM Press, New York. 1–9.

DAVIDSON, J. W. AND WHALLEY, D. B. 1989. Quick compilers using peephole optimization. *Software—Practice and Experience 19*, 1, 79–97.

DAVIDSON, J. W. AND WHALLEY, D. B. 1991. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems 15*, 9 (Nov.), 459–472.

DOW, C.-R., CHANG, S.-K., AND SOFFA, M. L. 1992. A visualization system for parallelizing programs. In *Supercomputing*. 194–203.

GRANLUND, T. AND KENNER, R. 1992. Eliminating branches using a superoptimizer and the GNU C compiler. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*. 341–352.

GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*.

HARVEY, B. AND TYSON, G. 1996. Graphical user interface for compiler optimizations with Simple-SUIF. Technical Report UCR-CS-96-5, Department of Computer Science, University of California Riverside, Riverside, CA.

KISUKI, T., KNIJNENBURG, P. M. W., AND O'BOYLE, M. F. P. 2000. Combined selection of tile sizes and unroll factors using iterative compilation. In *IEEE PACT*. 237–248.

KNIJNENBURG, P., KISUKI, T., GALLIVAN, K., , AND O'BOYLE, M. 2000. The effect of cache models on iterative compilation for combined tiling and unrolling. In *Proc. FDDO-3*. 31–40.

KULKARNI, P., ZHAO, W., MOON, H., CHO, K., WHALLEY, D., DAVIDSON, J., BAILEY, M., PAEK, Y., AND GALLIVAN, K. 2003. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM Press, New York. 12–23.

KULKARNI, P., HINES, S., HISER, J., WHALLEY, D., DAVIDSON, J., AND JONES, D. 2004. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*.

LIAO, S.-W., DIWAN, A., ROBERT P. BOSCH, J., GHULOUM, A., AND LAM, M. S. 1999. SUIF Explorer: An interactive and interprocedural parallelizer. In *Proceedings of the seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, New York. 37–48.

MARWEDEL, P. AND GOOSSENS, G. 1995. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Boston, MA.

MASSALIN, H. 1987. Superoptimizer: A look at the smallest program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*. 122–126.

NISBET, A. 1998. Genetic algorithm optimized parallelization. In *Workshop on Profile and Feedback Directed Compilation*.

NOVACK, S. AND NICOLAU, A. 1993. VISTA: The visual interface for scheduling transformations and analysis. In *Languages and Compilers for Parallel Computing*. 449–460.

POLYCHRONOPOULOS, C., GIRKAR, M., HAGHIGHAT, M., LEE, C., LEUNG, B., AND SCHOUTEN, D. 1989. Parafrase–2: An environment for parallelizing, partitioning, and scheduling programs on multiprocessors. In *International Journal of High Speed Computing*. 1, vol. 1. Pennsylvania State University Press, 39–48.

VEGDAHL, S. R. 1982. Phase coupling and constant generation in an optimizing microcode compiler. In *Proceedings of the Fifteenth Annual Workshop on Microprogramming*. 125–133.

WHALEY, R., PETITET, A., AND DONGARRA, J. 2001. Automated empirical optimization of software and the ATLAS project. In *Parallel Computing*. 1-2, vol. 27. 3–25.

WHITFIELD, D. L. AND SOFFA, M. L. 1997. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS) 19*, 6, 1053–1084.

ZHAO, W., CAI, B., WHALLEY, D., BAILEY, M. W., VAN ENGELEN, R., YUAN, X., HISER, J. D., DAVIDSON, J. W., GALLIVAN, K., AND JONES, D. L. 2002. VISTA: a system for interactive code improvement. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*. ACM Press, New York. 155–164.

ZHAO, W., KULKARNI, P., WHALLEY, D., HEALY, C., MUELLER, F., AND UH, G.-R. 2004. Tuning the wcet of embedded applications. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium*.