# Finding Effective Optimization Phase Sequences

Prasad Kulkarni[1], Wankang Zhao[1], Hwashin Moon[2], Kyunghwan Cho[2], David Whalley[1], Jack Davidson[3], Mark Bailey[4], Yunheung Paek[5], Kyle Gallivan[1]

[1]Computer Science Department, Florida State University, Tallahassee, FL 32306-4530; e-mail: whalley@cs.fsu.edu

[2]Electrical Engr Dept, Korea Advanced Institute of Science & Technology, Daejon 305-701, Korea

[3]Computer Science Department, University of Virginia, Charlottesville, VA 22904; e-mail: jwd@virginia.edu

[4]Computer Science Department, Hamilton College, Clinton, NY 13323; e-mail: mbailey@hamilton.edu

[5]School of Electrical Engineering, Seoul National University, Seoul 151-742, Korea; e-mail: ypaek@ee.snu.ac.kr

## ABSTRACT

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. In this paper, we describe support in VISTA, an interactive compilation system, for finding effective sequences of optimization phases. VISTA 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 specified fitness criteria. We have included a number of experimental results that evaluate the effectiveness of using a genetic algorithm in VISTA to find effective optimization phase sequences.

### Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – compilers, optimization D.4.7 [**Operating Systems**]: Organization and Design – real-time systems and embedded systems.

### General Terms

Measurement, Performance, Experimentation, Algorithms.

### Keywords

Phase ordering, interactive compilation, genetic algorithms.



*LCTES'03*, June 11-13, 2003, San Diego, California, USA.



## 1. INTRODUCTION

The phase ordering problem has long been known to be a difficult dilemma for compiler writers [21, 23]. A single sequence of optimization phases is highly unlikely to produce optimal code for every application (or even each function within an application) on a given machine. 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 [23].

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) [20]. 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.

An embedded application may reside in a product for which millions of units may be shipped (e.g. cellular phones, digital cameras, printers, etc.). For this reason an embedded application developer may be willing to expend considerable effort and time to produce an application that is faster, smaller, or consumes less power. However, it is clear that attempting all optimization phase orderings will be prohibitively expensive since many different optimizations are available and can be repeatedly applied. For this reason, we believe a viable approach is to let the embedded application developer steer the optimization process. In this paper, we describe support in the VISTA (VPO Interactive System for Tuning Applications) compilation system for finding sequences of optimization phases that meet the developer's goals.

## 2. RELATED WORK

Other researchers have developed systems that provide interactive compilation support. These systems include the *pat* toolkit [1], the *parafrase-2* environment [19], the *e/sp* system [4], a visualization system developed at the University of Pittsburgh [9], and SUIF explorer [15]. These systems provide support by illustrating the possible dependencies that may prevent parallelizing transformations. A user can 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 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) [12]. 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 [16] or to eliminate branches [10]. 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 [14, 22]. A system using genetic algorithms to better parallelize loop nests has been developed and evaluated [17]. 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 [5]. A low-level compilation system developed at Rice University uses a genetic algorithm to reduce code size by finding efficient optimization phase sequences [6]. However, this system is batch oriented instead of interactive and is designed to use the same optimization phase order for all of the functions within a file.

## 3. THE VISTA FRAMEWORK

In this section, we review the VISTA framework. Figure 1 illustrates the flow of information in VISTA, which consists of a compiler and a viewer. The programmer initially indicates a file to be compiled and then specifies requests through the viewer, which include sequences of optimization phases, user-defined transformations, and queries. The compiler performs the specified actions and sends program representation information back to the viewer. When the user chooses to terminate the session, VISTA writes the sequence of transformations to a file so they can be reapplied at a later time, enabling future updates to the program representation.

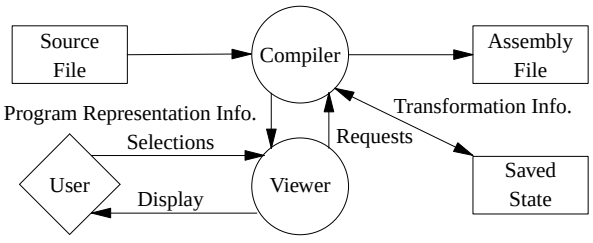


**Figure 1: Interactive Code Improvement Process**

The compiler used in VISTA is based on VPO (Very Portable Optimizer) [2, 3]. VPO is a compiler back end that performs all of its optimizations on a single low-level representation called RTLs (register transfer lists). Because VPO uses a single representation, it can apply most analyses and optimization phases repeatedly and in an arbitrary order. This feature facilitates finding effective sequences of optimization phases.

The VISTA framework supports the following features. First, it allows a user to view a low-level graphical representation of the function being compiled, which is much more convenient than extracting this information from a source-level debugger. Second, a user can select the order and scope of optimization phases. Selecting the order of optimization phases may allow a user to find a sequence of phases that are more effective for a specific function than the default optimization phase order. Limiting the scope of the optimization phases allows a user to concentrate resources, such as registers, on the critical regions of the code. Third, a user can manually specify transformations. This feature is useful when exploiting specialized architectural features that cannot be exploited automatically by the compiler. Fourth, a user can undo previously applied transformations or optimization phases. This feature eases experimentation with alternative phase orderings or user-defined transformations.

We have made several enhancements to VISTA that facilitates the selection of effective sequences of optimization phases. These enhancements include automatically obtaining performance feedback information, using structured statements for applying optimization phases, automatically using performance information for selecting optimization phase sequences. We describe these enhancements in the following three sections of the paper.

## 4. OBTAINING INTERACTIVE PERFORMANCE INFORMATION

VISTA supports obtaining both static and dynamic measurements. For our current experiments, we provide a rough approximation of the number of CPU cycles by counting the number of instructions executed. The same approach was used in a prior study that searches for effective optimization phase sequences using a genetic algorithm [6]. We measured the dynamic instruction count by instrumenting the assembly code with instructions to increment counters using the EASE (Environment for Architecture Study and Experimentation) system [7, 8]. For embedded applications, obtaining actual execution times is problematic due to cross compilation and this issue is described in more detail later in the paper.

Figure 2 shows a window that appears the first time a user obtains performance measurements for a program. Whenever measurements are needed, an instrumented executable for the program is produced and executed. The user is prompted for the information that is needed to link, execute, and verify correct program behavior. A user could accomplish this with a previous version of VISTA in a series of manual steps [24]. This process is automatically performed after applying each optimization phase in the latest version of VISTA.

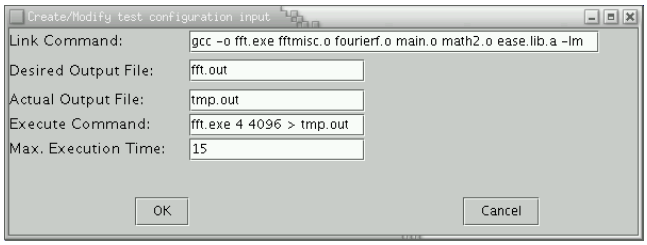


**Figure 2: Test Configuration Window**

After applying any remaining necessary optimization phases, VISTA produces instrumented assembly code for the function. As described in subsequent sections, we need to obtain performance measurements many times during the compilation of a single function. We compile functions within a file one at a time and each file is compiled separately. In order to reduce compilation overhead, we save the position in the assembly file at the beginning of the function and regenerate the assembly at that point. The assembly for the remaining functions in the file is generated by reading in the transformation information. Thus, obtaining new measurements requires producing instrumented assembly for only the remaining functions within the file and assembling only the current file. A link and execution step is also required each time measurements are obtained. We also save the position in the intermediate code file that is input to VPO and the position in the transformation file to further reduce I/O. After executing the program VISTA reads the initial program representation and the compiler reapplies the sequence of transformations to reproduce the program representation at the point where measurements were taken. In addition, VISTA reads the frequency counts for each basic block produced by the instrumented executable.

Figure 3 shows a snapshot of the viewer with the history of a 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. This information allows a user to quickly gauge the progress that has been made in improving the function. The frequency of each basic block relative to the function is also shown in each block header line, which allows a user to identify the critical regions of a function.

## 5. INTERACTIVELY SELECTING OPTIMIZATION PHASE SEQUENCES

A programmer has little or no control over the order in which code improvement phases are applied when using most compilers. VISTA provides the programmer with the flexibility to specify the exact sequence of optimization phases to be applied in a specified region of code. In order to support this flexibility, we identified the set of analyses required for each optimization phase and the set of analyses invalidated by each optimization phase. In addition, we identified the constraints among existing optimization phases. For instance, most optimization phases can only be applied before filling branch delay slots.

We find it useful to conditionally invoke an optimization phase based on whether a previous optimization phase caused any changes to the program representation. The application of one optimization phase often provides opportunities for another optimization phase. Such a feature allows a sequence of optimization phases to be applied until no further improvements are found. Likewise, an optimization phase that is unlikely to result in code-improving transformations unless a prior phase has changed the program representation can be invoked only if changes occurred, which may save compilation time.

Prior support in VISTA for conditionally applying optimization phases was only a low level branch operation (i.e., `if changes goto` <*label*>) [24]. VISTA now supports testing if changes to the program representation have occurred in the form of four structured control statements (*if-changes-then*, *if-changes-then-else*, *do-while-changes*, *while-changes-do*) a user can specify interactively. These structured statements, which can be nested, are provided to make selection of sequences of optimization phases more convenient to the user. In effect, we are providing an optimization phase programming language.

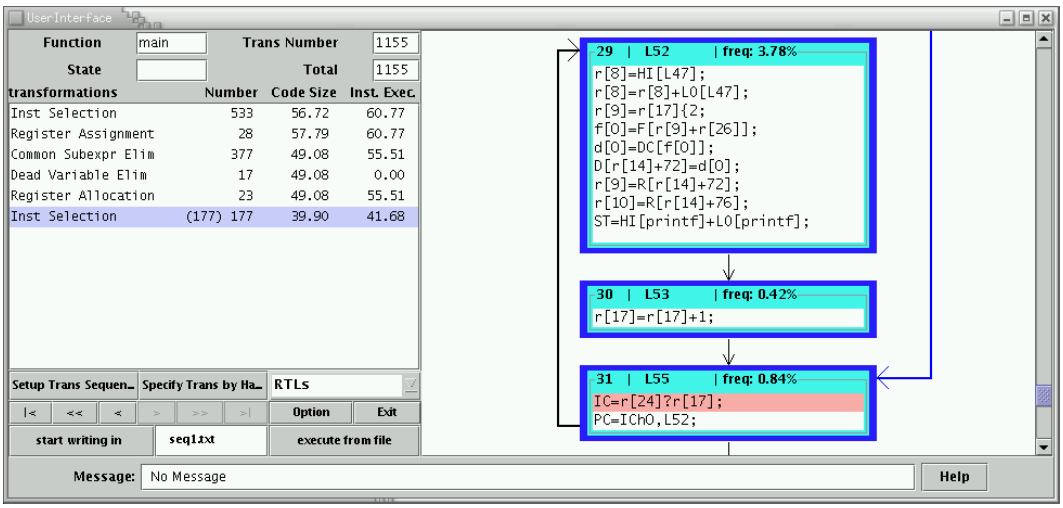


**Figure 3: Main Window of VISTA Showing History of Optimization Phases**

Consider the interaction between register allocation and instruction selection optimization phases. Register allocation replaces load and store instructions with register-to-register move instructions, which provides opportunities for instruction selection. Instruction selection combines instructions together and reduces register pressure, which may allow additional opportunities for register allocation. Figure 4 illustrates how to exploit the interaction between these two phases with a simple example. The user has selected two constructs, which are a *do-while-changes* statement and a *if-changes-then* statement. For each iteration, the compiler performs register allocation. Instruction selection is only performed if register allocation allocates one or more live ranges of a variable to a register. These phases will be iteratively applied until no additional live ranges are allocated to registers.

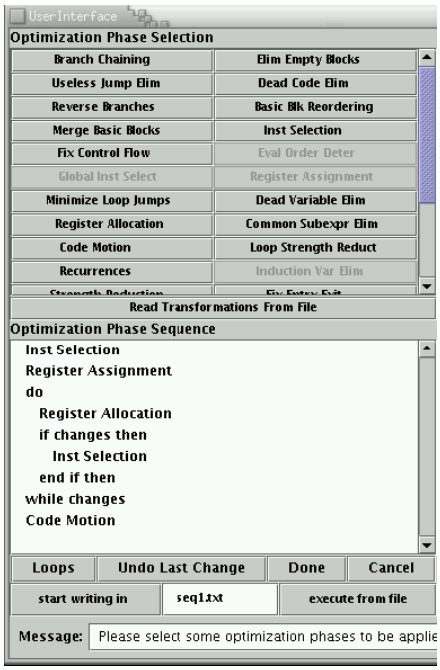


**Figure 4: Interactively Selecting Optimization Phases**

In order to communicate to VPO the sequence of optimizations phases to apply, the viewer translates the structured statements into a low-level sequence of requests. This sequence is interpreted by VPO and each resulting change to the program representation is sent to the viewer. This process continues until a stop operation has been encountered. Figure 5 reflects the operations to be performed by the selections shown in Figure 4.

1. Perform instruction selection
2. Perform register assignment
3. Enter loop
4. Perform register allocation
5. If no changes in last 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
10. Stop

**Figure 5: Operations Performed by Selections in Figure 4**

# 6. PERFORMANCE DRIVEN SELECTION OF OPTIMIZATION SEQUENCES

In addition to allowing a user to specify an optimization sequence, it is desirable for the compiler to automatically compare two or more sequences and determine which is most beneficial. VISTA provides two structured constructs that support automatic selection of optimization sequences. The first construct is the *select-best-from* statement and is illustrated in Figure 6. This statement evaluates two or more specified sequences of optimization phases and selects the sequence that best improves performance according to the selected criteria. For each sequence the compiler applies the specified optimization phases, determines the program performance (instruments the code for obtaining performance measurements, produces the assembly, executes the program, and obtains the measurements), and reapplies the transformations to reestablish the program representation at the point where the *select-best-from* statement was specified. After the best sequence is found, the compiler reapplies that sequence.

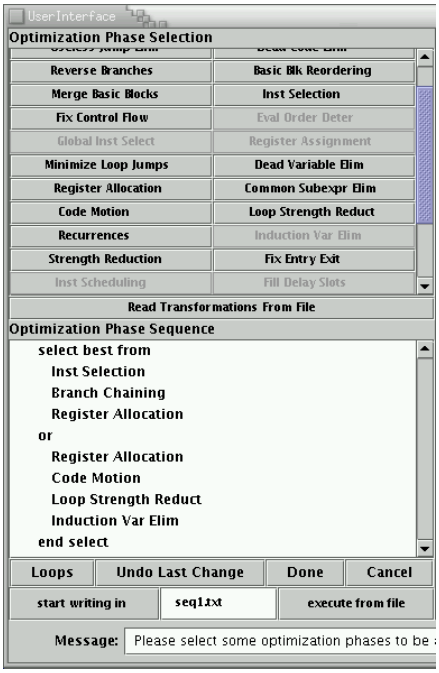


**Figure 6: Selecting the Best Sequence from a Specified Set**

The other construct, the *select-best-combination* statement, is depicted in Figure 7. This statement accepts a set of *m* distinct optimization phases and attempts to discover the best sequence for applying these phases. Figure 8 shows the different options that we provide the user to control the search. The user specifies the *sequence length*, *n*, which is the total number of phases applied in each sequence. An *exhaustive search* attempts all possible $m^n$ sequences, which may be appropriate when the total number of possible sequences can be evaluated in a reasonable period of time. The *biased sampling search* applies a genetic algorithm in an attempt to find the most effective sequence within a limited amount of time since in many cases the search space is too large to evaluate all possible sequences.

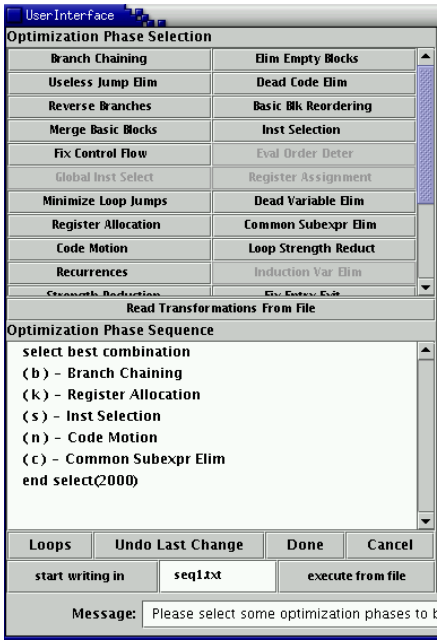


**Figure 7: Selecting the Best Sequence from a Set of Optimization Phases**

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

Performing these searches is often time-consuming. Thus, VISTA provides a window showing the current status of the search. Figure 9 shows a snapshot of the status of the search that was selected in Figures 7 and 8. The percentage of sequences completed along with the best sequence and its effect on performance is given. The user can terminate the search at any point and accept the best sequence found so far.

## 7. EXPERIMENTS

This section describes the results of a set of experiments to illustrate the effectiveness of VISTA's biased sampling search, which uses a genetic algorithm to find efficient sequences of optimization phases. We used a subset of the *mibench* benchmarks, which are C applications targeting specific areas of the embedded market [11]. We used one benchmark from each of the six categories of applications. Table 1 contains descriptions of these programs.

| 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 fast fourier transform |
| consumer | jpeg | image compression and decompression |
| security | sha | secure hash algorithm |
| office | stringsearch | searches for given words in phrases |

**Table 1: MiBench Benchmarks Used in the Experiments**

Our target architecture for these experiments is 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 [6]. 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 ours and we will contrast the two studies throughout this section.

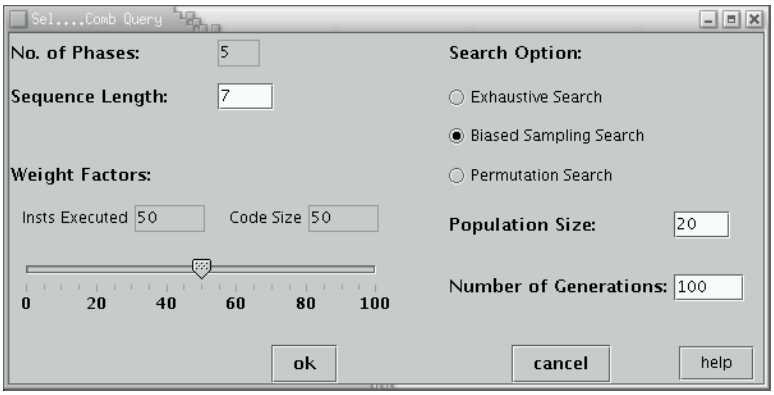


**Figure 8: Selecting Options to Search the Space of Possible Sequences**

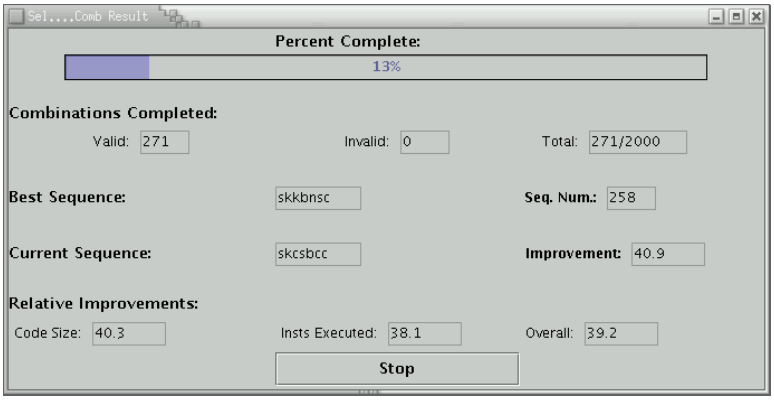


**Figure 9: Window Showing the Status of Searching for an Effective Sequence**

The Rice study 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 are obtained. Figure 10 shows the algorithm used to determine the order in which optimization phases are applied in VPO. This algorithm has evolved over the years and the primary goal is 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 *fix 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 10 must be applied after *register assignment* and before *fix entry exit*. Thus, we decided to use the genetic algorithm to find the best sequence of code-improving phases that can be applied between these two required phases. These candidate sequences include fourteen unique phases that can be applied in any order between the two required phases. Table 2 describes each of these fourteen phases and gives a designation (gene) for each phase that will be used later when representing each optimization in a sequence.

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). An appropriate uniform limit is not easily determined since the number of optimization phases attempted by the batch compiler can 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 in the batch compilation before establishing the sequence length for our searches.

```
...
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
...
```

**Figure 10: VPO's Order of Optimization Phases Applied in *Batch* Mode**

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 produced for the application after batch optimization. In four of the benchmarks, some functions were not executed even though we used the

| Optimization Phase | Gene | Description |
|---|---|---|
| instruction selection | s | Combine instructions together when the combined effect is a legal instruction. |
| minimize loop jumps | j | Removes a jump associated with a loop by duplicating a portion of the loop. |
| merge basic blocks | m | Merges two consecutive basic blocks *a* and *b* when *a* is only followed by *b* and *b* is only preceded by *a*. |
| dead assignment elim | h | Removes assignments when the assigned 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. |
| loop transformations | l | Performs loop-invariant code motion, recurrence elimination, loop strength reduction, and induction variable elimination on each loop ordered by loop nesting level. Each of these transformations can also be individually selected by the user. |
| remove useless jumps | u | Removes jumps and branches whose target is the following block. |
| strength reduction | q | Replaces an expensive instruction with one or more cheaper ones. |
| branch chaining | b | Replaces a branch or jump target with the target of the last jump in a jump chain. |
| remove unreachable code | d | Removes basic blocks that cannot be reached from the entry block of the function. |
| remove useless blocks | e | Removes empty blocks from the control-flow graph. |
| reverse jumps | 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. |

**Table 2: Candidate Optimization Phases in the Genetic Algorithm Experiments**

input data that was supplied with the benchmark. In these cases we aggregated such functions in the results as *unexecuted functions* since we did not have the space to list results for all of the functions. The third and fourth columns give each function's contribution, expressed as a percentage, to the dynamic and static instruction count of the whole program after applying the optimization sequence. The fifth column shows the sequence and number of optimization phases successfully applied by the batch compiler between *register assignment* and *fix entry exit* that affected the instructions. The number applied varies depending upon the size and loop structure of the function. Note these sequences of phases are applied after attempting the optimization phases that precede *register assignment* in Figure 10. 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, which is always significantly larger than the number of successfully applied phases.

The last two columns in Table 3 demonstrate 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 10 with all the `do-while`'s iterated only once. The iteration impact result shows the power of iteratively applying optimization phases until no more improvements can be found. In particular the number of instructions executed is often reduced. The only cases where the dynamic instruction 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.

The remainder of this section compares 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 what 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.

The following aspects of our experiments were identical to those performed in the Rice study. We set the population size (fixed number of sequences or chromosomes) to twenty and each of these initial sequences is randomly initialized. We sort the sequences in the population by fitness values (using the dynamic and static instruction 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 the remaining sequences from the upper half of the population and performing a crossover (mating) operation to create a pair of 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 two new sequences. Fifteen sequences are then changed (mutated) by considering each optimization phase (gene) in the sequence. Mutation of each optimization phase in the sequences occurs with a probability of 10% and 5% for the lower and upper halves of the population, respectively. When an optimization phase is mutated, it is randomly replaced with another phase. The four sequences subjected to crossover and the best performing sequence are not mutated.

There were additional differences between our experiments and the Rice study besides using a different compiler, set of optimization phases, target architecture, benchmarks, and optimization phase sequence lengths. Rather than performing a search for each program or module, we used the genetic algorithm to perform a different search for each function. In the Rice study the fitness criteria was based on code size with dynamic instruction count used as a secondary fitness value to break ties. As shown in Figure 8, we can vary both the instructions executed and code size

| program and size | function | % of dynamic | % of static | applied sequence and length | attempted phases | iteration impact | |
|---|---|---|---|---|---|---|---|
| | | | | | | dynamic % | 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 | sjmhkscllqscllslhsc (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 functions* | 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 | sjmhkscllcllsc (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 | sjmhksclllhsch (15) | 57 | -8.84 | -7.64 |
| | main | 29.98 | 39.56 | sjmhksclllehscll (17) | 58 | -1.90 | -1.23 |
| | *unexecuted functions* | 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 functions* | 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 | skscllllllllhsclllllhs (21) | 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 functions* | 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 |

**Table 3: Batch Optimization Measurements**

weight factors. For each function, we performed three different searches, which are based on static instruction count only, dynamic instruction count only, and 50% for each factor. Finally, to obtain the results in a timely fashion we only performed 100 generations instead of 1000 generations for each search.

Table 4 shows the results obtained for each function by applying the genetic algorithm. As in Table 3, *unexecuted functions* indicate those functions that were not executed using the benchmark's input data. The last six columns show the effect on dynamic and static instruction counts for each of the three sets of fitness criteria. The results that were supposed 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.

Figures 11 and 12 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

| program | functions | optimizing for speed | | optimizing for space | | optimizing for both | |
|---|---|---|---|---|---|---|---|
| | | % effect on dynamic | % effect on static | % effect on dynamic | % effect on static | % effect on dynamic | % effect on 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 functions* | **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** |
| | NumberOfBitsNeeded | **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 functions* | **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 functions* | **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** |
| stringsearch | 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 functions* | **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** |

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

from the sequence found by the batch compilation when iteratively applying optimization phases and is normalized to 1. The results show that iteratively applying 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 batch compilation baseline that iteratively applies optimization phases until no more improvements could be made. Note that many batch compilers do not iteratively apply phases and the use of a genetic algorithm to select optimization sequences will have greater benefits as compared to such noniterative 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 primarily 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 speed and space.

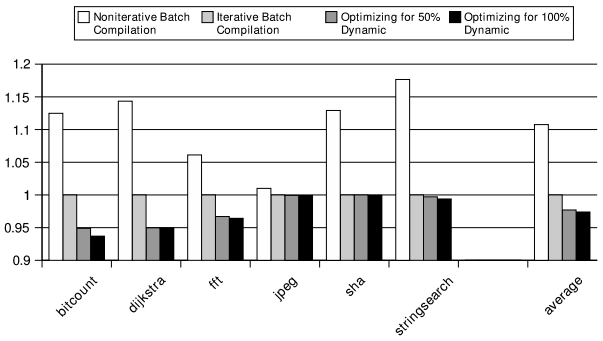


**Figure 11: Overall Effect on Dynamic Instruction Count**

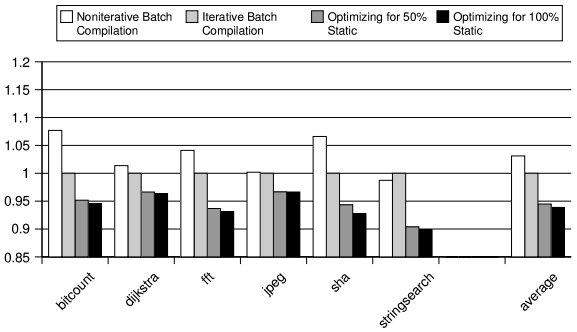


**Figure 12: Overall Effect on Static Instruction Count**

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 2 are rarely applied since they have already been applied once before *register assignment*. These are the control-flow transformations that include the third phase and the last five phases listed in Table 2. *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.

# 8. 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

| 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 | **slljckllhschllllmc** | **chllkc** | **chllskllclslsch** |
| | ntbl_bitcnt | **ckshc** | **ksc** | **ckhsc** |
| | ntbl_bitcount | ks | ks | ks |
| dijkstra | dequeue | ksc | ksc | ksc |
| | dijkstra | **chlllchkljsc** | **chkllclc** | **ckclllsclhsc** |
| | enqueue | **kshc** | **khsc** | **khsc** |
| | main | **shkllclljc** | **skhc** | **chkllclc** |
| | print_path | kch | kch | kch |
| | qcount | | | |
| fft | CheckPointer | **hkc** | **kch** | **hksc** |
| | IsPowerOfTwo | kcs | kcs | kcs |
| | NumberOfBitsNeeded | **hkjcs** | **khs** | **hkjmsc** |
| | ReverseBits | **kcjhsc** | **kcs** | **ksc** |
| | fft_float | **jkcslllchclh** | **kslllhschc** | **kclllclllhsclh** |
| | main | **sklllsjmch** | **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 | **cksllllllsclllllllhllllllch** | **llllllslllkssc** | **skclllllllhclllllsh** |
| | sha_update | **kshcjc** | **kschc** | **kschc** |
| stringsearch | init_search | **llkcjllhclc** | **ckhscllch** | **kslslllhs** |
| | main | **kslhlcjhc** | **skslhlc** | **kslhls** |
| | strsearch | **clskclhs** | **cksch** | **sllksls** |

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

space. Selecting sequences for regions of code may result in the best measures when both speed and size are considered.

# 9. CONCLUSIONS

There are several contributions that we have presented in this paper. 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 specified fitness criteria. A user can enter specific sequences and the compiler chooses the sequence that produces the best result. A user can also specify a set of phases along with options for exploring the search space of possible sequences. The user is provided with feedback describing the progress of the search and may abort the search and accept the best sequence found at that point.

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

An environment that allows a user to easily tune the sequence of optimization phases for each function in an embedded application can be very beneficial. 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.

## ACKNOWLEDGEMENTS

Clint Whaley, Bill Kreahling, and the anonymous reviewers provided helpful suggestions that improved the quality of the paper. This research was supported in part by NSF grants CCR-9904943, EIA-0072043, CCR-0208892, and ACI-0203956.

## 10. REFERENCES

[1] B. Appelbe, K. Smith, and C. McDowell, "Start/Pat: A Parallel-Programming Toolkit," *IEEE Software* **6**(4) pp. 29-40 (July 1988).

[2] M. E. Benitez and J. W. Davidson, "A Portable Global Optimizer and Linker," *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pp. 329-338 (June 1988).

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

[4] J. Browne, K. Sridharan, J. Kiall, C. Denton, and W. Eventoff, "Parallel Structuring of Real-Time Simulation Programs," *COMPCON Spring '90: Thirty-Fifth IEEE Computer Society International Conference. Intellectual Leverage. Digest of Papers.*, pp. 580-584 (February 1990).

[5] K. Chow and Y. Wu, "Feedback-Directed Selection and Characterization of Compiler Optimizations," *Workshop on Feedback-Directed Optimization*, (November 1999).

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

[7] J. W. Davidson and D. B. Whalley, "Ease: An Environment for Architecture Study and Experimentation," *Proceedings SIGMETRICS '90 Conference on Measurement and Modeling of Computer Systems*, pp. 259-260 (May 1990).

[8] J. W. Davidson and D. B. Whalley, "A Design Environment for Addressing Architecture and Compiler Interactions," *Microprocessors and Microsystems* **15**(9) pp. 459-472 (November 1991).

[9] Chyi-Ren Dow, Shi-Kuo Chang, and Mary Lou Soffa, "A Visualization System for Parallelizing Programs," *Proceedings of Supercomputing '92*, pp. 194-203 IEEE Computer Society Press, (November 1992).

[10] T. Granlund and R. Kenner, "Eliminating Branches using a Superoptimizer and the GNU C Compiler," *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 341-352 (June 1992).

[11] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *IEEE Workshop on Workload Characterization*, (December 2001).

[12] B. Harvey and G. 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).

[13] J. Holland, *Adaptation in Natural and Artificial Systems*1989.

[14] T. Kisuki, P. Knijnenburg, and M. O'Boyle, "Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation," *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pp. 237-248 (October 2000).

[15] S. Liao, A. Diwan, R. Bosch, A. Ghuloum, and M. Lam, "Suif Explorer: an Interactive and Interprocedural Parallelizer," *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 37-48 (1999).

[16] H. Massalin, "Superoptimizer - A Look at the Smallest Program," *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 122-126 (October, 1987).

[17] A. Nisbet, "Genetic Algorithm Optimized Parallelization," *Workshop on Profile and Feedback Directed Compilation*, (1998).

[18] S. Novack and A. Nicolau, "VISTA: The Visual Interface for Scheduling Transformations and Analysis," *Languages and Compilers for Parallel Computing*, pp. 449-460 (1993).

[19] D. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors," *Proceedings of 1989 International Conference on Parallel Processing*, pp. 39-48 Pennsylvania State University Press, (August 1989).

[20] S. Segars, K. Clarke, and L. Goudge, "Embedded Control Problems, Thumb, and the ARM7TDMI," *IEEE Micro* **15**(5) pp. 22-30 (October 1995).

[21] S. Vegdahl, "Phase Coupling and Constant Generation in an Optimizing Microcode Compiler," *International Symposium on Microarchitecture*, pp. 125-133 (1982).

[22] R. Whaley, A. Petitet, and J. Dongarra, "Automated Empirical Optimization of Software and the ATLAS Project," *Parallel Computing* **27**(1) pp. 3-35 (2001).

[23] D. Whitfield and M. L. Soffa, "An Approach for Exploring Code-Improving Transformations," *ACM Transactions on Programming Languages and Systems* **19**(6) pp. 1053-1084 (November 1997).

[24] W. Zhao, B. Cai, D. Whalley, M. Bailey, R. van Engelen, X. Yuan, J. Hiser, J. Davidson, K. Gallivan, and D. Jones, "VISTA: A System for Interactive Code Improvement," *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 155-164 (June 2002).