THE FLORIDA STATE UNIVERSITY
COLLEGE OF ARTS AND SCIENCES

PERFORMANCE DRIVEN OPTIMIZATION TUNING IN VISTA

By
PRASAD KULKARNI

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

Degree Awarded:
Summer Semester, 2003

The members of the Committee approve the thesis of Prasad Kulkarni defended on May 30th, 2003.

David Whalley
Major Professor

Xin Yuan
Committee Member

Kyle Gallivan
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

Acknowledgements

I am deeply indebted to my adviser, Dr. David Whalley for his support and guidance, without which this work would not have been accomplished. I would also like to thank the other members of my committee for their comments and criticism during my work and also Dr. Robert Engelen for helping me out on many occasions.

I also appreciate the help and support of my friends, particularly Wankang Zhao for his feedback on VISTA, William Kreahling for helping me understand some aspects of VPO, Clint Whaley for his ever ready and useful suggestions and Hwa-Shin Moon and Kyung-Hwan Cho for their work on the user interface of VISTA which proved to be very useful.

Finally, I would like to thank my parents and my sister for the support and encouragement they always offered.

Table of Contents

List of Tables

List of Figures

Abstract

This thesis describes the support provided in VISTA for finding effective sequences of optimization phases. VISTA is a software tool for a compiler which supports an interactive compilation paradigm. It has long been known that a single ordering of optimization phases will not produce the best code for every application. This phase ordering problem can be more severe when generating code for embedded systems due to the need to meet conflicting constraints on time, code size and power consumption. Given that many embedded application developers are willing to spend time tuning an application, we believe a viable approach is to allow the developer to steer the process of optimizing a function. With this in mind, we have enhanced VISTA with many new features and programming-language-like constructs. VISTA also provides the user with dynamic and static performance information that can be used during an interactive compilation session to gauge the progress of improving the code. In addition, VISTA provides support for automatically using performance information to select the best optimization sequence among several attempted. One such feature is the use of a genetic algorithm to search for the most efficient sequence based on specific fitness criteria. This thesis also includes a number of experimental results that evaluate the effectiveness of using a genetic algorithm in VISTA to find effective optimization phase sequences.

Chapter 1

Introduction

The phase ordering problem has long been known to be a difficult dilemma for compiler writers [1]. A single sequence of optimization phases is unlikely to produce optimal code for every application (or even each function within an application) on a given machine. A particular optimization phase may provide or prevent opportunities for improvements by a subsequent optimization phase. Some optimizations can provide opportunities for other optimizations to be applied. For instance, register allocation replaces load and store instructions with register-to-register moves, which in turn provides many opportunities for instruction selection. Likewise, some optimizations can eliminate opportunities for other optimizations. One example of this situation is when register allocation may consume the last available register within a specific region of code, which would prohibit other optimizations that require registers from being applied in that region. 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 [2].

The problem of ordering optimization phases can be more severe when generating code for embedded applications. Many applications for embedded systems need to meet constraints on time, code size, and power consumption. Often an optimization that could improve one aspect (e.g., speed) can degrade another (e.g., size) [3]. For example, many

loop transformations may reduce execution time and increase code size. In fact, it may be desirable on many systems to enhance execution time for the frequently executed code portions and reduce code size for the less frequently executed portions. In addition, embedded microprocessors often have irregular instruction sets with small register files that can quickly be exhausted during the compilation. Exploiting special purpose architectural features may require other optimizations to be performed. Thus, embedded systems are more likely to be sensitive to the order in which optimization phases are applied.

The traditional compilation framework has a fixed order in which the optimization phases are applied and there is no control over individual transformations, except for compilation flags to turn code-improving transformations on or off. The compilation framework, called VISTA (VPO Interactive System for Tuning Applications) [4], gives the application user the ability to finely control the code-improvement process. It, however, failed to provide the user with a structured language, at a high level, for specifying the optimization phases. In the later sections we will see a number of control statements added to VISTA which now enables the user to better steer the code-improvement process. Such statements provide the user with more control in specifying the order of optimization phases.

Also, the earlier version of VISTA provided no feedback to the user by which he/she can gauge the code improvement results. In such cases, it is impossible for the user to conclude if a particular optimization sequence is the best for that region of code or even to infer if the performance goals set for that program have actually been met (even if the optimization sequence applied is not best one). In this thesis, we have provided two different solutions for this problem. The first solution, which we call *Interactive Performance Driven Code Tuning* requires user knowledge and intuition to steer the compilation process in the right direction and produce efficient code. In this approach the user manually applies the transformations, but gets regular feedback from the compiler regarding the performance of the resulting code at each stage. These performance statistics make it easier to decide on the next step in the code improvement process. VISTA also allows the user to undo previous transformations, which facilitates experimentation with different optimization sequences. We call the other approach *Automatic Performance Driven Code Tuning*, where the compiler looks at different

sequences of optimizations and based on their performances it automatically selects the best one and then applies it to the program. To support this feature we have added two new constructs to VISTA, called *select best sequence* and *select best combination*. It should be noted that given any set of optimization phases, the number of different orderings can potentially be exponential. Evaluating an exponential number of sequences is not likely to be feasible in a majority of cases. In this report we also discuss ways in which we have attempted to effectively probe the search space to quickly obtain an acceptable optimization sequence.

The first section of this report familiarizes the reader with some existing technologies which were used during its creation. The next section describes how the performance feedback measures can be obtained and its implementation. The next two sections discuss the support provided in VISTA for interactive and automatic performance-driven code tuning. This is followed by some experimental results which demonstrate the usefulness of this work. In the following section, some interesting implementation issues are presented. The final sections are devoted to a discussion of related work and scope for future work.

Chapter 2

VPO, EASE and VISTA Framework

VPO and EASE are central to the success of this work, and this chapter will give an overview of the essential principles of both of these techniques. We will also describe the functionality of VISTA from a user's viewpoint.

Figure 1 illustrates the flow of information in VISTA and gives the reader a better idea as to how VPO and EASE are associated with VISTA. The user initially specifies a file to be compiled. The user then specifies requests through the viewer, which include sequences of optimization phases, user-defined transformations, queries and performance measures. The compiler performs the specified actions and sends the program representation information back to the viewer. In response to a request to get performance measures, the compiler in turn requests EASE to instrument the assembly with additional instructions to get static and dynamic instruction counts. This instrumented code when executed returns the performance counts. When the user chooses to terminate the session, VISTA saves the sequence of transformations in a file so they can be reapplied at a later time, enabling updates to the program in multiple sessions.

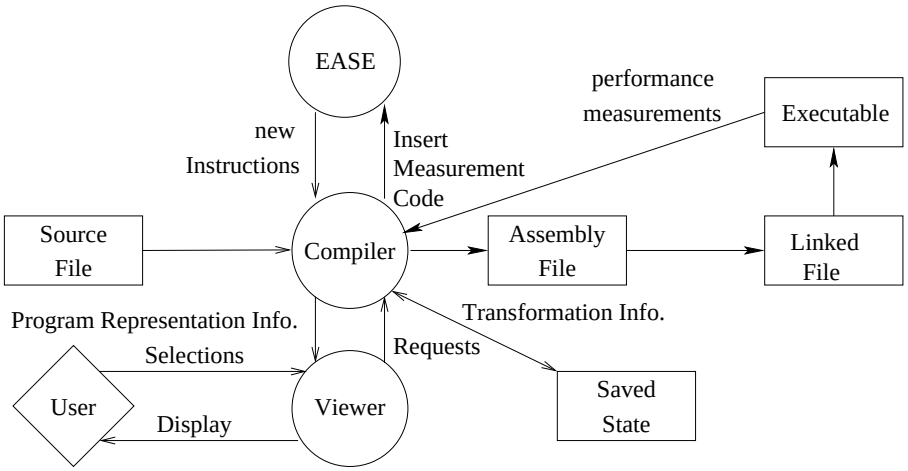

Figure 1: VPO, EASE and VISTA interaction

2.1 VPO (Very Portable Optimizer)

VISTA's optimization engine is based on VPO, as it has several properties that allowed easy integration into VISTA. VPO employs a paradigm of compilation that has proven to be flexible and adaptable - all code improving transformations are performed on a single target-specific representation of the program [5], called RTLs (Register Transfer Lists). RTLs are a low-level, machine and language independent representation that encodes machine-specific instructions. The comprehensive use of RTLs in VPO has several important consequences.

One advantage of using RTLs as the sole intermediate representation is that the synthesis phases of the compiler can be invoked in any order and repeatedly if necessary [6]. This largely eliminates code inefficiencies often caused by the ordering of the phases. In contrast, a more conventional compiler system will perform optimizations on various different representations. For instance, machine-independent transformations are often performed on intermediate code and machine-dependent transformations are often performed on assembly code. Local transformations (within a basic block) are often performed on DAG representations and global transformations (across basic blocks) are often performed on three-address codes. Thus, the order in which optimizations are performed is difficult to change. To be of

any practical use, an interactive compilation system like VISTA has be flexible enough to allow the user to select optimization phases in an arbitrary order without many restrictions. VPO, was ideally suited to this compilation framework. 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 code registers, etc. VPO, in effect, improves object code. Machine-specific optimizations are important because it is a viable approach for realizing high-level language compilers that produce code that effectively balances target-specific constraints such as code-density, power consumption and execution speed.

The second advantage of using RTLs is that it is easily retargetted to a new machine. Retargetability is key for embedded microprocessors where chip manufacturers provide many different variants of the same base architecture and some chips are custom designed for a particular application. To target VPO to a new machine, one must write a description of the architecture's instruction set, which consists of a grammar and semantic actions. The grammar is used to produce a parser that checks the syntax of an RTL, and accepts all legal RTLs (instructions) and rejects all illegal RTLs. It is easier to write a machine description for an instruction set than it is to write a grammar for a programming language. The task is further simplified by the similarity of RTLs across machines, which permits a grammar from one machine to be used as a model for a description of another machine. Since the general RTL form is machine-independent, the algorithms that manipulate RTLs are also machine-independent, which makes most optimization code machine independent. So the bulk of VPO is machine- and language- independent. Overall, no more than 15 percent of the code of VPO may require modification to handle a new machine or language.

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

The fourth advantage of VPO is that VPO's analysis phases (e.g. data flow analysis, control flow analysis) were designed so that information is easily extracted and updated.

This information can be useful to a user of interactive compilation system.

Finally, by using RTLs, the effect of an optimization can be easily understood since each RTL represents an instruction on the machine. Thus, the impact that each transformation has on performance can be easily grasped.

VISTA takes advantage of VPO's strengths. VISTA can graphically display the low-level representation of the program either in assembly or RTLs. The user can select the type and the order of transformations to be performed because phase ordering problems are to a great extent eliminated in VPO. Since RTLs are easily understood, it is useful to allow users to see the transformations applied at each step in VISTA. Best of all, understandable machine-independent RTLs make it easier for the users to specify the transformations by hand to further exploit special features of an architecture.

2.2 EASE (Environment for Architecture Study and Experimentation)

The EASE environment [7] is used in VISTA to collect performance measures at any stage in the compilation process in order to evaluate the improvements in code size and number of dynamic instructions executed at that stage. This gives the user a better perspective as to if the sequence of optimizations applied are giving the expected benefits or if he 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 attention on improving that portion of the program. It also helps VISTA automatically determine the best sequence of optimizations for a function.

The EASE environment was developed to integrate the tasks of translating a source program to machine instructions for a proposed architecture, imitating the execution of these instructions and collecting measurements. The environment, which is easily retargetted and quickly collects detailed measurements, facilitates experimentation with a proposed architecture and a compiler.

The first task in EASE is to translate the test programs to instructions for the proposed

machine. This task is accomplished by VPO, which was described in the previous section.

The second task is imitating the execution of code for the proposed architecture. To be able to evaluate an architecture, one should determine the effect of executing instructions from representative test programs for the architecture. If the architecture has not yet been implemented, then one must imitate this execution by other means. In the EASE environment an instruction for the proposed machine can either be generated as an assembly instruction for the proposed architecture or as one or more equivalent assembly instructions for an existing architecture. This conversion of an RTL to assembly language is the last step in the VPO compilation system. EASE can also be used to emulate architectural features that are not directly equivalent to features on an existing architecture. For instance, it can generate code for an architecture having more registers than the number of registers on the host machine.

The final step is to extract measures for the proposed machine in order to evaluate the new architecture's performance. To accomplish this EASE modifies the back end of the compiler to store the characteristics of the instructions to be executed and to instrument the assembly code with instructions that will either count the number of times that each instruction is executed or invoke a routine to record events that are dependent on the order of the instructions executed. This method is illustrated in Figure 2.

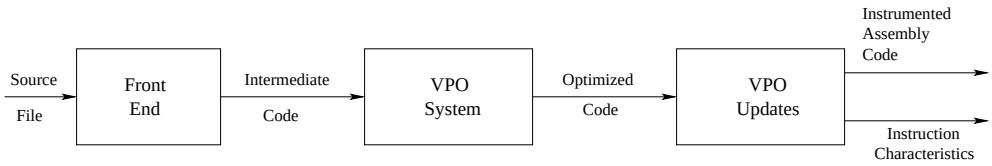

Figure 2: Method of Gathering Data in EASE

Many modifications were made to the VPO compiler system to support collecting measurements. The first modification of VPO to produce code for collecting measurements is to have the optimizer save the characteristics of the instruction that will be executed. As each assembly instruction is produced, the characteristics of the instruction are also written

to the assembly file. The second modification to VPO was to have the compiler instrument the assembly code after all optimizations have occurred to either increment counters or invoke measurement routines. Counters are used to obtain information that is independent of the order in which the instructions are executed, such as the number of times each type of instruction is executed. Measurement routines are invoked to record order-dependent events, which includes trace generation and analysis of memory references.

In VISTA, the EASE environment is mainly used to instrument the assembly code with instructions to determine the number of times each basic block is executed and to count the number of instructions in a function. This instrumentation of code is then used to determine the static and dynamic measures at any point in the compilation process. Comparing this with previous measures gives us the current improvement as compared to the baseline where no optimizations were performed.

2.3 VISTA (VPO Interactive System for Tuning Applications)

In this section we will review the VISTA framework, upon which the current work is based. VISTA is a new code improvement paradigm that was initiated to achieve the cost/performance trade-offs (i.e. size, power, speed, cost etc.) demanded for embedded applications. A traditional compilation framework has a fixed order in which the code improvement phases are executed and there is no control over individual transformations, except for compilation flags to turn code improvement phases on or off. In contrast, VISTA gives the application programmer the ability to finely control the code-improvement process.

The developers of VISTA had the following goals when developing the VISTA compilation framework. First, the user should be able to direct the order of the compilation phases that are to be performed. The order of the code-improvement phases in a typical compiler is fixed, which is unlikely to be the best order for all applications. Second, hand-specified transformations should be possible. For instance, the user may provide a sequence of instructions that VISTA inserts and integrates into the program. Third, the

user should be able to undo code-improving transformations previously applied since a user may wish to experiment with other alternative phase orderings or types of transformations. In contrast, the effects of a code transformation cannot be reversed once it is applied in a typical compiler. Finally, the low-level program representation should appear in an easily readable display. The use of dynamically allocated structures by optimizing compilers and the inadequate debugging facilities of conventional source level symbolic debuggers makes it difficult for a typical user to visualize the low-level program representation of an application during the compilation process. To assist the user when interacting with the optimization engine, VISTA should provide the ability for the user to view the current program representation and any relevant compilation state information (i.e. live registers, available registers, def-use information etc.) and performance metrics.

In addition to achieving these goals, VISTA was shown to have several other uses. First, VISTA can assist the compiler writer to develop new low-level code-improving transformations. The ability to specify transformations by hand and obtain performance measurements can help a compiler writer to prototype new low-level code-improving transformations. The ability of viewing low-level representations can help a compiler writer diagnose problems when developing new transformations. Second, it can help compiler writers understand the interactions and interplay of different optimizations. Finally, an instructor or educator teaching compilation techniques can use the system to illustrate code-improving transformations to students.

VISTA provides the user with the functionality for viewing the low-level representation, controlling when and where optimization phases are applied, specifying code-improving transformations by hand, reviewing previously applied transformations, reversing previously applied changes, proceeding to the next function and supporting multiple sessions. In addition, the user may wish to collect some preliminary results about the performance of the generated code, which can be accomplished by producing assembly that is instrumented with additional instructions that collect a variety of measurements during the program's execution. VISTA also allows the user to limit the scope in which an optimization phase will be applied. The scope can be a loop or a set of basic blocks. The user can first tune

critical portions of a function, and then use the resources (e.g. registers) that remain to optimize the rest of the function.

We have made several important enhancements to VISTA that facilitate the selection of effective sequences of optimization phases. These enhancements include automatically obtaining performance feedback information, the use of structured statements for applying optimization phases, and the automatic evaluation of performance information for selecting optimization phase sequences. These enhancements are described in the following sections of this report.

Chapter 3

Interactively Obtaining Performance Information

The ability for a user to acquire a measure of the performance of the program being compiled is a very important feature for any environment providing interactive compilation. Examining the performance measures at various stages during the compilation can give the user a clear image of how the code improvement process is progressing. In fact, actual program performance measures is the only concrete basis the user has when hand tuning the code. It can also indicate to the user if he/she can stop the optimization process, when the performance delivered by the program has reached the desired level. VISTA gives the user the option to get both the static and dynamic performance counts at any stage at the click of a button. Here, static counts mean the count of the number of static instructions in that function, i.e. an indication of the code size when each instruction is the same size. Dynamic counts imply the number of instructions executed during a particular run of the program on some representative input data (which the user has to provide). More accurate measures, such as operation or simulation times, could also potentially be used. VISTA also tells the user the percentage of time each basic block is executed, so that the user can concentrate his attention to the more critical portions of the code, like the blocks comprising the inner loops in that function. This can be easily accomplished in VISTA by restricting the scope

of the optimizations to only those critical blocks.

3.1 Options in VISTA to get Measures

Figure 3 shows the options provided in VISTA, which include the two options which enable the user to get performance counts at any stage. These options are:

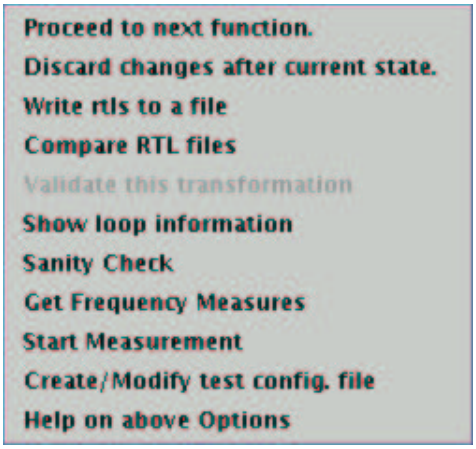

Figure 3: Measurement Options in VISTA

1. **Get Frequency Measures:** Selecting this option commands the compiler to immediately execute the program once and get the frequency of execution of each basic block. The program is executed using sample input which the user must provide. The compiler sends a count of the number of times each basic block is executed, which the viewer reads and converts into percentages. These are then displayed at the top of each basic block (see figure 4). This allows the user to identify the critical regions of a function.

2. **Start/Stop Measurements:** This option allows the user to obtain both the static and dynamic counts after every optimization phase. This is a toggle option, which allows the measurement process to be stopped/resumed at any point. When measurements

are started, the compiler executes the program once and records the static and dynamic measures at that stage, which are used as the baseline measures. After each optimization phase the program is again executed and measures are collected. These are sent to the viewer which in turn compares them with the baseline measures taken at the start of the measurement process. The relative improvements in code size and instructions executed are then displayed in the viewer window. This information allows the user to quickly gauge the progress that has been made in improving the function. The viewer still displays the execution frequency of each basic block (see Figure 4).

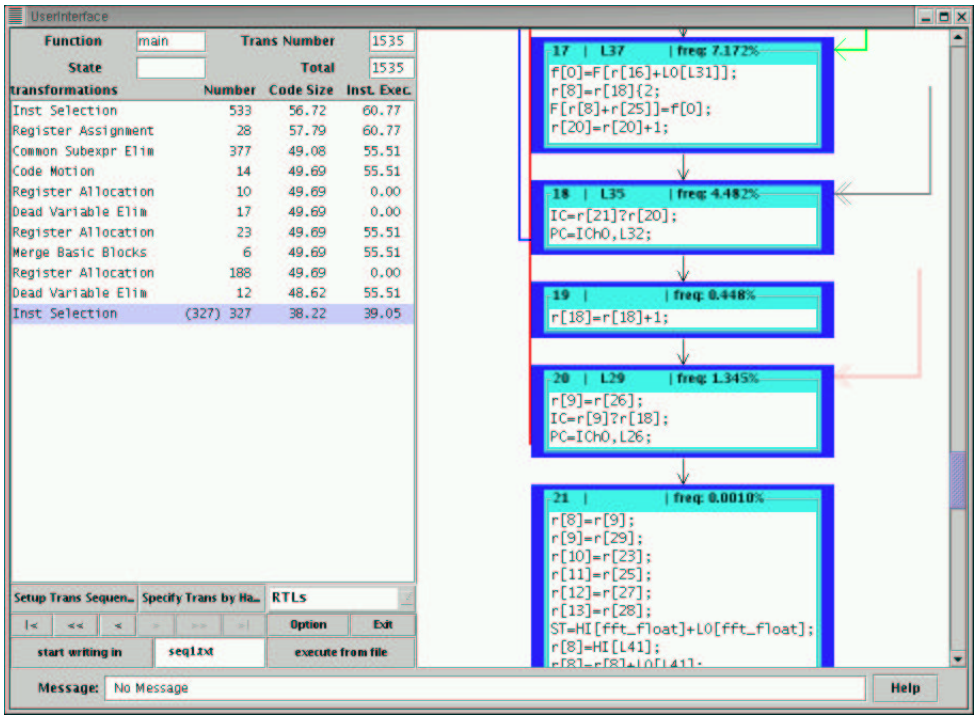

Figure 4: VISTA with Measurements Turned On After Each Phase

3.2 Specifying Configuration File

As mentioned earlier, to get the dynamic measures the user has to provide VISTA with a set of commands to link and execute the program and produce the output file. This information is assumed, by VISTA to be present in a file, before the user specifies any measurements. In case VISTA is unable to find this file, it prompts the user to enter the required information at the very start of the measurement process. The window that is opened is shown in Figure 5.

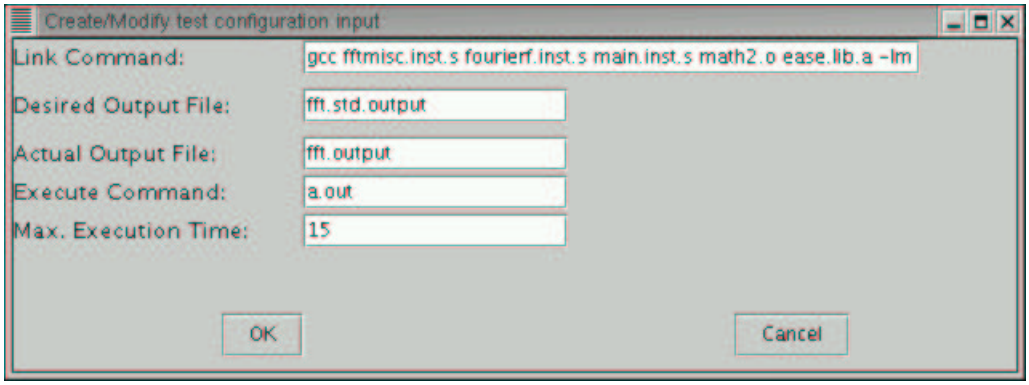

Figure 5: Test Configuration Window

The fields in this window are:

Link Command: This is the command to be used to link the current file and produce an executable. If this file requires other source files or libraries to be linked, then these also have to be specified here.

Desired Output File: This is the output data that should be produced whenever this program is executed. The compiler assumes that such correct output data is provided to it, so that it can compare the "current output data" with the "known correct output data" and check for inconsistencies. This is very important especially for a new transformation, to ensure that the output produced is correct. The compiler must never perform a transformation that results in incorrect output.

Actual Output File: This is the name of the output file produced after running the executable.

Execute Command: This command is used to execute the program and produce the output file.

Max. Execution Time: An incorrect program transformation, performed by the compiler, may result in the program going into an infinite loop. This is certainly undesirable. This field specifies the maximum time the compiler should allow the program to finish executing. If, by this time the program has not finished, then it is terminated. On termination, the compiler assumes that the program state is incorrect and the user is notified.

3.3 Getting Performance Measures

When the compiler gets a request to determine the performance measures, it performs a series of steps as shown in the Figure 6. In case the program consists of multiple input files, which need to be linked together to produce the executable, the user must ensure that an instrumented assembly file (.inst.s file) corresponding to each input file is present before the measurement process is initiated. This can be done by simply running each input file through the interactive compiler without performing any optimizations.

The compiler always stores a pointer to the start of the current function in the current input file (.cex file), which is produced by the code expander. No optimizations have been performed on this file. For each function, the compiler always generates both the assembly code (in the .s file) and the instrumented assembly code (in the .inst.s file), which is the assembly code instrumented with additional instructions to collect performance measures. At the start of the measurement process, the instrumented assembly file generated at that point (for the preceding functions) is stored as the compiler would now need to generate assembly for the remaining functions in the file in order to be able to execute the program. Later, after the program is executed and measurements taken, the stored instrumented

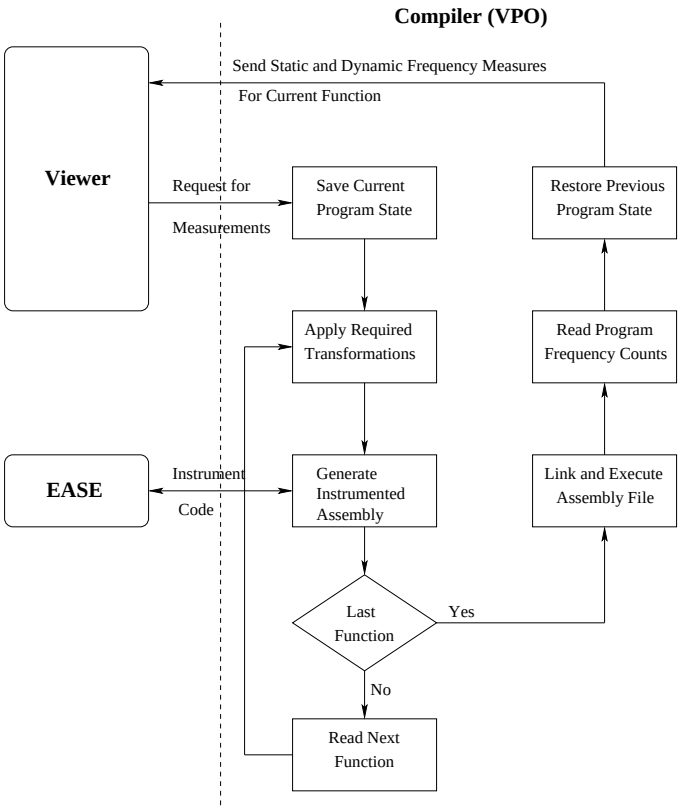

Figure 6: The Measurement Process

assembly file is again renamed to the current output file. We also have to store the list of transformations applied to this function up to that point so that they can be reapplied after getting the measurements, to get back to the same program state as it was before the current measurement process.

There are some required transformations that have to be always performed before assembly code for a function can be generated. These are *register assignment* and *fix entry-exit*. Temporary values are initially stored in pseudo registers by the code expander. These have higher numbers than the hardware registers. The register assignment phase assigns pseudo registers to the hardware registers. Also, after most optimizations have been applied, typically several adjustments have to be made to the entry and exit points of a

function. These adjustments include allocating/deallocating space on the run-time stack and saving/restoring registers. This is done in the fix entry-exit phase. Thus, it is obvious that correct assembly code cannot be generated unless these compulsory phases have been applied. It may be possible that these have not been performed on the current function as yet. Before outputting assembly for measurements, we check if these phases have been applied and if not they are performed now.

The instrumentation of code by EASE is done as the next stage. As explained in an earlier section, to take measurements, the EASE environment instruments the code with additional instructions, which among other things increment counters associated with each basic block. The instrumented assembly is output to an instrumented assembly file. After code for the function is generated, all the data structures for holding the state of this function are cleared and re-initialized. We now have to read in the remaining functions in the file, apply the required transformations (register assignment and fix entry-exit) for each remaining functions and output the assembly.

We now have an assembly file instrumented with additional instructions by EASE and ready to be executed. The configuration file containing the link and execute commands, described earlier, is opened and commands read. The assembly file is linked and then executed. Upon execution, the instrumented EASE code in the function produces a new counts (.cnt) file containing measurement information. This file can be directly read and interpreted to extract the frequency counts and number of executable instructions in each basic block. Alternatively, we have a program called *genreport* which does this work for us and produces the output in report format. The static and dynamic counts for each function in the file are displayed here. The frequency counts for each individual basic block still have to be read directly from the counts file.

We now have to get back to the same exact state for the current function as it was when measurements were initiated. The stored pointer to the current function in the input file is used to do a *file seek* to reset the file pointer and the function is read back into VPO's internal data structures. The user might have applied some transformations to this function before he started the measurement phase. These transformations have to be performed as

well. The compiler stores the transformations, not as individual phases (like instruction selection), but as a sequence of small changes during each phase (like delete RTL, insert RTL, modify RTL etc.). This is done so that the user can view each individual change applied to the function in the viewer and to also support manually specified transformations. After these changes are reapplied the function is set to the same state as it was when we started the measurement. The measures are then sent to the viewer, which displays it in its windows.

Figure 4 shows a snapshot of the viewer with a history of the sequence of optimization phases displayed. Note that not only is the number of transformations associated with each optimization phase displayed, but also the improvements in instructions executed and code size are shown.

Chapter 4

Support For Interactive Code Tuning

In conventional compilers a programmer has little control over the order in which code-improving transformations are applied. It has been shown that a fixed sequence of optimizations may not produce optimal code for all programs, or even for individual functions in a program. The problem of automatically generating acceptable code for embedded microprocessors is much more complicated than for general-purpose processors. First, embedded applications are optimized for a number of conflicting constraints. In addition to speed, other common constraints are code size and power consumption. An optimization sequence tuned for producing faster code, may in fact increase the code size, which may be undesirable. For many embedded applications, code density and power consumption are often more critical than speed. In fact, in many applications, the conflicting constraints of speed, code density and power consumption are managed by the software designer writing and tuning assembly code. Unfortunately, the resulting software is less portable, less robust, and more costly to develop and maintain.

Automatic compilation for embedded microprocessors is further complicated because embedded microprocessors often have specialized architectural features that make code improvement and code generation difficult [8, 9]. While some progress has been made in

developing compilers and embedded software development tools, many embedded applications still contain substantial amounts of assembly language because current compiler technology cannot produce code that meets the cost and performance goals for the application domain. The code improvement paradigm provided by VISTA, has the potential to achieve the cost/performance trade-offs demanded for embedded applications. The earlier version of VISTA [4] provided support for interactive code optimization. In this section, we describe the feedback-based interactive performance tuning provided in this version of VISTA.

4.1 Getting Performance Measures

The earlier version of VISTA, enabled the user to specify a sequence of optimizations in any order supported by the compiler and also allowed some user-specified transformations (e.g. for exploiting advanced architectural features which the compiler does not). But, it did not automatically give the user any feedback about the improvements in code produced after applying the optimizations. Whenever measurements are needed, an instrumented executable for the program has to be produced, executed and measures are obtained as explained in the previous section. In the earlier version of VISTA, a user could accomplish this in a series of steps. First, after applying the desired sequence of optimizations, VISTA is exited causing the instrumented assembly file to be produced. Next, commands are issued to assemble, link and execute the program. This will produce the *counts* file containing the frequency of execution of each basic block. Next time VISTA is re-invoked it will automatically reapply the previous transformations to reach the same point in the compilation. Also, now it will detect the presence of the *counts* file, and will send that information to the viewer. The viewer displays relative execution frequency of each basic block in that block's header.

We felt the need to automate this process in order to give the user instant feedback regarding the program's performance after each transformation, or whenever the programmer felt necessary. This ability is provided by the two options, described in the previous

section, namely:

1. **Start/Stop measurements:** Start getting code-size and instruction count measurements after each transformation phase.

2. **Get Frequency Measures:** Get the relative execution frequency of each basic block at any stage during the compilation process.

Thus, the programmer now does not have to rely on pure intuition to guide the code improvement process. The instant performance feedback provided by VISTA alleviates the programmer's job of transforming the program to one which gives acceptable performance. Currently the viewer shows improvements in both the code size as well as dynamic instruction counts. It should be possible to display relative improvements based on any other criteria.

4.2 New Constructs in VISTA

We also found that it is 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 of optimization phases to be applied until no more improvements can be 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.

Prior support in VISTA for conditionally applying optimization phases was only a low-level branch operation (i.e. if changes goto <label>) [4]. We now provide support for testing if changes have occurred in the form of four structured control statements which the user can interactively specify.

1. **if-changes-then:** This statement is similar to the *if-then-endif* construct in higher level programming languages. It performs the optimizations specified in the *then* block only

if the transformation immediately preceding the *if* produces changes to the program representation. This is converted by the viewer into a low-level sequence of requests:

```
if-changes-then :: <IF-FALSE-GOTO-TRANS>
                          transforms in if-block
```

The compiler interprets this to mean that if *changes* (in the previous phase) is false, then jump over the number of *transforms in if-block*.

2. **if-changes-then-else:** This statement is similar to the *if-then-else* construct. Depending on if the phase preceding the *if* produces changes, either the *then* branch is taken or the *else* branch is taken. The low-level sequence corresponding to this construct is:

```
if-changes-then-else :: <IF-FALSE-GOTO-TRANS>
                              transforms in if-block
                        <GOTO-TRANS>
                              transforms in else block
```

This tells the compiler that if *changes* (in the preceding phase) is false then jump over the transformations in the *if-block* and directly go to the transforms in the *else-block*. If *changes* is true, then the transformations in the *if-block* will be performed and then, on encountering the *unconditional goto*, control will be transferred to the transform after the *else-block*.

3. **do-while-changes:** This statement is a looping construct similar to *do-while* in high-level languages. The transformations in the *do block* are always performed at least once and are repeated as long as any transformation in that block produces any changes to the program being compiled. The low-level translation for this is:

```
do-while-changes ::  <BEGIN_LOOP>
                           transforms in while-block
```

```
                                <IF-LOOP-TRUE-GOTO-TRANS>
                            <END_LOOP>
```

To the compiler this means that after performing the transformations in the *while-block* if any of those made any changes to the program representation, then again go back to the first transform in the *while-block*. The *BEGIN_LOOP* and *END_LOOP* delimiters indicate the scope of each loop to the compiler. When multiple such *while* and *if* constructs are nested, these delimiters enables the compiler to check the proper flag (*changes* variable) and make the right branching decisions at each loop nesting level.

4. **while-changes-do:** This statement is similar to the *while* statement in high level languages. It is similar to the previous construct, except that application of the transformations in the *while block* for the first iteration depends on, if the preceding phase produced changes. The viewer will translate this to:

```
while-changes-do ::  <BEGIN_LOOP>
                         <IF-LOOP-FALSE-GOTO-TRANS>
                             transforms in while-block
                         <GOTO-TRANS>
                     <END_LOOP>
```

For the first iteration the compiler checks if the preceding phase made changes. For all future iterations the compiler checks if any of the phases in the *do block* make any change to the program representation. As long as any phase makes a change in any iteration, the compiler executes that loop again. After the last iteration control is transferred to the command after the *END_LOOP* command. As in the previous construct *BEGIN_LOOP* and *END_LOOP* are used to increment and decrement the loop nesting levels.

These statements can be nested within one-another. The introduction of these structured statements will make the selection of sequences of optimization phases more convenient to the user. Figure 7 illustrates this 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 loop iteration, the compiler will perform register allocation. Instruction selection will only be performed if register allocation allocates one or more live ranges of a variable to a register. 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. Thus, in effect we are providing an optimization phase programming language.

These operations are converted by the viewer into a low-level sequence of requests which the compiler interprets. VPO applies the sequence and sends each resulting change to the program representation back to the viewer. The process continues until a stop operation is encountered. The following list reflects the operations to be performed by the selections shown in Figure 7 .

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

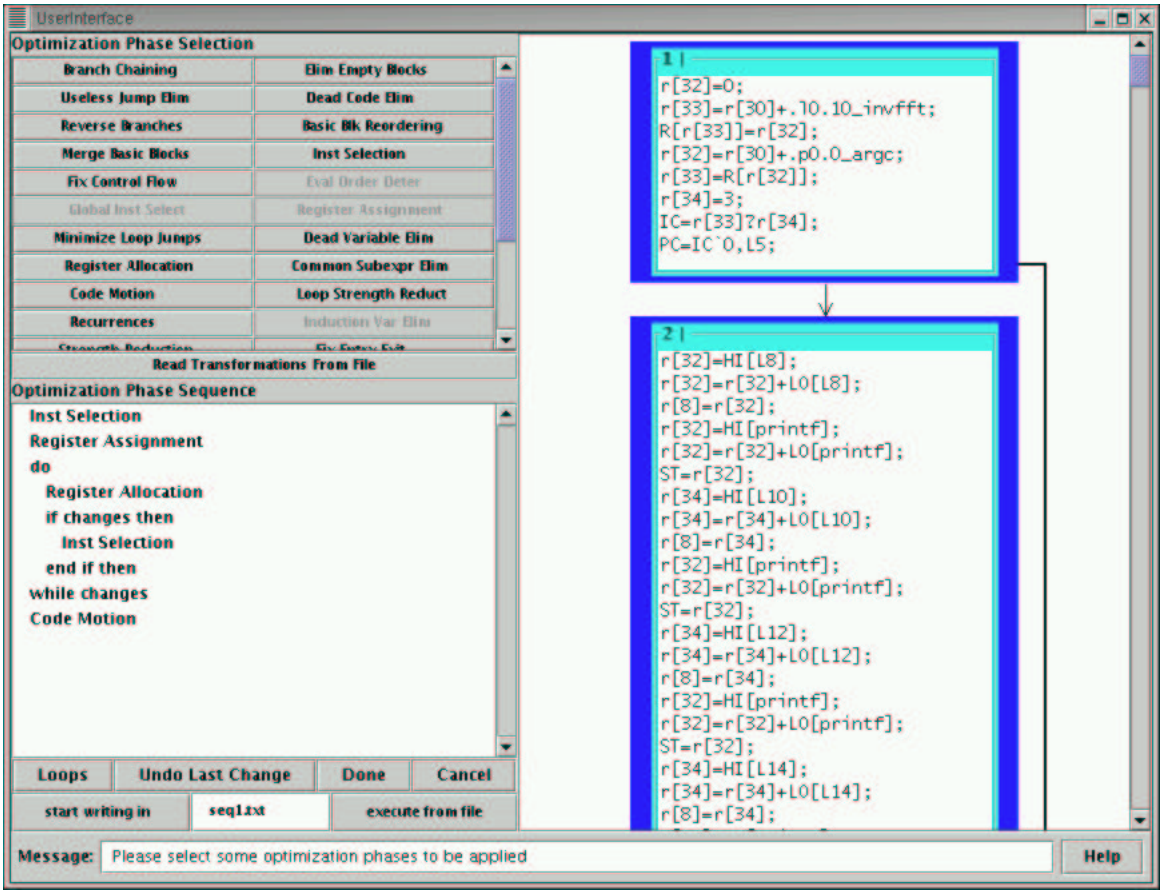

Figure 7: Interactively Selecting Optimization Phases

10. Stop

Figure 8 shows the resulting viewer state after the above sequence of optimizations have been applied. In the figure, after the first invocation of register allocation, instruction selection did not produce any changes. Dead variable elimination (which is actually dead variable identification) is sometimes performed as side-effect of register allocation automatically by the compiler.

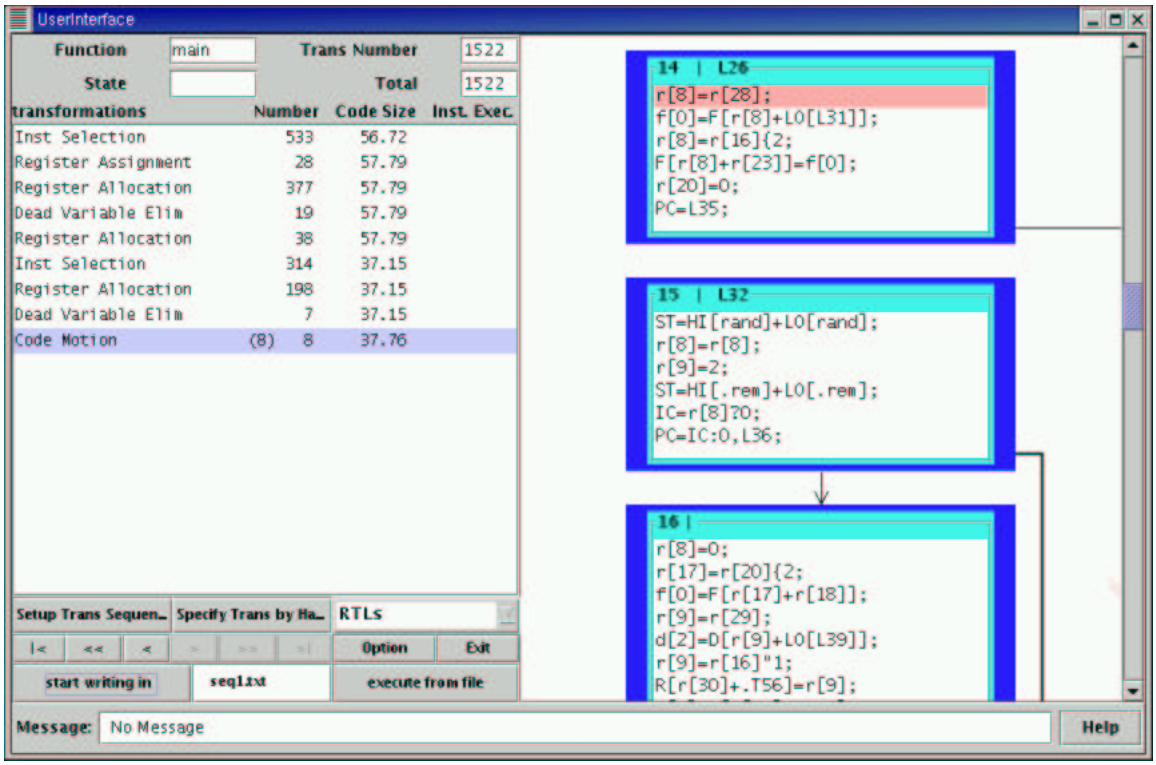

Figure 8: Interactively Selecting Optimization Phases

Chapter 5

Support for Performance Driven Automatic Code Tuning

The interactive performance tuning approach detailed in the preceding section, in spite of being very powerful, requires a lot of user intuition and effort to steer the optimization process to obtain acceptable performance. The programmer may make poor selections and may have to undo some earlier optimizations and try other sequences to check if it gives better performance. We felt it was necessary to provide the user with another technique which can come up with an effective sequence of optimizations automatically. We have added two structured constructs in VISTA which have the ability to automatically compare two or more sequences and determine which is most beneficial. The first is the *select-best-from* statement and the other is *select-best-combination*.

5.1 Constructs for Automatic Selection of Optimization Sequence

The constructs to automatically select optimization sequences are described in this section.

5.1.1 Select Best From

The *select-best-from* statement is illustrated in Figure 9. As seen from the figure, the user has selected 2 different sequences of optimizations separated by an *or*. The programmer only wants one of the two sequences to be applied to the program, the one giving better performance. The viewer first converts this statement into a low-level form to send to the compiler.

```
select-best-from :: <SELECT-BEST-FROM>
                         optimization sequence 1
                    <OR>
                         optimization sequence 2
                    <DONE-SELECT>
```

Note that the user can select any number of different optimization sequences separated by <OR>. The viewer then prompts the user to select the weights between instructions executed and code size, where the relative improvement in each is used to determine the overall performance. The compiler can make a choice of the better sequence based on only the code size, only dynamic instruction counts or any combination of both, as per the weight selected by the user. Figure 10 shows the window that pops up during *select-best-from* to enable the user to specify the relative weights.

When the compiler gets this command, it first stores the number of transformations applied to this function up to this point. This will be used to get the program state back to what is was before starting this construct. The compiler then applies the transformations in the first sequence and evaluates the program performance, according to the selected criteria. For this the compiler has to assemble, link and execute the current program as stated in an earlier section. It then gets the program state back to what it was when execution of this construct was initiated. This can be easily done by reading back the current function again and only applying the transformations until a specific point in the compilation process, in this case all transformations until the start of the *select-best-from* statement. This process is repeated for all the alternative optimizations sequences. After evaluating each sequence,

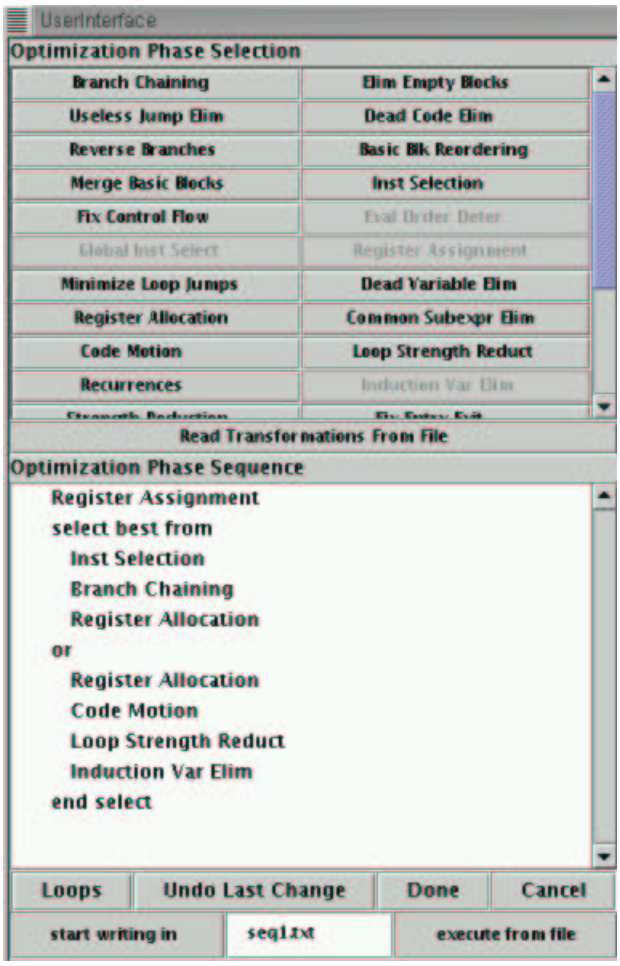

Figure 9: Selecting the Best of a Set of Specified Sequences

its performance is compared to the performance of the best sequence found so far. If the current sequence is better, then it is made the new best sequence. After all the sequences are evaluated for their performance, we have identified the best sequence, which is reapplied to the program and the new program state is sent to the viewer. Here we need to always reapply the best sequence, even if the best sequence found is the last sequence in the *select best from* construct. This is because while testing for the best sequence no messages were sent to the viewer. When we reapply the best sequence the changes made by only this sequence are sent to the viewer. Reapplying the best sequence would also be needed in

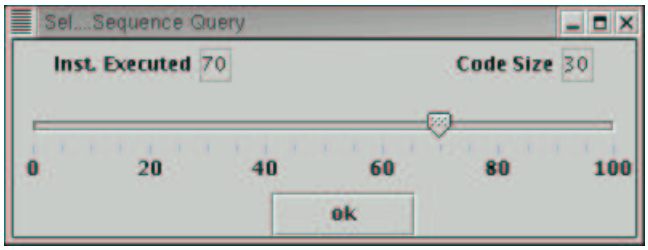

Figure 10: Selecting the Relative Weights during Select Best From

the case when the flag for getting performance measures after each phase is set. Note that while determining the best sequence we are only getting the measures after each complete sequence as compared to getting measures after each individual phase.

5.1.2 Select Best Combination

The other option is *select-best-combination*, which accepts a set of optimization phases and attempts to discover the best sequence of phases. Figure 11 shows an illustration of how this works. Here the user has selected a sequence consisting of 5 optimization phases. The user is attempting to discover the best ordering of this sequence of five optimization phases.

The basic intuition is to try all combinations of the specified transformations, evaluating each one for its performance and coming up with the best sequence, which could be reapplied to the program in a manner similar to *select-best-from*. The only problem here is the search space which grows exponentially based on the length of the input sequence. Therefore it was necessary to somehow manage the exponential search space.

Figure 12 shows the different options that we provide the user to control the search for the best sequence. The *No. of Phases* field shows us the length of the input sequence of optimizations we selected. The *Sequence Length* can be different from the *No. of Phases* in two of the three *Search Options*. VISTA also allows the user to select the weights between instructions executed and code size, similar to the option provided during *select-best-from*. The user can thus optimize the code based on only code-size, only instruction counts or any

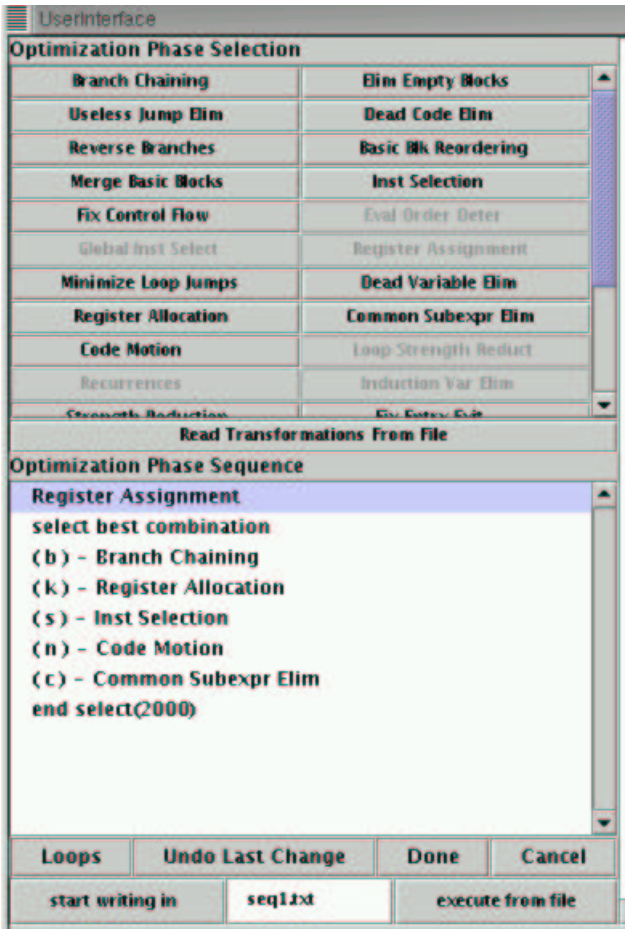

Figure 11: Selecting the Best Sequence from a Set of Optimization Phases

combination of these two performance measures. VISTA also offers the users three different *Search Options*:

1. **Exhaustive Search:** An exhaustive search results in all possible sequences being attempted. If the user has selected m distinct optimization phases with a sequence length of n, then there will be m^n different sequences attempted. An exhaustive search may be appropriate when the total number of possible sequences can be evaluated in a reasonable period of time. But in many cases the search space is too great to feasibly evaluate all possible sequences of optimization phases. The next two options

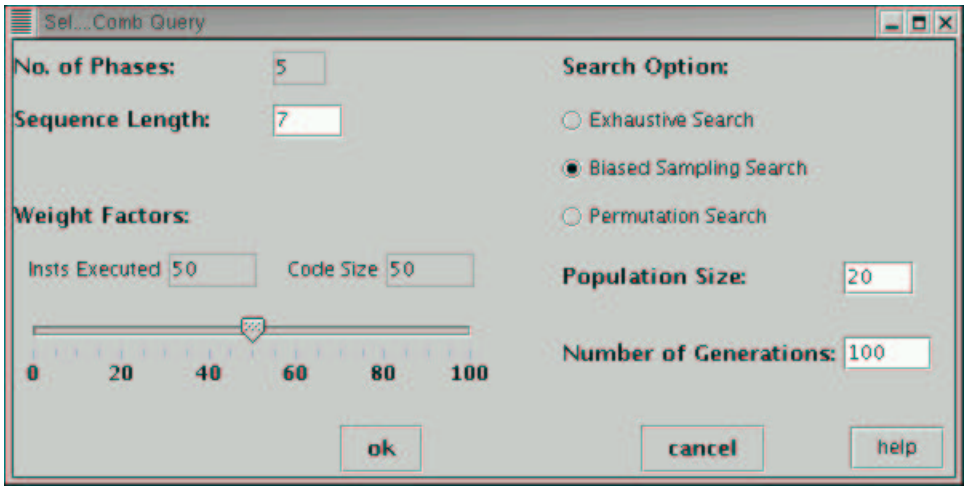

Figure 12: Selecting Options to Search the Space of Possible Sequences

attempt to intelligently probe the search space for effective sequences.

2. **Biased Sampling Search:** In this option we apply a genetic algorithm to probe the search space to find an effective sequence quickly. The goal of using a genetic algorithm is to find an effective sequence given a restricted number of sequences to attempt. Genetic algorithms are basically search algorithms designed to mimic the process of natural selection and evolution in nature. We define a few terms related to genetic algorithm here. For a more detailed overview refer to Appendix A.

   A population consists of a fixed number of members, or *chromosomes*. Each chromosome is a fixed-length string of *genes*. The *fitness* of a chromosome is some measure of how desirable it is to have that chromosome in the *population*. A *generation* is a time step in which several events occur. Some of the most 'unfit' chromosomes die and are removed from the population. To replace these chromosomes, some number of *crossover* operations are applied to the population. A crossover is an operation analogous to mating or gene splicing. It combines part of one chromosome with part of another chromosome to create a new chromosome. Finally, some amount of *mutation* occurs in the population, in which individual genes are changed randomly with

some (typically low) probability. It is also possible to have *elitism* in the population in which some of the most fit genes are immune from mutation between generations. In our case, a chromosome corresponds to a sequence of optimization phases and a gene to an individual phase in the sequence. Some number of optimization sequences together form a population. The operations of crossover and mutation are defined so that the genetic algorithm finds good optimization sequences quickly. Fitness value of a chromosome is the performance measures obtained when that sequence is applied to the program. Thus a lower fitness value (lower code-size, lower number of instructions executed), is considered a better sequence. The actual genetic algorithm we used in described in Appendix B.

Thus, after selecting *Biased Sampling Search*, the user also needs to specify the Population Size and the Number of Generations for the genetic algorithm. The default values of 100 generations and a population size of 20, would produce 2000 sequences of optimization phases, to be applied on the current function.

3. **Permutation Search:** The permutation search attempts to evaluate all permutations of the specified length. Unlike the other two searches, a permutation cannot have any of its optimization phases repeated. Thus the sequence length must be less than or equal to the number of distinct phases. A permutation search may be an appropriate option when the user is sure that each phase should be attempted at most once. This also results in reducing the number of attempted optimization sequences over an exhaustive search.

The viewer converts this control statement into a low-level sequence of requests to be sent to the compiler.

```
select best combination :: <SELECT-BEST-COMBINATION>
                                    sequence of optimizations
                                    list of options specified by user
                                <END-COMBINATION>
```

The compiler after receiving this request, saves the current transformations applied on the function thus far, similar to *select-best-from*. It then applies each sequence to the function, assembles, links and executes it to get the performance measures. The program state is rolled back to that before starting this construct. Depending on the search option selected by the user, the compiler determines the next sequence to be applied to the function. The sequence of optimizations giving the best program performance is saved. After all sequences are applied the compiler determines the best sequence and reapplies that sequence to the function and sends the resulting program representation to the viewer.

Performing these searches can be quite time consuming. Thus, VISTA provides a window showing the current status of the search. Figure 13 shows a snapshot of the status of the search that was selected in Figures 11 and 12. The window displays the percentage of the

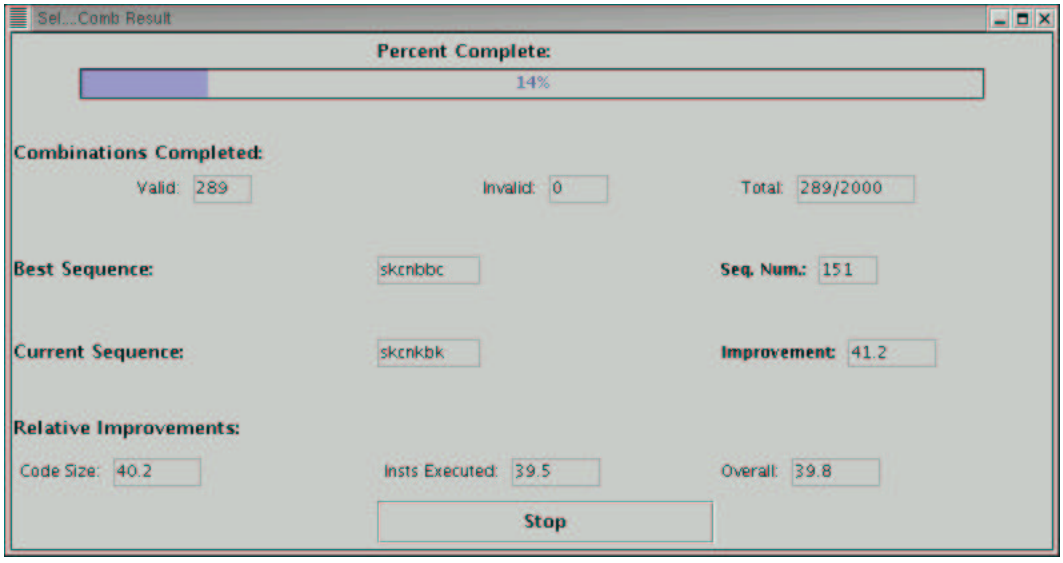

Figure 13: Window Showing the Status of Searching for an Effective Sequence

search completed, the number of valid and invalid sequences, an encoded representation of the sequence which was last tried and the best sequence so far along with their static and dynamic measures. All the improvement numbers shown in this figure are relative to the program state when select best combination was started. A sequence is declared to be

invalid if the sequence is such that it does not adhere to all the constraints imposed on the ordering of individual phases, e.g. if register allocation appears before register assignment in the sequence. We also declare a sequence invalid if it results in a program state which produces a linker error or upon execution does not produce the desired output or if it goes into an infinite loop. Ideally the second case for invalidating a sequence should never occur in a compiler. If such a sequence is encountered then a log of that sequence is maintained so the developers can try to resolve the problem. Table 1 describes each phase in the compiler and gives a designation (gene) of each phase that is used for displaying the sequence in the window in Figure 13.

It may be the case that the performance desired by the user is achieved by an optimization sequence early on in the search process. In that case it is not required to sit through the remaining sequences since that could take a very long time. VISTA provides the option of interrupting the search process whenever the user wants. If the process is interrupted then the compiler applies the best sequence found so far during the search process and returns the resulting program representation to the viewer.

Table 1: 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 an 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 in a legal instruction
fix control flow	f	Change code that computes a boolean value and then uses that value to jump, to code that just jumps
eval order determination	o	Reorder instructions within a basic block to calc. expressions that require the most registers first
global instruction selection	g	Perform instruction selection across basic block
register assignment	a	Assign pseudo registers to hardware registers
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 elim.	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 subexpr. elim.	c	Eliminates fully redundant calculations
code motion	n	Move loop invariant code into the preheader
loop strength reduction	i	Replace register increments by constant values on each iteration of a loop by simple increment
recurrences	p	Avoid recurrences in loops by retaining values in registers across iterations of the loop
induction variable elimination	v	Removes unnecessary register increments after loop strength reduction
strength reduction	q	Replaces an expensive instruction with one or more cheaper ones
fix entry exit	w	fix entry and exit of a function to manage the run time stack
instruction scheduling	t	Rearrange the order of instructions within a basic block in an attempt to reduce pipeline stalls
fill delay slots	u	Fill the delay slots after transfer of control instructions in RISC machines

Chapter 6

Experimental Results

In the previous sections we have described the VISTA framework and the support it provides for feedback based performance tuning. In this section we describe 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 [10]. We used one benchmark from each of the six categories of applications. Descriptions of the programs we used are shown in Table 2.

Table 2: 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

Our target architecture for these experiments was the SPARC, as we do not currently have a robust version of VISTA targeted to an embedded architecture. Using a genetic algorithm to find effective optimization phase sequences can result in thousands of sequences

being applied. This provides a severe stress test for any compiler. In the future we plan to test VISTA's ability to find effective optimization phase sequences on embedded processors.

Our experiments have many similarities to the Rice study, which used a genetic algorithm to reduce code size [11]. 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 more improvements can be obtained. Figure 14 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. Initially it was not obvious how to best assess VISTA's ability to find effective optimization sequences as compared to the batch VPO compiler. One complication is that the *register assignment* (assigning pseudo registers to hardware registers) and *fixed entry exit* (fixing the entry and exit of the function to manage the run-time stack) phases are required, which means that they have to be applied once and only once. Many of the other phases shown in Figure 14 have to be applied after *register assignment* and before *fix entry exit*. Thus, we decided to use the genetic algorithm to find the best sequence of improving phases that can be applied between these two required phases. These candidate sequences involve fourteen unique phases, which can be applied in any order between the two required phases. These phases are: *instruction selection*, *minimize loop jumps*, *merge basic blocks*, *dead assignment elimination*, *register allocation*, *common subexpression elimination*, *loop transformations* (which include *loop-invariant code motion*, *recurrence elimination*, *loop strength reduction* and *induction variable elimination*), *remove useless jumps*, *strength reduction*, *branch chaining*, *remove unreachable code*, *remove useless blocks*, *reverse jumps* and *block reordering*. For a description of what each of these phases do refer Table 1.

```
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
```

Figure 14: VPO's Order of Optimizations Applied in the *batch* mode

Another issue is 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 determined both the number of successfully applied optimization phases (those which affected one or more instructions in the compiled function) and the total number of phases attempted during batch compilation.

6.1 Batch Compilation Measures

Table 3 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 have anything significant to report we have designated such functions together as *unexecuted functions*. 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. Also, iteration 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 between *register assignment* and *fix entry exit*. Note that this sequence of phases was applied after attempting the optimization phases that precede *register assignment* in Figure 14. We found that the sequence of optimization phases selected before *register assignment* and after *fix entry exit* were much more consistent. One can see that the sequences of successful optimization

phases can vary greatly between functions in the same application. The next 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.

The last two columns in Table 3 depict that iteratively applying optimization phases had a significant impact on dynamic and static instruction count. We obtained this measurement by comparing the results of the default batch compilation to results obtained without iteration, which uses the algorithm in Figure 14 with all the `do-while's` iterated only once. The iteration impact result shows the power of iteratively applying optimization phases until no more improvement can be found. In particular the number of instructions executed is often reduced. The only cases where dynamic count increased was when *loop invariant code motion* was performed and the loop was either never entered or only executed once. In fact, we were not sure if any additional dynamic improvements could be obtained using a genetic algorithm given that iteration may mitigate many phase ordering problems. In the rest of this section we compare the results we obtained using a genetic algorithm to search for effective optimization sequences to the sequences found by the iterative batch version of VPO. For our genetic algorithm experiments we set the optimization phase sequence (chromosome) length to 1.25 times the length of the number of successfully applied optimization phases for each function. We felt this sequence length is a reasonable limit for each function and still gives us an opportunity to successfully apply more optimization phases than the batch compiler was able to accomplish. Note that the number of attempted phases for each function by the batch compiler far exceeded this length.

Table 3: Batch Optimization Measurements

program and size	function	% of dynamic	% of static	applied sequence and length	attempted phases	iteration impact % dynamic	iteration impact % static
bitcount (496)	AR_btbl_bitcount	3.22	3.86	kschsc (6)	53	-9.52	-9.52
	BW_btbl_bitcount	3.05	3.66	emsaks (6)	24	0.00	0.00
	bit_count	13.29	3.25	sksc (4)	42	-18.64	-14.29
	bit_shifter	37.41	3.86	sks (3)	26	-9.09	-5.26
	bitcount	8.47	10.16	ksc (3)	40	0.00	0.00
	main	13.05	19.51	sjmhksscllqsclllsllhsc (21)	125	-27.27	-8.16
	ntbl_bitcnt	14.40	3.66	sksc (4)	40	-11.10	-11.76
	ntbl_bitcount	7.12	8.54	ks (2)	24	0.00	0.00
	unexecuted func.	0.00	43.49	5.00	40.57	N/A	-9.57
	average			5.13	43.87	-11.10	-7.19
dijkstra (327)	dequeue	0.85	10.40	sksc (4)	40	0.00	0.00
	dijkstra	83.15	44.04	sjmhksclllcllsc (15)	71	-14.54	-4.44
	enqueue	15.81	12.84	shksc (5)	42	0.00	0.00
	main	0.06	22.94	sjmhksllsllsc (13)	71	-12.13	+3.23
	print_path	0.01	8.26	shksc (5)	41	0.00	0.00
	qcount	0.12	1.53	(0)	21	0.00	0.00
	average			7.17	47.67	-12.54	-1.36
fft (728)	CheckPointer	0.00	2.34	shksc (5)	41	0.00	0.00
	IsPowerOfTwo	0.00	2.61	sksc (4)	40	0.00	0.00
	NumberOfBits...	0.00	3.98	sjmhksc (7)	43	0.00	0.00
	ReverseBits	14.13	2.61	sjmksc (6)	42	0.00	+5.56
	fft_float	55.88	38.87	sjmhksclllllhsch (15)	57	-8.84	-7.64
	main	29.98	39.56	sjmhksclllehscll (17)	58	-1.90	-1.23
	unexecuted func.	0.00	10.03	3.00	65.00	N/A	-2.99
	average			8.29	49.43	-5.77	-3.95
jpeg (5171)	finish_input_ppm	0.01	0.04	(0)	21	0.00	0.00
	get_raw_row	48.35	0.48	sksc (4)	40	0.00	0.00
	jinit_read_ppm	0.10	0.35	ksc (3)	39	0.00	0.00
	main	43.41	3.96	sjmhksclschc (12)	70	-0.03	-1.14
	parse_switches	0.51	11.26	sjmhksc (7)	43	0.00	0.00
	pbm_getc	5.12	0.81	sksch (5)	41	0.00	0.00
	read_pbm_integer	1.41	1.26	sksc (4)	41	0.00	0.00
	select_file_type	0.27	2.07	sksec (5)	40	0.00	0.00
	start_input_ppm	0.79	5.96	sjmkschc (8)	55	0.00	0.00
	write_stdout	0.03	0.12	kss (3)	40	0.00	0.00
	unexecuted func.	0.00	73.69	6.27	44.35	N/A	-0.19
	average			6.08	44.13	-0.01	-0.19
sha (372)	main	0.00	13.71	sksclsl (7)	55	+6.67	+5.26
	sha_final	0.00	10.75	shksc (5)	41	0.00	0.00
	sha_init	0.00	5.11	sks (3)	25	0.00	0.00
	sha_print	0.00	3.76	sksc (4)	40	0.00	0.00
	sha_stream	0.00	11.02	sjmkscl (7)	42	0.00	0.00
	sha_transform	99.51	44.62	skscllllllllhsclllllllhs(23)	56	-11.46	-12.50
	sha_update	0.49	11.02	sjmhkscc (8)	56	-0.08	-2.78
	average			7.86	45.00	-11.44	-6.20
string-search (760)	init_search	92.32	6.18	sjmkscllscllhs (14)	70	-15.99	0.00
	main	3.02	14.08	sjmksclhsclhl (13)	69	+0.01	+2.08
	strsearch	4.66	7.37	skscllslscl (11)	69	-3.10	0.00
	unexecuted func.	0.00	71.44	14.00	66.57	N/A	+1.37
	average			13.50	67.40	-15.01	+1.28
average				8.01	49.58	-9.31	-2.94

6.2 Interactive Compilation Measures using Genetic Algorithms

There are a number of parameters in a genetic algorithm which can be varied to give algorithms quite different in performance and the best algorithm for a particular application only comes from experience and continuous fine tuning based on empirical results. Due to lack of aquaintance with a better substitute, we decided to use an algorithm based on the one used in the Rice experiments. The population size (fixed number of sequences or chromosomes) was set to twenty and each of these initial sequences is randomly initialized. 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) half of the population chosen at random. Each of the removed sequences are replaced by randomly selecting a pair of sequences from the upper half of the population and then performing a crossover operation on that pair to generate two new sequences. The crossover operation combines the lower half of one sequence with the upper 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 half of the population and a probability of 10% in the lower half. This was done for a set of 100 generations. Note that all these parameters can be varied interactively by the user during compilation as shown in Figure 12. For more on genetic algorithms refer appendix A.

Table 4 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 3, *unexecuted functions* 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 the effect on static and dynamic instruction counts 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 genetic algorithm was able to find a sequence for each function that either achieves the same result or obtains an improved result as compared to the batch compilation. In two cases the dynamic instruction count increased when optimizing for both speed and space. But in each case the overall benefit was improved since the percentage decrease in static instruction count was larger than the percentage increase in dynamic instruction count.

Table 4: Effect on Speed and Space Using the Three Fitness Criteria

program	functions	optimizing for speed		optimizing for space		optimizing for both	
		dynamic	static	dynamic	static	dynamic	static
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	**-25.29**	-12.50	-25.29	**-12.50**	**-25.29**	**-12.50**
	bit_shifter	**0.00**	0.00	0.00	**0.00**	**0.00**	**0.00**
	bitcount	**-2.00**	-2.00	-2.00	**-2.00**	**-2.00**	**-2.00**
	main	**-10.00**	-4.90	+20.00	**-11.76**	**-0.00**	**-7.84**
	ntbl_bitcnt	**-10.46**	-11.11	-5.82	**-5.56**	**-10.46**	**-11.11**
	ntbl_bitcount	**0.00**	0.00	0.00	**0.00**	**0.00**	**0.00**
	unexecuted func.	**N/A**	-2.55	N/A	**-3.73**	**N/A**	**-3.73**
	total	**-6.30**	-3.82	-2.02	**-5.42**	**-5.10**	**-4.82**
dijkstra	dequeue	**0.00**	0.00	0.00	**0.00**	**0.00**	**0.00**
	dijkstra	**-6.05**	-3.47	-3.02	**-4.86**	**-6.05**	**-6.25**
	enqueue	**0.00**	0.00	0.00	**0.00**	**0.00**	**0.00**
	main	**0.00**	0.00	+23.12	**-6.67**	**0.00**	**-2.67**
	print_path	**0.00**	0.00	0.00	**0.00**	**0.00**	**0.00**
	qcount	**0.00**	0.00	0.00	**0.00**	**0.00**	**0.00**
	total	**-5.03**	-1.53	-2.50	**-3.67**	**-5.03**	**-3.36**
fft	CheckPointer	**0.00**	0.00	0.00	**0.00**	**0.00**	**0.00**
	IsPowerOfTwo	**0.00**	0.00	0.00	**0.00**	**0.00**	**0.00**
	NumberOfBits...	**0.00**	0.00	+16.47	**-6.90**	**0.00**	**0.00**
	ReverseBits	**-0.93**	-5.26	0.00	**-15.79**	**0.00**	**-15.79**
	fft_float	**-6.14**	-4.59	+0.71	**-8.83**	**-6.14**	**-8.13**
	main	**-0.00**	-1.74	+0.44	**-5.21**	**+0.44**	**-5.21**
	unexecuted func.	**N/A**	-4.11	N/A	**-6.85**	**N/A**	**-6.85**
	total	**-3.57**	-3.02	+0.53	**-6.87**	**-3.30**	**-6.32**
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	**0.00**	0.00	0.00	**0.00**	**0.00**	**0.00**
	main	**-0.04**	-1.95	-0.03	**-3.90**	**-0.03**	**-3.90**
	parse_switches	**0.00**	-1.72	+2.17	**-2.06**	**0.00**	**-1.72**
	pbm_getc	**0.00**	0.00	0.00	**0.00**	**0.00**	**0.00**
	read_pbm_integer	**-3.54**	-1.54	-3.54	**-1.54**	**-3.54**	**-1.54**
	select_file_type	**-2.08**	0.00	-2.08	**0.00**	**-2.08**	**0.00**
	start_input_ppm	**0.00**	-0.65	0.00	**-0.65**	**0.00**	**-0.65**
	write_stdout	**-16.67**	-16.67	-16.67	**-16.67**	**-16.67**	**-16.67**
	unexecuted func.	**N/A**	-3.15	N/A	**-3.94**	**N/A**	**-3.94**
	total	**-0.08**	-2.67	-0.06	**-3.36**	**-0.07**	**-3.33**
sha	main	**-17.07**	-9.80	-17.07	**-9.80**	**-17.07**	**-9.80**
	sha_final	**0.00**	0.00	0.00	**0.00**	**0.00**	**0.00**
	sha_init	**0.00**	0.00	0.00	**0.00**	**0.00**	**0.00**
	sha_print	**-7.14**	-7.14	-7.14	**-7.14**	**-7.14**	**-7.14**
	sha_stream	**-6.65**	-29.27	+6.59	**-31.71**	**-6.65**	**-29.27**
	sha_transform	**-0.04**	-0.60	+6.07	**-3.01**	**0.00**	**0.00**
	sha_update	**-0.06**	-2.44	0.00	**-7.32**	**0.00**	**-7.32**
	total	**-0.04**	-5.38	+6.04	**-7.26**	**-0.00**	**-5.65**
string-search	init_search	**-0.37**	-6.38	-0.31	**-19.15**	**-0.37**	**-21.28**
	main	**-1.90**	-5.61	-1.90	**-10.28**	**+5.67**	**-7.48**
	strsearch	**-4.40**	-7.14	+0.61	**-7.14**	**-2.24**	**-3.57**
	unexecuted func.	**N/A**	-9.64	N/A	**-9.64**	**N/A**	**-9.64**
	total	**-0.61**	-8.68	-0.32	**-10.13**	**-0.28**	**-9.61**
average		**-2.61**	-4.18	+0.28	**-6.12**	**-2.30**	**-5.52**

Figures 15 and 16 show the overall effect of using the genetic algorithm for each test program on the dynamic and static results, respectively. The second measure for each function is obtained from the sequence found by the batch compilation when iteratively applying optimization phases and is normalized to 1. The results show that iteratively applying optimization phases has a significant impact on dynamic instruction count and less of an impact on the code size. 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 speed 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 speed and space showed that we were able to achieve close to the same dynamic benefits when optimizing for speed 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, and large functions with high dynamic counts can be optimized for both space and speed.

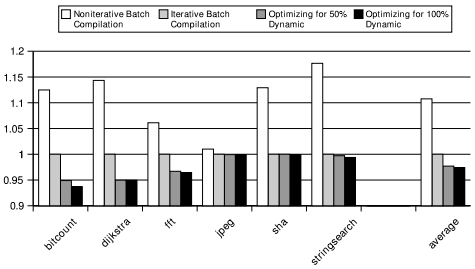

Figure 15: Overall Effect on Dynamic Instruction Count

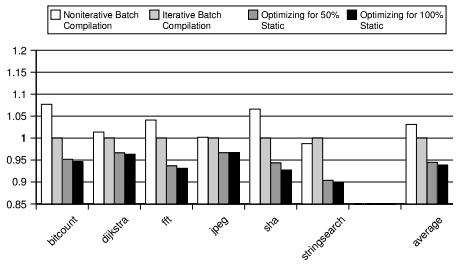

Figure 16: Overall Effect on static Instruction Count

The optimization phase sequences selected by the genetic algorithm for each function are shown in Table 5. The sequences shown are the ones that produced the best results for the specified fitness criteria. Sequences for a function shown in boldface varied between the different fitness criteria. Similar to the results in Table 3, these sequences represent the optimization phases successfully applied as opposed to all optimization phases attempted.

Some optimization phases listed in Table 1 are rarely applied since they have already been applied once before *register assignment*. These are the control-flow transformations that include phases 1, 2, 3, 5, 6 and 7 listed in Table 1. *Strength reduction* was not applied due to using 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 5 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.

We use a hash table containing fitness values and indexed by the chromosomes to reduce the search overhead. If the sequence has already been attempted, then we do not recompute it. We found that on average 54% of the sequences were found in the hash table. The functions with shorter sequence lengths had a much higher percentage of redundant sequences. A shorter sequence length results in fewer possible sequences and less likelihood that mutation will change a sequence in the population.

The overhead of finding the best sequence using the genetic algorithm for 100 generations with a population size of twenty required about 30-45 minutes for each function on a SPARC Ultra-80 processor. 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 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 less than one minute.

Table 5: Optimization Phase Sequences Selected Using the Three Fitness Criteria

program	functions	optimizing for speed	optimizing for space	optimizing for both
bitcount	AR_btbl_bitcount	chks	chks	chks
	BW_btbl_bitcount	ks	ks	ks
	bit_count	kchs	kchs	kchs
	bit_shifter	ks	ks	ks
	bitcount	ks	ks	ks
	main	**slljckllhschlllmc**	**chllkc**	**chllskllcllslch**
	ntbl_bitcnt	**ckshc**	**ksc**	**ckhsc**
	ntbl_bitcount	ks	ks	ks
dijkstra	dequeue	ksc	ksc	ksc
	dijkstra	**chlllchkljsc**	**chkllcllc**	**ckcllscllhsc**
	enqueue	**kshc**	**khsc**	**khsc**
	main	**shkllcljc**	**skhc**	**chkllcllc**
	print_path	kch	kch	kch
	qcount			
fft	CheckPointer	**hkc**	**kch**	**hksc**
	IsPowerOfTwo	kcs	kcs	kcs
	NumberOfBits...	**hkjcs**	**khs**	**hkjmsc**
	ReverseBits	**kcjhsc**	**kcs**	**ksc**
	fft_float	**jkcslllchclh**	**kslllhschc**	**kcllllcllhscllh**
	main	**skllllsjmch**	**shllllksc**	**skshc**
jpeg	finish_input_ppm			
	get_raw_row	kc	kc	kc
	jinit_read_ppm	kc	kc	kc
	main	**kchcj**	**kchc**	**kchc**
	parse_switches	**jsksch**	**kshc**	**jkshcm**
	pbm_getc	ksch	ksch	ksch
	read_pbm_integer	**kcs**	**kchs**	**kchs**
	select_file_type	rkch	rkch	rkch
	start_input_ppm	kschc	kschc	kschc
	write_stdout	ks	ks	ks
sha	main	kcsh	kcsh	kcsh
	sha_final	ksch	ksch	ksch
	sha_init	kc	kc	kc
	sha_print	chkc	chkc	chkc
	sha_stream	**kcj**	**chkc**	**chkcl**
	sha_transform	**cksllllllsclllllll**	**llllslllllkssc**	**skcllllllhcllllllsh**
		lhllllllch		**llllllsh**
	sha_update	**kshcjc**	**kschc**	**kschc**
string-search	init_search	**llkcjllhclc**	**ckhscllch**	**kslsllhs**
	main	**kslhlcjhc**	**skslhlc**	**kslhls**
	strsearch	**clskclhs**	**cksch**	**sllksls**

Chapter 7

Implementation Issues

The earlier sections described the main work accomplished as part of this thesis along with some empirical results. There were some changes made to the part already implemented in an earlier version of the compiler to make the compiler more stable and robust. Also, there were some other interesting implementation issues which could not be discussed in the earlier sections. This chapter is devoted for the mention of some such apparently minor issues, whose proper handling was necessary for the success of the main work.

7.1 Undoing Transformations

VISTA provides the ability to undo previously applied changes or transformations for two purposes. First, this ability can help the user to experiment with different orderings of phases and/or hand-specified transformations in an attempt to improve the generated code. Second, this feature gives the user an opportunity to change their mind if the earlier transformations are not giving the desired results. This work was done by Baosheng Cai [12] as part of his master's thesis. To do this, a linked list was used to keep a history of all the changes that occurred in the compiler. This resulted in a lot of additional code to store the information and also resulted in some space overhead. Moreover, this was difficult to maintain because knowing the exact things to store after each change is very difficult. Thus, although the work was done very meticulously, it had certain shortcomings. These were

first noticed during the implementation of *select best combination* and *select best sequence*, where we need to undo all the changes after each sequence has been applied, to get the program back to the initial state to apply the next sequence.

To make the undoing of transformations more stable and maintainable, it was decided to implement it differently. This is now accomplished as follows: when the compiler gets a request to undo some changes, it dumps the transformations applied so far to a file (this was also done in the earlier version of VISTA to enable compilation of a file over multiple sessions). Then the current function along with all its data structures are discarded and reinitialized. The same function is read back in and the transformations are reapplied, but only up to the point we want. Thus the remaining changes are automatically discarded. This is the the same thing that would have happened when the file is compiled over multiple sessions, but now there is no need to store all the state information at each point. Doing this we were able to remove a lot of redundant code and data structures which ultimately helped in making the code more robust. The only drawback with this scheme is that, when the user only wants to undo one change, the compiler needs to go through the entire process of reloading the current function and reapplying most of the transformations again. This is arguably slower than the previous approach, but the increase in compilation time was found to be a small cost to pay for the greatly increased maintainability and smoothness achieved by the new approach.

7.2 Required Analysis by Each Optimization Phase

Each optimization phase needs some data and control flow analysis to be done before it can be successfully applied. Absence of the correct analysis can prevent the compiler from recognizing all the points where the optimization can be usefully applied. Even worse is the case when incorrect analysis leads the compiler into making changes at points that produces wrong output code. Figuring out all the analysis required by each optimization phase is difficult. Also, in some cases when the required analysis has already been done and is valid, there is no need to spend more time redoing the analysis. So we also need to determine

which analysis are invalidated by each phase. An example of this is *instruction selection* invalidating *live variable analysis* and *registers used analysis*. This important work was painfully attempted during the previous version of VISTA. Some bugs in that process were revealed during the implementation of the current version, especially when thousands of different sequences were attempted on the same function during a single run of *select best combination*. Some changes were made to correct the faults.

7.3 Sanity Check

In VISTA the program representation information for each function is maintained at two places, one with the compiler and a copy of it with the viewer to be displayed to the user. It is essential that these two versions be consistent at each distinct point during the compilation process. Whenever the compiler makes a change to the program representation, it sends a message to the viewer to do the same to its version of the function. We felt it was useful to have an option to check if the two representations are consistent. This would be very helpful whenever a change is made to the compiler or a new optimization is implemented in the compiler. *Sanity check* is the option we provide to accomplish this, during which the compiler essentially just sends its complete state information to the viewer and the viewer cross-checks it with the information it holds in its data structures. Discrepancies, if any, are reported to the user.

7.4 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 select-able 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 non-trivial task requiring meticulous and careful handling.

7.5 Batch Experiments

VISTA is an interactive system, which is controlled by the user sending requests to the compiler by clicking buttons. But this was found to be a hindrance while performing experiments or extensive testing, like that done during regression testing. It was not reasonable to expect the user to sit at his desk and do the testing manually by clicking buttons every so often. To make testing easier, we support a new mode in VISTA where selections are read from a file instead of requiring mouse clicks. Such a file could either be written by hand or could also be generated automatically by the viewer at run-time. VISTA provides an option to the user to store all the mouse clicks in a separate file. This file can be read at any later time and mouse clicks are not required. This was found to be very useful during testing as we could construct test files that could be initiated in a batch mode whenever any major changes were made to the compiler. It was also found to be helpful for conducting experiments as the ones described in chapter 5.

7.6 Obtaining Measurements on a Host Machine

As mentioned earlier, to get the dynamic instruction counts the compiler needs to produce assembly code, instrument the code with additional instructions to collect measures and then link and execute the program. Currently, the compiler produces SPARC assembly.

Consequently, it was not possible to execute the code and get measures on a non-SPARC architecture. It was observed that we can still get the static instruction counts (code size) on a non-SPARC architecture. A small modification made to the compiler ensured that it can now detect if the architecture is SPARC and if not then it only gets the code size measures. This allows limited demonstration on a laptop.

Chapter 8

Related Work

Other researchers have developed systems that provide interactive compilation support. These systems include the *pat* toolkit [13], the *parafrase-2* environment [14], the *e/sp* system [15], a visualization system developed at the University of Pittsburgh [16] , and SUIF explorer [17]. 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 supports low-level transformations and user-specified changes, which are needed for tuning embedded applications.

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 [18]. 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) [19]. 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 [20] or to eliminate branches [21]. 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 [22, 23]. A system using genetic algorithms to better parallelize loop nests has been developed and evaluated [24]. 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 [25]. A low-level compilation system developed at Rice University uses a genetic algorithm to reduce code size by finding efficient optimization phase sequences [11]. However, this system is batch oriented instead of interactive, concentrated 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.

Chapter 9

Future Work

There is much future work to consider on the topic of selecting effective optimization sequences. It would be informative to obtain measurements on a real embedded systems architecture. However, most of these systems only provide execution time measurements via simulation on a host processor. The actual embedded processor may often not be available or downloading the executable onto the embedded machine and obtaining measurements may not be easily automated. 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. One option is to translate the assembly produced for the embedded machine to an equivalent assembly program on a host processor. This assembly can be instrumented in order to produce a dynamic instruction count of each basic block when executed. An estimation of the number of CPU cycles for each basic block can be multiplied by the count to give a responsive and reasonably accurate measure of dynamic performance on an embedded processor that does not have a memory hierarchy.

Another area of future work is to vary the characteristics of the experiments. We only obtained measurements for 100 generations and a optimization sequence that is 1.25 times the length of the successfully applied batch optimization sequence. It would be interesting to see how performance improves as the number of generations and the sequence length

increases. The actual crossover and mutation operations could also be varied. In addition, the set of candidate optimization phases could be extended. Finally, the set of benchmarks evaluated could be increased.

All of the experiments in our study involved selecting optimization phase sequences for entire functions. We 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.

Chapter 10

Conclusions

There are several contributions that we have presented in this thesis. First, we have developed an interactive compilation system that automatically provides performance feedback information to a user after each successfully applied optimization phase. This feedback allows a user to gauge the progress when tuning an application. Second, we allow a user to interactively select structured constructs for applying optimization phase sequences. These constructs allow the conditional or iterative application of optimization phases. In effect, we have provided an optimization phase programming language. Third, we have provided constructs that automatically select optimization phase sequences based on the specified fitness criteria. A user can enter specific sequences and the compiler will choose the sequence that produces the best result. A user can also specify a set of optimization 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. 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. The VISTA system supports tuning of applications by providing the ability to supply performance feedback information, select optimization phases, and automatically search for efficient sequences of optimization phases. Embedded programmers often resort to coding in assembly to meet stringent constraints on time, size, and power consumption. Besides using VISTA to obtain a more efficient executable, such an environment may encourage more users to develop applications in a high level language, which can result in software that is more portable, more robust, and less costly to develop and maintain.

Appendix A

Overview of Genetic Algorithms

A.1 Introduction

Genetic Algorithms (GAs) are a family of computational models inspired by evolution. These were invented by John Holland in the 1960's and were developed by Holland and his students and colleagues at the University of Michigan in the 1960's and 1970's. In contrast with evolution strategies and evolutionary programming, Holland's original goal was not to design algorithms to solve specific problems, but rather to formally study the phenomenon of adaptation as it occurs in nature and to develop ways in which the mechanisms of natural adaptation might be imported into computer systems. Holland's 1975 book *Adaptation in Natural and Artificial Systems* presented the genetic algorithm as an abstraction of biological evolution and gave a theoretical framework for adaptation under the GA. Holland's GA is a method for moving from one population of *chromosomes* (e.g., strings of ones and zeros, or "bits") to a new population by using a kind of *natural selection* together with the genetics-inspired operators of crossover, mutation, and inversion.

In the 1950s and the 1960s several computer scientists independently studied evolutionary systems with the idea that evolution could be used as an optimization tool for engineering problems. The idea in all these systems was to evolve a population of candidate solutions to a given problem, using operators inspired by natural genetic variation and

natural selection. Holland's introduction of a population-based algorithm with crossover, inversion, and mutation was a major innovation. Moreover, Holland was the first to attempt to put computational evolution on a firm theoretical footing. Until recently this theoretical foundation, based on the notion of *schemas*, was the basis of almost all subsequent theoretical work on genetic algorithms.

The mechanisms of *evolution* and *adaptation* seem very well suited for some of the most pressing computation problems in many fields. Many computational problems require searching through a huge number of possibilities for solutions. One example is the problem of computational protein engineering, in which an algorithm is sought that will search among the vast number of possible amino acid sequences for a protein with specified properties. What is needed in such cases is both computational parallelism and an intelligent strategy for choosing the next set of sequences to evaluate. Many computational problems also require a computer program to be adaptive - to continue to perform well in a changing environment. This can be seen in some computer interfaces which need to adapt to the idiosyncrasies of different users. Biological evolution is an appealing source of inspiration for addressing these problems. Evolution is, in effect, a method of searching among an enormous number of possibilities for *solutions*. In biology the enormous set of possibilities is the set of possible genetic sequences, and the desired solutions are highly fit organisms - organisms well able to survive and reproduce in their environments. Evolution can also be seen as a method for designing innovative solutions to complex problems. For example, the mammalian immune system is a marvelous evolved solution to the problem of germs invading the body. Seen in this light, the mechanisms of evolution can inspire computational search methods.

A.2 Biological Terminology

All living organisms consist of cells, and each cell contains the same set of one or more *chromosomes* (strings of DNA) that serve as a blueprint for the organism. A chromosome can be conceptually divided into *genes* (functional blocks of DNA), each of which encodes

a particular protein. Very roughly, one can think of a gene as encoding a trait, such as eye color. The different possible settings for a trait (e.g., blue, brown, hazel) are called *alleles*. Each gene is located at a particular *locus* (position) on the chromosome.

Many organisms have multiple chromosomes in each cell. The complete collection of genetic material (all chromosomes taken together) is called the organism's *genome*. The term *genotype* refers to the particular set of genes contained in a genome. Two individuals that have identical genomes are said to have the same genotype. The genotype gives rise, under fetal and later development, to the organism's *phenotype* - its physical and mental characteristics, such as eye color, height, brain size, and intelligence.

During sexual reproduction, *recombination* (or *crossover*) occurs: in each parent, genes are exchanged between each pair of chromosomes to form a *gamete* (a single chromosome), and then gametes from the two parents pair up to create a full set of diploid chromosomes. Offspring are subject to *mutation*, in which single nucleotides (elementary bits of DNA) are changed from parent to offspring, the changes often resulting from copying errors. The *fitness* of an organism is typically defined as the probability that the organism will live to reproduce (*viability*) or as a function of the number of offspring the organism has (*fertility*).

In genetic algorithms, the term chromosome typically refers to a candidate solution to a problem, often encoded as a bit string. The genes are either single bits or short blocks of adjacent bits that encode a particular element of the candidate solution (e.g., in the context of multi-parameter function optimization the bits encoding a particular parameter might be considered to be a gene). An allele in a bit string is either 0 or 1; for larger alphabets more alleles are possible at each locus. Crossover typically consists of exchanging genetic material between two single-chromosome parents. Mutation consists of flipping the bit at a randomly chosen locus (or, for larger alphabets, replacing a the symbol at a randomly chosen locus with a randomly chosen new symbol). The genotype of an individual in a GA using bit strings is simply the configuration of bits in that individual's chromosome.

A.3 A Simple Genetic Algorithm

Given a clearly defined problem to be solved and a bit string representation for candidate solutions, a simple GA works as follows:

1. Start with a randomly generated population of n l-bit chromosomes (candidate solutions to a problem).

2. Calculate the fitness $f(x)$ of each chromosome x in the population.

3. Repeat the following steps until n offspring have been created:

   (a) Select a pair of parent chromosomes from the current population, the probability of selection being an increasing function of fitness. Selection is done *with replacement* meaning that the same chromosome can be selected more than once to become a parent.

   (b) With probability P_c (the "crossover probability" or "crossover rate"), cross over the pair at a randomly chosen point (chosen with uniform probability) to form two offspring. If no crossover takes place, form two offspring that are exact copies of their respective parents. (Note that here the crossover rate is defined to be the probability that two parents will cross over in a single point. There are also "multi-point crossover" versions of the GA in which the crossover rate for a pair of parents is the number of points at which a crossover takes place).

   (c) Mutate the two offsprings at each locus with probability p_m (the mutation probability or mutation rate), and place the resulting chromosomes in the new population.

4. Replace the current population with the new population.

5. Go to step 2.

Each iteration of this process is called a *generation*. A GA is typically iterated for anywhere from 50 to 500 or more generations. The entire set of generations is called

a *run*. At the end of a run there are often one or more highly fit chromosomes in the population. Since randomness plays a large role in each run, two runs with different random-number seeds will generally produce different detailed behaviors. GA researchers often report statistics (such as the best fitness found in a run and the generation at which the individual with that best fitness was discovered) averaged over many different runs of the GA on the same problem.

A.4 Some Applications of Genetic Algorithms

The version of the genetic algorithm described above is very simple, but variations on the basic theme have been used in a large number of scientific and engineering problems and models. Some common applications are:

Optimization: GAs have been used in a wide variety of optimization tasks, including numerical optimization and such combinatorial optimization problems as circuit layout and job-shop scheduling.

Automatic programming: GAs have been used to evolve computer programs for specific tasks, and to design other computational structures such as cellular automata and sorting networks.

Machine learning: GAs have been used for many machine learning applications, including classification and prediction tasks, such as the prediction of weather or protein structure. GAs have also been used to evolve aspects of particular machine learning systems, such as weights for neural networks, rules for learning classifier systems or symbolic production systems, and sensors for robots.

Economics: GAs have been used to model processes of innovation, the development of bidding strategies, and the emergence of economic markets.

Immune systems: GAs have been used to model various aspects of natural immune systems, including somatic mutation during an individual's lifetime and the discovery of

multi-gene families during evolutionary time.

Ecology: GAs have been used to model ecological phenomena such as biological arms races, host-parasite coevolution, symbiosis, and resource flow.

Population genetics: GAs have been used to study questions in population genetics, such as "Under what conditions will a gene for recombination be evolutionarily viable?"

Evolution and learning: GAs have been used to study how individual learning and species evolution affect one another.

Social systems: GAs have been used to study evolutionary aspects of social systems, such as the evolution of social behavior in insect colonies, and, more generally, the evolution of cooperation and communication in multi-agent systems.

Appendix B

An Implementation of a Genetic Algorithm

We have implemented a version of the genetic algorithm for use during the *select best combination* construct in VISTA. The algorithm employed is listed in this chapter.

When the *biased sampling search* option is selected during *select best combination*, VISTA also allows the user to specify the parameters for the genetic algorithm, namely the number of chromosomes (population size) and the number of generations. Let the user specify numbers m and n for the population size and the number of generations respectively. The algorithm then works as follows:

1. Create an initial population of m chromosomes (optimization sequences) by randomly choosing optimizations in each sequence.

2. Compute a fitness value for each chromosome. To do this, the optimization sequence defined by the chromosome is applied to the function being compiled. The resulting code is instrumented with instructions to calculate the static and dynamic counts. This instrumented code is executed and the static and dynamic counts are collected. The final fitness value depends on the relative weights assigned by the user to the static and dynamic counts respectively.

3. The chromosomes are sorted by fitness values from lowest to highest. The population is split into a lower and upper half, based on fitness value, each half consisting of $m/2$ chromosomes. The upper half consists of $m/2$ chromosomes with the lowest fitness value. (Note that for our case the lower the fitness value the better, since we are interested in lowering both the static code size and dynamic instruction count.)

4. The chromosome with the highest fitness value (worst performance) is removed from the population. $m/5-1$ additional chromosomes are chosen at random from the lower half of the population and removed.

5. To fill the vacancies in the population, new chromosomes are generated using the crossover operation. Two parent chromosomes are randomly chosen from the upper half of the population. The first half of one chromosome is concatenated with the second half of the other chromosome and vice versa, creating two new chromosomes. This operations is performed as many times as required to fill all the vacancies.

6. The $m - m/5$ chromosomes not altered during the previous step are subjected to mutation. The best performing chromosome is also exempted from mutation. For each such chromosome, each gene is considered. For a chromosome in the lower half of the population, mutation occurs with a probability of 0.1 (or 10 percent). For a chromosome in the upper half of the population, the probability of mutation is reduced to 0.05 (or 5 percent). To mutate a gene, it is replaced with a randomly selected gene.

This process is repeated for n generations, and we keep track of the best chromosome found over the course of the run.

Appendix C

Protocols of Compiler-Viewer Messages

Below is a definition of the protocols of the messages transferred between the viewer and the compiler. Words in upper case denote constants used in either the compiler or the viewer.

1. Messages sent when VISTA is started

```
          Viewer                                  Compiler


                                          <  USER_INTERACT
                                          <  BEGINFUNCTION {function name}
   SANITY_CHECK / ABORT_VPO        >
   TRUE / FALSE                    >
                                          <  {base name of source file}
                                          <  {send basic blocks}
                                          <  ENDINITSET
                                          <  [    {initial transformations
                                                   from the .trans file}
                                                 ENDSEQ                    ]
```

```
                                    <  [    BEGINFREQ
                                           {send frequency measures}
                                           ENDFREQ                    ]
```

2. Proceed to the next function (when next function exists)

```
NEXT_FUNC                      >
QUIT_TRANS                     >
                                    <  ENDFUNCTION
                                    <  BEGINFUNCTION {function name}
SANITY_CHECK / ABORT_VPO       >
TRUE / FALSE                   >
                                    <  {base name of source file}
                                    <  {send basic blocks}
                                    <  ENDINITSET
                                    <  [    {initial transformations
                                             from the .trans file}
                                           ENDSEQ                     ]
                                    <  [    BEGINFREQ
                                           {send frequency measures}
                                           ENDFREQ                    ]
```

3. Proceed to the next function (next function does not exist)

```
NEXT_FUNC                      >
QUIT_TRANS                     >
                                    <  ENDVIEW
STOP_TRANS                     >
QUIT_TRANS                     >
                                    <  {compiler exits}
```

4. Discard changes after current state

```
UNDO_TRANS {number of trans}   >
QUIT_TRANS                     >
                                    <  [    BEGINFREQ
                                           {send frequency measures}
                                           ENDFREQ                    ]
```

5. Show loop information

```
LOOPS_QUERY                    >
QUIT_TRANS                     >
                                    <  {loops information}
                                    <  ENDSEQ
```

6. Sanity Check

```
SANITY_CHECK_REQUEST           >
QUIT_TRANS                     >
                                    <  {basic block information}
                                    <  ENDSEQ
```

7. Get Frequency Information

```
FREQUENCY_REQ                  >
QUIT_TRANS                     >
                                    <  [    CHANGE_TEST_CONFIG_ID
                                           {send initial values in file}
                                           ENDSEQ
      {send new config values}     >
      QUIT_TRANS                                                      ]
                                    <  BEGINFREQ
```

```
                                        <  {send frequency measures}
                                        <  ENDFREQ
```

8. Start measurements

```
START_MEASUREMENT              >
QUIT_TRANS                     >
                                        <  [    CHANGE_TEST_CONFIG_ID
                                                {send initial values in file}
                                                ENDSEQ
      {send new config values}  >
      QUIT_TRANS                                                        ]
                                        <  BEGINFREQ
                                        <  {send frequency measures and set flag}
                                        <  ENDFREQ
```

9. Create / Modify test configuration file

```
CHANGE_CONFIG_FILE             >
QUIT_TRANS                     >
                                        <  CHANGE_TEST_CONFIG_ID
                                        <  {send initial values in file}
                                        <  ENDSEQ
{send new config values}       >
QUIT_TRANS                     >
```

10. Specifying optimization phase sequences

```
{list of selected blocks}      >
{list of optimization phases}  >
QUIT_TRANS                     >
```

```
                                    <  {sequence of changes}
                                    <  ENDSEQ
```

11. Exit

```
STOP_TRANS                  >
QUIT_TRANS                  >
```

Bibliography

[1] Steven R. Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. In *Proceedings of the fifteenth annual workshop on microprogramming on Microprogramming*, pages 125–133, 1982.

[2] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):1053–1084, 1997.

[3] Keith Clarke Simon Segars and Liam Goudge. Embedded control problems, thumb, and the arm7tdmi. *IEEE Micro*, 15(5):22–30, October 1995.

[4] Wankang Zhao, Baosheng Cai, David Whalley, Mark W. Bailey, Robert van Engelen, Xin Yuan, Jason D. Hiser, Jack W. Davidson, Kyle Gallivan, and Douglas L. Jones. Vista: a system for interactive code improvement. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 155–164. ACM Press, 2002.

[5] Manuel E. Benitez and Jack W. Davidson. Target-specific global code improvement: Principles and applications. Technical Report CS-94-42, 4, 1994.

[6] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 conference on Programming Language design and Implementation*, pages 329–338. ACM Press, 1988.

[7] Jack W. Davidson and David B. Whalley. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems*, 15(9):459–472, November 1991.

[8] Rainer Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, Boston, 1997.

[9] Haris Lekatsas and Wayne Wolf. Code compression for embedded systems. In *Design Automation Conference*, pages 516–521, 1998.

[10] Dan Ernst Todd M. Austin Trevor Mudge Matthew R. Guthaus, Jeffrey S. Ringenberg and Richard B. Brown. Mibench: A free, commercially representative embedded

benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[11] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 1–9. ACM Press, 1999.

[12] Baosheng Cai. Compiler modifications to support interactive compilation. Master's thesis, Florida State University, Tallahassee, Florida, 2001.

[13] Kevin Smith Bill Appelbe and Charlie McDowell. Start/pat: a parallel- programming toolkit. In *IEEE Software*, volume 6 of *4*, pages 29–40, 1989.

[14] M. Haghighat C. Lee B. Leung C. Polychronopoulos, M. Girkar and D. Schouten. Parafrase–2: An environment for parallelizing, partitioning, and scheduling programs on multiprocessors. In *International Journal of High Speed Computing*, volume 1 of *1*, pages 39–48, Pensylvania State University Press, August 1989.

[15] J. Kiall C. Denton J. Browne, K. Sridharan and W. Eventoff. Parallel structuring of real-time simulation programs. In *COMPCON Spring '90: Thirty-Fifth IEEE Computer Society International Conference*, pages 580–584, February 1990.

[16] Chyi-Ren Dow, Shi-Kuo Chang, and Mary Lou Soffa. A visualization system for parallelizing programs. In *Supercomputing*, pages 194–203, 1992.

[17] Shih-Wei Liao, Amer Diwan, Jr. Robert P. Bosch, Anwar Ghuloum, and Monica S. Lam. Suif explorer: an interactive and interprocedural parallelizer. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 37–48. ACM Press, 1999.

[18] S. Novack and A. Nicolau. Vista: The visual interface for scheduling transformations and analysis. In *Languages and Compilers for Parallel Computing*, pages 449–460, 1993.

[19] Brian Harvey and Gary Tyson. 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, 1996.

[20] Henry Massalin. Superoptimizer: a look at the smallest program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating systems*, pages 122–126, October 1987.

[21] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the gnu c compiler. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 341–352, June 1992.

[22] Toru Kisuki, Peter M. W. Knijnenburg, and Michael F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *IEEE PACT*, pages 237–248, 2000.

[23] A. Petitet Whaley, R. and J. Dongarra. utomated empirical optimization of software and the atlas project. In *Parallel Computing*, volume 27 of *1-2*, pages 3–25, 2001.

[24] A. Nisbet. Genetic algorithm optimized parallelization. In *Workshop on Profile and Feedback Directed Compilation*, 1998.

[25] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *Workshop on Feedback-Directed Optimization*, November 1999.

[26] Melanie Mitchell. *An Introduction to Genetic Algorithms.* Cambridge, Mass. MIT Press, 1996.

Biographical Sketch

Prasad Kulkarni was born on October 13, 1979 in Thane, India. He recieved his Bachelor of Computer Engineering degree from Pune University, India in 2001. In 2003, he graduated from Florida State University with a Master of Science Degree in Computer Science. He is currently seeking his Ph.D. in Computer Science at Florida State University. His areas of interest include computer architecture, compilers, embedded systems and real-time systems.