Fast Searches for Effective Optimization Phase Sequences

Prasad Kulkarni[1], Stephen Hines[1], Jason Hiser[2],
David Whalley[1], Jack Davidson[2], Douglas Jones[3]

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

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

[3]Electrical and Computer Eng. Dept, University of Illinois, Urbana, IL 61801; e-mail: dl-jones@uiuc.edu

ABSTRACT

It has long been known that a fixed ordering of optimization phases will not produce the best code for every application. One approach for addressing this phase ordering problem is to use an evolutionary algorithm to search for a specific sequence of phases for each module or function. While such searches have been shown to produce more efficient code, the approach can be extremely slow because the application is compiled and executed to evaluate each sequence's effectiveness. Consequently, evolutionary or iterative compilation schemes have been promoted for compilation systems targeting embedded applications where longer compilation times may be tolerated in the final stage of development. In this paper we describe two complementary general approaches for achieving faster searches for effective optimization sequences when using a genetic algorithm. The first approach reduces the search time by avoiding unnecessary executions of the application when possible. Results indicate search time reductions of 65% on average, often reducing searches from hours to minutes. The second approach modifies the search so fewer generations are required to achieve the same results. Measurements show that the average number of required generations decreased by 68%. These improvements have the potential for making evolutionary compilation a viable choice for tuning embedded applications.

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.

PLDI'04, June 9-11, 2004, Washington, DC, USA.

1. INTRODUCTION

The phase ordering problem has long been known to be a difficult dilemma for compiler writers [17, 19]. One sequence of optimization phases is highly unlikely to be the most effective sequence for every application (or even for each function within a single 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 [19].

One approach to deal with this problem is to search for effective optimization phase sequences using genetic algorithms [5, 11]. When the fitness criteria for such searches involve dynamic measures (e.g., cycle counts or power consumption), thousands of direct executions of an application may be required. The search time can be significant, often needing hours or days when finding effective sequences for a single application, making it less attractive for developers.

There are application areas where long compilation times are acceptable. For example, long compilation times may be tolerated in application areas where the problem size is directly related to the execution time to solve the problem. In fact, the size of many computational chemistry and high-energy physics problems is limited by the elapsed time to reach a solution (typically a few days or a week). Long compilation times may be acceptable if the resulting code allows larger problem instances to be solved in the same amount of time.

Evolutionary compilation systems have also been proposed for compilation systems targeting embedded systems where meeting strict constraints on execution time, code size, and power consumption is paramount. Here long compilation times are acceptable because in the final stages of development an application is compiled and embedded in a product where millions of units may be shipped. For embedded systems, the problem is further exacerbated because the software development environment is often different from the target environment. Obtaining performance measures on cross-platform development environments often requires simulation which can be orders of magnitude slower than native execution. Even when it is possible to use the target machine to gather performance data directly, the embedded processor may be significantly slower (slower clock rate, less memory, etc.) than available general-purpose processors. We have found that searching for an effective optimization sequence can easily require hours or days even when using direct execution on a general-purpose processor. For example, using a conventional genetic algorithm to search for effective optimization sequences for the *jpeg* application on an Ultra SPARC III processor required over 20 hours to

complete. Thus, finding effective sequences to tune an embedded application may result in an intolerably long search time.

In this paper we describe approaches for achieving faster searches for effective optimization sequences using a genetic algorithm. We performed our experiments using the VISTA (VPO Interactive System for Tuning Applications) framework [20]. One feature of VISTA is that it can automatically obtain performance feedback information which can be presented to the user and can be used to make phase ordering decisions [11]. We use this performance information to drive the genetic algorithm searches for effective optimization sequences.

The remainder of the paper is structured as follows. First, we review other aggressive compilation techniques that have been used to tune applications. Second, we give an overview of the VISTA framework in which our experiments are performed. Third, we describe methods for reducing the overhead of the searches for effective sequences. Fourth, we discuss techniques for finding effective sequences in fewer generations. Fifth, we show results that indicate the effectiveness of using our techniques to perform faster searches for optimization sequences. Finally, we outline future work and present the conclusions of the paper.

2. RELATED WORK

Prior work has used aggressive compilation techniques to improve performance. Superoptimizers have been developed that use an exhaustive search for instruction selection [12] or to eliminate branches [7]. Selecting the best combination of optimizations by turning on or off optimization flags, as opposed to varying the order of optimizations, has also been investigated [4].

Some systems perform transformations and use performance feedback information to tune applications. 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 [10, 18]. Another technique uses compile-time performance estimation [16]. All of these systems are limited in the set of optimizations they apply.

Specifications of code-improving transformations have been automatically analyzed to determine if one type of transformation can enable or disable another [19]. This information can provide insight into how to specify an effective optimization phase ordering for a conventional optimizing compiler.

A number of systems have been developed that use evolutionary algorithms to improve compiler optimizations. A neural network has been used to tune static branch predictions [3]. Genetic algorithms have been used to better parallelize loop nests [13]. Another system used genetic algorithms to derive improved compiler heuristics for hyperblock formation, register allocation, and data prefetching [15]. A low-level compilation system developed at Rice University uses a genetic algorithm to reduce code size by finding efficient optimization phase sequences [5, 6]. The Rice system uses a similar genetic algorithm as in VISTA for finding phase sequences. However, the Rice system is batch oriented instead of interactive and applies the same optimization phase order for all of the functions within a file. Some aspects of the approaches described in our paper may be useful for obtaining faster searches in all of these systems.

3. THE VISTA FRAMEWORK

This section provides a brief overview of the framework used for the experiments reported in this paper. A more detailed description of VISTA's architecture can be found in prior publications [20, 11]. 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, manually specified transformations, and queries. The compiler performs the specified actions and sends program representation information back to the viewer. Each time an optimization sequence is selected for the function being tuned, the compiler instruments the code, produces assembly code, links and executes the program, and gets performance measures from the execution. 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.

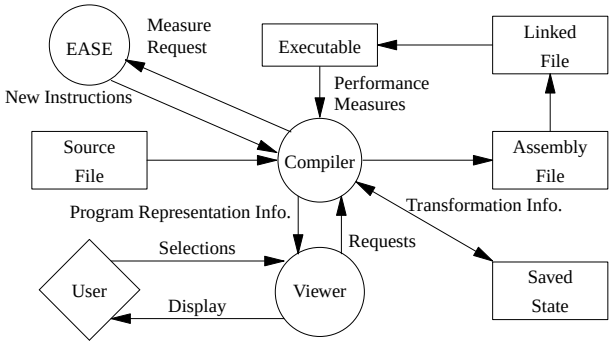

Figure 1: Interactive Code Improvement Process

The compiler used in VISTA is based on VPO (Very Portable Optimizer), which is a compiler back end that performs all of its optimizations on a single low-level representation called RTLs (register transfer lists) [1, 2]. Because VPO uses a single representation, it can apply most analyses and optimization phases repeatedly and in an arbitrary order. This feature facilitates finding more effective sequences of optimization phases.

Figure 2 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.

VISTA allows a user to specify a set of distinct optimization phases and have the compiler attempt to find the best sequence for applying these phases. Figure 3 shows the different options that we provide the user to control the search. The user specifies the *sequence length*, which is the total number of phases applied in each sequence. Our experiments used the *biased sampling search*, which 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 [9]. A population is the set of solutions (sequences)

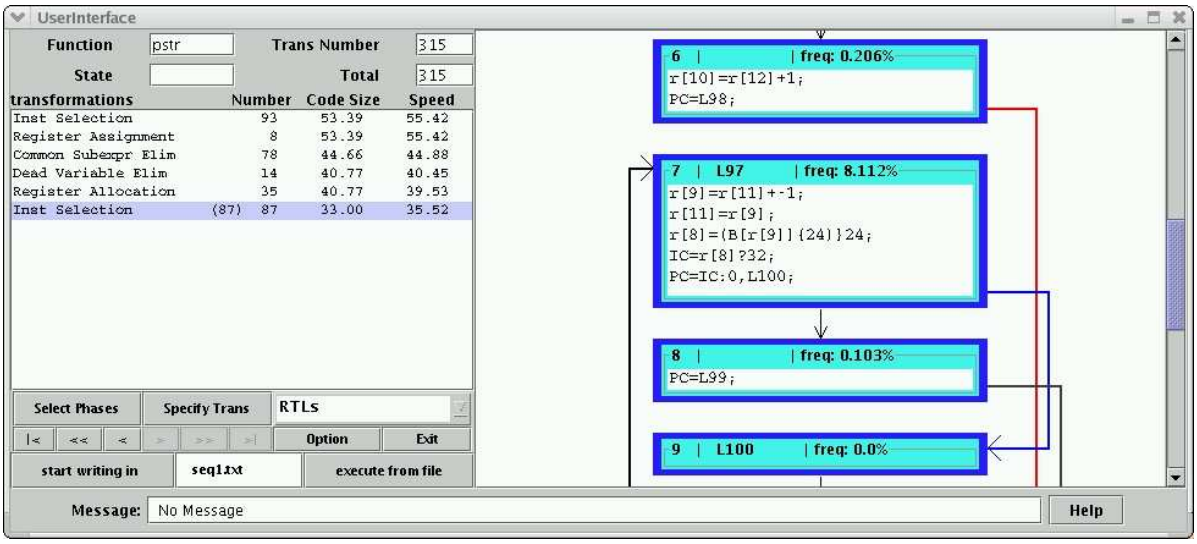

Figure 2: Main Window of VISTA Showing History of Optimization Phases

that are under consideration. The number of generations indicates how many sets of populations are to be evaluated. The population size and the number of generations limits the total number of sequences evaluated. VISTA also allows the user to choose dynamic and static weight factors, where the relative improvement of each is used to determine the overall fitness.

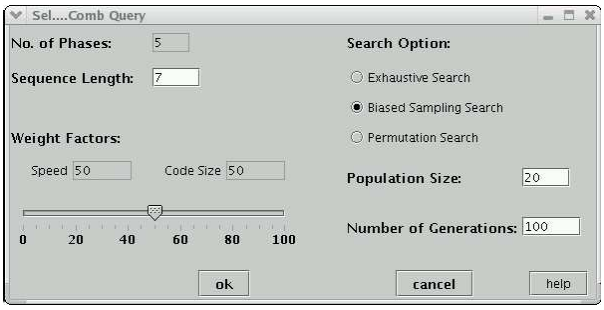

Figure 3: Selecting Options to Search for Possible Sequences

Performing these searches is time consuming, typically requiring tens of minutes for a single function, and hours or days for an entire application even when using direct execution. Thus, VISTA provides a window showing the current search status. Figure 4 shows a snapshot of the status of the search selected in Figure 3. The percentage of sequences completed, the best sequence, and its effect on performance are given. The user can terminate the search at any point and accept the best sequence found so far.

4. REDUCING THE SEARCH OVERHEAD

Performing a search for an effective optimization phase sequence can be quite expensive, perhaps requiring hours or days for an entire application even when using direct execution. One obvious benefit for speeding up these searches is that the technique is more likely to be used. Another benefit is that the search can be made more aggressive, such as increasing the number of generations, in an attempt to produce a better tuned application.

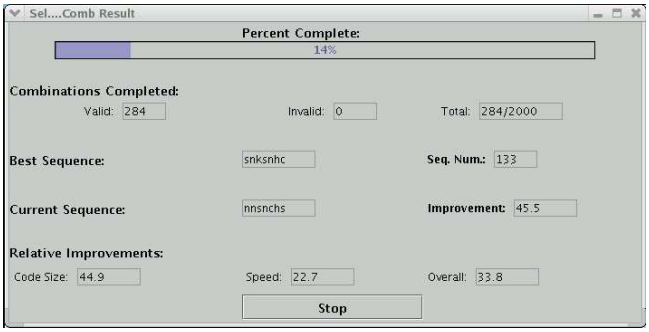

Figure 4: Window Showing the Search Status

VISTA performs the following tasks to obtain dynamic performance measurements for a single sequence. (1) The compiler applies the optimization phases in the order specified by the sequence. (2) The generated code for the function is instrumented if required to obtain performance measurements and the assembly code for that function and the remaining assembly code for the functions in the current source file are written to a file. (3) The newly generated assembly file is assembled. (4) The object files comprising the entire program are linked together into an executable by a command supplied in a configuration file. (5) The program is executed using a command in a configuration file, which may involve direct execution or simulation. As a side effect of the execution, performance measurements are produced. (6) The output of the execution is compared to the desired output to provide assurance that the new sequence did not cause the generated code to become invalid. Tasks 2-6 often dominate the search time, which is probably due to these tasks requiring I/O and task 1 being performed in memory.

The following subsections describe methods to reduce the search overhead by inferring the outcome of a sequence. Figure 5 illustrates the order in which the different methods are attempted. The methods are ordered according to cost. Each method handles

a superset of the sequences handled by the methods applied before it, but the later methods are more expensive.

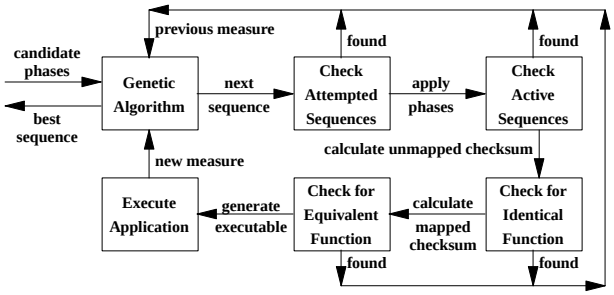

Figure 5: Methods for Reducing Search Overhead

4.1 Finding Redundant Attempted Sequences

Sometimes the same optimization phase sequence is reattempted during the search. Consider Figure 6, where each optimization phase in a sequence is represented by a letter. The same sequence can be reattempted due to mutation not occurring on any of the phases in the sequence (e.g. sequence *i* remaining the same in Figure 6). Likewise, a crossover operation or mutation changing some individual phases can produce a previously attempted sequence (e.g. sequence *k* mutates to be the same as sequence *j* before mutation in Figure 6). A hash table of attempted sequences along with the performance result for each sequence is maintained. If a sequence is found to be previously attempted, then the evaluation of the sequence is not performed and the previous result is used. This technique of using a hash table to capture previously attempted solutions has been previously used to reduce search time [5, 15, 11].

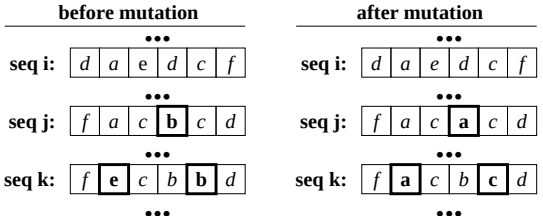

Figure 6: Example of Redundant Attempted Sequences

4.2 Finding Redundant Active Sequences

A transformation is a sequence of changes to the program representation, where the semantic behavior is preserved. A phase is a sequence of transformations caused by a single type of optimization. Borrowing from biological terminology, an *active* optimization phase (gene) is one that applies transformations, while a *dormant* optimization phase (gene) is one that has no effect. An optimization phase is dormant when the enabling conditions for the optimization to be applied are not satisfied. As one would expect, only a subset of the attempted phases in a sequence will typically be active. It is common that a dormant phase may be mutated to another dormant phase, but it would not affect the compilation. Figure 7 illustrates how different attempted sequences can map to the same active sequence, where the bold boxes represent active phases and the nonbold boxes represent dormant phases. A second hash table is used to record sequences where only the active phases are represented.

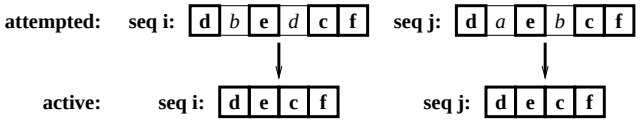

Figure 7: Example of a Redundant Active Sequence

4.3 Detecting Identical Code

Sometimes *identical* code can be generated from different active sequences. Often different optimization phases can be applied and can have the same effect. Consider the two different ways that the pair of instructions in Figure 8 can be merged together. Instruction selection symbolically merges the instructions and checks to see if the resulting instruction is legal. The same effect in this case can be produced by constant propagation followed by dead assignment elimination. We also found that performing some optimization phases in a different order will have no effect on the final code that is generated. For instance, consider applying branch chaining before and after register allocation. Both branch chaining and register allocation will neither inhibit nor enable the other phase.

original code segment
`r[2]=1;`
`r[3]=r[4]+r[2];`
 	
after instruction selection
`r[3]=r[4]+1;`
	original code segment
	`r[2]=1;`
	`r[3]=r[4]+r[2];`
	
	after constant propagation
	`r[2]=1;`
	`r[3]=r[4]+1;`
	
	after dead assignment elimination
	`r[3]=r[4]+1;`

Figure 8: Different Optimizations Having the Same Effect

VISTA has to efficiently detect when different active sequences generate identical code to be able to reduce the search overhead. A search may result in thousands of unique function instances, which may be too large to store in memory and very expensive to access on disk. The key realization in addressing this issue was that while we need to detect when function instances are identical, we can tolerate occasionally treating different instances as being identical since the sequences within a population are sorted and the best sequence found by the genetic algorithm must be completely evaluated. Thus, we calculate a CRC (cyclic redundancy code) checksum on the bytes of the RTLs and keep a hash table of these checksums. CRCs are commonly used to check the validity of data transmitted over a network and have an advantage over conventional checksums in that the order of the bytes of data does affect the result [14]. If the checksum has been generated for a previous function instance, then we use the performance results of that instance. We have verified it is rare that we generate the same checksum for different function instances and that the best fitness value found is never affected in our experiments.

4.4 Detecting Equivalent Code

Sometimes the code generated by different optimization sequences are *equivalent*, in regard to speed and size, but not identical. Consider two function instances that have the same sequence of instruction types, but use different registers. This can occur since different optimization phases compete for registers. For instance, consider the source code in Figure 9(a). Figures 9(b) and 9(c) show two possible translations given two different orderings of optimization phases that consume registers.

To detect this situation, we identify the live ranges of all of the registers in the function and map each live range to a distinct pseudo register. Equivalent function instances become identical after mapping, which is illustrated for the example in Figure 9(d). We compute the CRC checksum for the mapped function instance and check in a separate hash table of CRC checksums to see if the mapped function had been previously generated.

```
sum = 0;
for (i = 0; i < 1000; i++)
   sum += a[i];
```

(a) Source Code

```
  r[10]=0;
  r[12]=HI[a];
  r[12]=r[12]+LO[a];
  r[1]=r[12];
  r[9]=4000+r[12];
L3
  r[8]=M[r[1]];
  r[10]=r[10]+r[8];
  r[1]=r[1]+4;
  IC=r[1]?r[9];
  PC=IC<0,L3;
```

(b) Register Allocation before Code Motion

```
  r[11]=0;
  r[10]=HI[a];
  r[10]=r[10]+LO[a];
  r[1]=r[10];
  r[9]=4000+r[10];
L3
  r[8]=M[r[1]];
  r[11]=r[11]+r[8];
  r[1]=r[1]+4;
  IC=r[1]?r[9];
  PC=IC<0,L3;
```

(c) Code Motion before Register Allocation

```
  r[32]=0;
  r[33]=HI[a];
  r[33]=r[33]+LO[a];
  r[34]=r[33];
  r[35]=4000+r[33];
L3
  r[36]=M[r[34]];
  r[32]=r[32]+r[36];
  r[34]=r[34]+4;
  IC=r[34]?r[35];
  PC=IC<0,L3;
```

(d) After Mapping Registers

Figure 9: Different Functions with Equivalent Code

On most machines there is a uniform access time for each register in the register file. Likewise, most statically scheduled processors do not generate stalls due to anti (write after read) and output (write after write) dependences. However, these dependences could inhibit future optimizations. Thus, comparing register mapped functions to avoid executions in the search should only be performed after all remaining optimizations (e.g. filling delay slots) have been applied. Given that these assumptions are true, if we find that the current mapped function is equivalent to a previous mapped instance of the function, then we can assume the two are equivalent and will produce the same result after execution.

5. PRODUCING SIMILAR RESULTS IN FEWER GENERATIONS

Another approach that can be used to reduce the search time for finding effective optimization sequences is to produce the same results in fewer generations of the genetic algorithm. If this approach is feasible, then users can either specify fewer generations to be performed in their searches or they can stop the search sooner once the desired results have been achieved. The following subsections describe the different techniques that we use to obtain effective sequences of optimization phases in fewer generations. All of these techniques identify phases that are likely to be active or dormant at a given point in the compilation process.

5.1 Using the Batch Sequence

The traditional or *batch* version of our compiler always attempts the same order of optimization phases for each function. We obtain the sequence of active phases (those phases that were able to apply one or more transformations) from the batch compilation of the function. We have used the length of the active batch sequence to establish the length of the sequences attempted by the genetic algorithm in previous experiments [11].

We propose to use the active batch sequence for the function as one of the sequences in the initial population. The premise is that if we initialize a sequence in the population with optimization phases that are likely to be active, then this may allow the genetic algorithm to converge faster on the best sequence it can find. This approach is similar to including in the initial population the compiler writer's manually specified priority function when attempting to tune a compiler heuristic [15].

5.2 Prohibiting Specific Phases

While many different optimization phases can be specified as candidate phases for the genetic algorithm, sometimes specific phases can never be active for a given function. If the genetic algorithm only attempts phases that have an opportunity to be active, then the algorithm may converge on the best sequence it can find in fewer attempts. There are several situations when specific optimizations should not be attempted. Loop optimization phases cannot be active for a function that does not contain any loops. Register allocation in VPO cannot be active for a function that does not contain any local variables or parameters. Branch optimizations and unreachable code elimination cannot be active for a function that contains a single basic block. Detecting that a specific set of optimization phases can never be active for a given function requires simple analysis that only needs to be performed once at the beginning of the genetic algorithm.

5.3 Prohibiting Prior Dormant Phases

When compiling a function, we find certain optimization phases will be dormant given that a specific prefix of active phases has been performed. Given that the same prefix of phases is attempted again, there is no benefit from attempting the same dormant phase in the same situation since it will remain dormant. To avoid repeating these dormant phases, we represent the active phases as nodes in a tree, where each child corresponds to the next phase in an active sequence. We also store at each node the set of phases

that were found to be dormant for that prefix of active phases. Figure 10 shows an example tree where the bold portions represent active prefixes and the nonbold boxes represent dormant phases given that prefix. For instance, *a* and *f* are dormant phases for the prefix **bac**. To prohibit applying a prior dormant phase, we force a phase to change during mutation until we find a phase that has either been active with the specified prefix or has not yet been attempted.

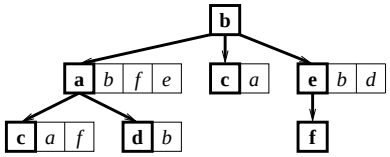

Figure 10: A Tree Representing Active Prefixes

5.4 Prohibiting Unenabled Phases

Certain optimization phases when performed cannot become active again until enabled. For instance, register allocation replaces references to variables in live ranges with registers. A live range is assigned to a register when a register is available at that point in the coloring process. After the compiler applies register allocation, this optimization phase will not have an opportunity to be active again until the register pressure has changed. Unreachable code elimination and a variety of branch optimizations will not affect the register pressure and thus will not enable register allocation. Figure 11 illustrates that a specific phase, the nonbold box of the sequence on the right, will at times be unenabled and cannot be active. Again the premise is that if the genetic algorithm concentrates on the phases that have an opportunity to be active, then it will be able to apply more active phases in a sequence and converge to the best sequence it can find in fewer attempts. Note that determining which optimization phases can enable another phase requires careful consideration by the compiler writer.

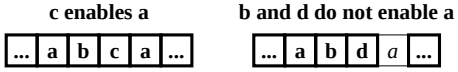

Figure 11: Enabling Previously Applied Phases

We implemented this technique by forcing a phase to mutate if the same phase has already been performed and there are no intervening phases that can enable it. We realized that a specific phase can become unenabled after an attempted phase is found to be active or dormant. We first follow the tree of active prefixes, which was described in the previous subsection, to determine which phases are currently enabled. For example, consider again Figure 10. Assume that **b** can be enabled by **a**, but cannot be enabled by **c**. Given the prefix **bac**, we know that **b** cannot be active at this point since **b** was dormant after the prefix **ba** and **c** cannot reenable it. After reaching a leaf of the tree we track which phases cannot be enabled by just examining the subsequently attempted phases.

6. EXPERIMENTS

This section describes the results of a set of experiments to illustrate the effectiveness of the previously described techniques for obtaining fast searches for effective optimization phase sequences. We first perform experiments on a Ultra SPARC III processor so that the results could be obtained in a reasonable time. After ensuring ourselves that the techniques were sound, we use these techniques when obtaining results for the Intel StrongARM SA-110 processor, which has a clock rate that is more than 5 times slower than the Ultra SPARC III.

We used a subset of the *mibench* benchmarks, which are C applications targeting specific areas of the embedded market [8]. We used one benchmark from each of the six categories of applications. When executing each of the benchmarks, we used the sample input data that was provided with the benchmark. Table 1 contains descriptions of these programs.

Category	Program	Description
auto/industrial	bitcount	test bit manipulation abilities
network	dijkstra	calculates shortest path between nodes using Dijkstra's algorithm
telecomm	fft	performs fast fourier transform
consumer	jpeg	image compression & decompression
security	sha	secure hash algorithm
office	stringsearch	searches for words in phrases

Table 1: MiBench Benchmarks Used in the Experiments

Table 2 shows each of the candidate code-improving phases that we used in the experiments when compiling each function. In addition, register assignment, which is a compulsory phase that assigns pseudo registers to hardware registers, has to be performed. VISTA implicitly performs register assignment before the first code-improving phase in a sequence that requires it. After applying the last code-improving phase in a sequence, we perform another compulsory phase which inserts instructions at the entry and exit of the function to manage the activation record on the run-time stack. Finally, we also perform additional code-improving phases afterwards, such as filling delay slots.

Our genetic algorithm search for obtaining the baseline measurements was accomplished in the following manner. Unlike past studies using genetic algorithms to generate better code [13, 5, 15], we perform a search on each function (a total of 106 functions in our test suite), which requires longer compilations but results in better overall improvements [11]. In fact, most of the techniques we are evaluating would be much less effective if we searched for a single sequence to be applied on an entire application. We set the sequence (chromosome) length to be 1.25 times the number of active phases that were applied for the function by the batch compiler. We felt this length was a reasonable limit and gives us an opportunity to apply more active phases than what the batch compiler could accomplish, which is much less than the number of phases attempted during the batch compilation. The sequence lengths used in these experiments varied between 4 and 48 with an average of 14.15. We set the population size (fixed number of sequences or chromosomes) to twenty and each of these initial sequences is randomly initialized with candidate optimization phases. We performed 100 generations when searching for the best sequence for each function. We sort the sequences in

Optimization Phase	Description
branch chaining	Replaces a branch or jump target with the target of the last jump in a jump chain.
common subexpression elimination	Eliminates fully redundant calculations, which also includes constant and copy propagation.
remove unreachable code	Removes basic blocks that cannot be reached from the entry block of the function.
remove useless blocks	Removes empty blocks from the control-flow graph.
dead assignment elimination	Removes assignments when the assigned value is never used.
block reordering	Removes a jump by reordering basic blocks when the target of the jump has only a single predecessor.
minimize loop jumps	Removes a jump associated with a loop by duplicating a portion of the loop.
register allocation	Replaces references to a variable within a specific live range with a register.
loop transformations	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.
merge basic blocks	Merges two consecutive basic blocks *a* and *b* when *a* is only followed by *b* and *b* is only preceded by *a*.
evaluation order determination	Reorders RTLs in an attempt to use fewer registers.
strength reduction	Replaces an expensive instruction with one or more cheaper ones.
reverse jumps	Eliminates an unconditional jump by reversing a conditional branch when it branches over the jump.
instruction selection	Combine instructions together and perform constant folding when the combined effect is a legal instruction.
remove useless jumps	Removes jumps and branches whose target is the following block.

Table 2: Candidate Optimization Phases in the Genetic Algorithm Experiments

the population by a *fitness value* calculated using 50% weight on speed and 50% weight on code size. The speed factor we used was the number of instructions executed since this was a measure that could be consistently obtained, it has been used in similar studies [5, 11], and allowed us to obtain baseline measurements within a reasonable period of time. We could obtain a more accurate measure of speed by using a cycle-accurate simulator. However, the main point of our experiments was to evaluate the effectiveness of techniques for obtaining faster searches, which can be applied with any type of fitness evaluation criteria. 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 two new sequences. Fifteen sequences are then changed (mutated) by considering each optimization phase (gene) in the sequence. Mutation of each phase in a sequence 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. Finally, if we find identical sequences in the same population, then we replace the redundant sequences with ones that are randomly generated.

Figures 12, 13, and 14 show the percentage improvement that we obtained for the SPARC when optimizing for speed only, size only, and 50% for each factor, respectively. Performance results for the ARM, a widely used embedded processor, are presented later in this section. The baseline measures were obtained using the batch VPO compiler, which iteratively applies optimization phases until no more improvements can be obtained. This baseline is much more aggressive than always using a fixed length sequence of phases [11]. The average benefits shown in the figure are slightly improved from previously published results [11] since the searches now include additional optimization phases that were not previously exploited by the genetic algorithm. Note that the contribution of our paper is that the search for these benefits is more efficient, rather than the actual benefits obtained.

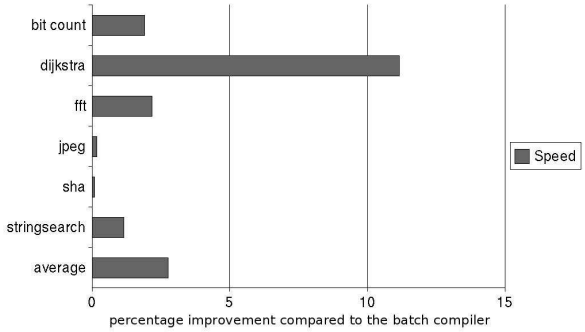

Figure 12: Speed Only Improvements for the SPARC

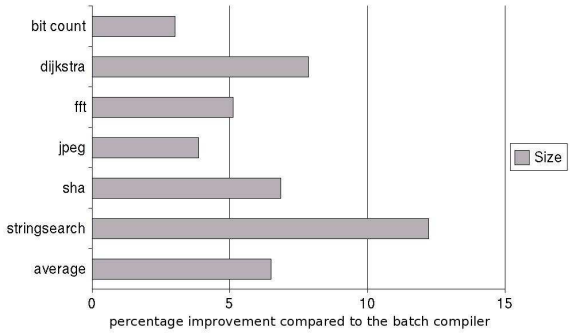

Figure 13: Size Only Improvements for the SPARC

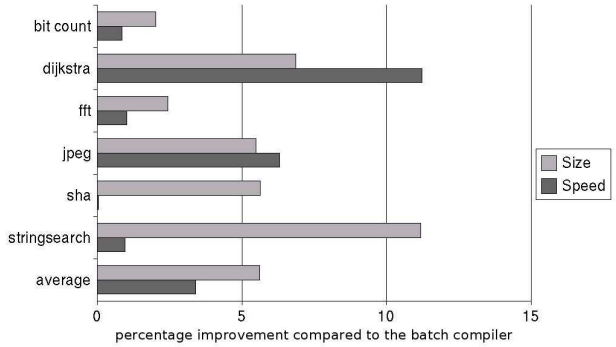

Figure 14: Size and Speed Improvements for the SPARC

Figure 15 shows the average number of sequences whose executions were avoided for each benchmark using the methods described in Section 4. These results do not include the functions in the benchmarks that were not executed when using the sample input data since these functions were evaluated on code size only and did not require execution of the application. Consider for now only the top bar for each benchmark, which represents the results without applying any of the techniques in Section 5. As mentioned previously, each method in Section 4 is able to find a superset of the sequences handled by methods applied before it. On average 41.3% of the sequences were detected as redundantly attempted, 27.0% were caught as redundant active sequences, 14.9% were discovered to produce identical code as generated by a previous sequence, and 1.0% were found to produce unique, but equivalent code. Thus, over 84% of the executions were avoided. We found that we could avoid a higher percentage of the executions when tuning smaller functions since we used shorter sequence lengths that were established by the batch compilation due to fewer optimization phases being active. A shorter sequence length results in more redundant sequences. For instance, the likelihood of mutation is less when there are fewer phases in a sequence to mutate. Also, identical or equivalent code is more likely when fewer phases could be applied.

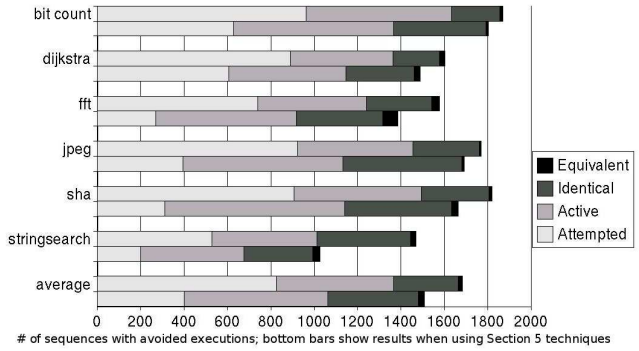

Figure 15: Number of Avoided Executions

Figure 16 shows the relative search time required when applying the methods described in Section 4 to not applying these methods. The average search time required 0.35 of the time when no executions were avoided and 0.51 of the time when redundant attempted sequences were avoided. The average time required to evaluate each of the six benchmarks improved from 5.57 hours to 2.27 hours. The reduction appears to be affected not only by the percentage of the avoided executions, but also by the size of the functions. The larger functions tended to have fewer avoided executions and also had longer compilations. While the average search time was significantly reduced for these experiments using direct execution on a SPARC processor, the savings would only increase when using simulation since the executions of the application would comprise a larger portion of the search time.

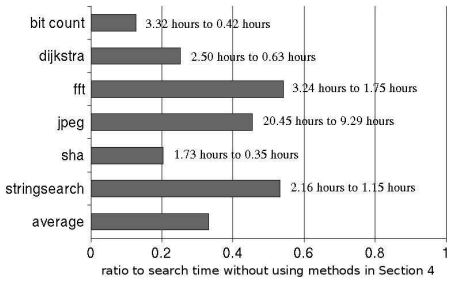

Figure 16: Relative Total Search Time

Figures 17-21 show the average number of generations that were evaluated for each of the functions before finding the best fitness value in the search. The *baseline* result is without using any of the techniques described in Section 5. The other results indicate the generation when the first sequence was found whose performance equaled the best sequence found in the baseline search. To ensure a fair comparison, we did not include the results for the functions when the best fitness value found was not identical to the best fitness value in the baseline, which occurred on about 18% of the functions. This caused the baseline results to vary slightly since the functions with different fitness values were not always the same when applying each of the techniques. About 11.3% of the functions had improved fitness values and about 6.6% of the functions had worse fitness values when *all* of the techniques were applied. On average the best fitness values improved by 0.24% (by 1.33% for only the differing functions). The maximum number of generations before finding the best fitness value for any function was 91 out of a possible 100 when not applying any of the four techniques. The maximum was 56 when all four techniques were used. The techniques occasionally caused the best fitness value to be found later, which we believe is due to the inherent randomness of using a genetic algorithm. However, all of the techniques were beneficial on average.

Figure 17 shows the effect of *using the batch sequence* in the initial population, which in general was quite beneficial. We found that this technique worked well for the smaller functions in the applications since it was often the case that the batch compiler produced code that was as good as the code generated by the best sequence found in the search. However, the smaller functions tended to converge on the best sequence in the search in fewer generations anyway since the sequence lengths were typically shorter. In fact, it is likely that performing a search for an effective optimization sequence is in general less beneficial for smaller functions since there is less interplay between phases. Using the batch sequence for the larger functions often resulted in finding the best sequence in fewer generations even though the batch compiler typically did not produce code that was as good as produced by the best sequence found in the baseline results. Thus,

simply initializing the population with one sequence containing phases that are likely to be active is quite beneficial.

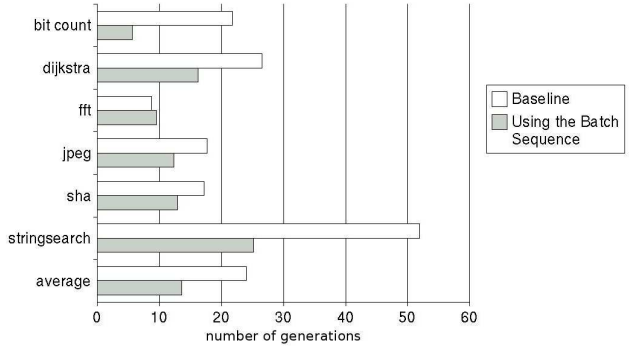

Figure 17: Number of Generations before Finding the Best Fitness Value When Using the Batch Sequence

The effect of *prohibiting specific phases* throughout the search was less beneficial, as shown in Figure 18. Specific phases can only be safely prohibited when the function is relatively simple and a specific condition (such as no loops, no variables, or no unconditional jumps) can be detected. Several applications, such as *stringsearch*, had no or very few functions that met these criteria. The simpler functions also tended to converge faster to the best sequence found in the search since the sequence length established by the length of the batch compilation was typically shorter. Likewise, the simpler functions also have little impact on the size of the entire application and have little impact on speed when they are not frequently executed.

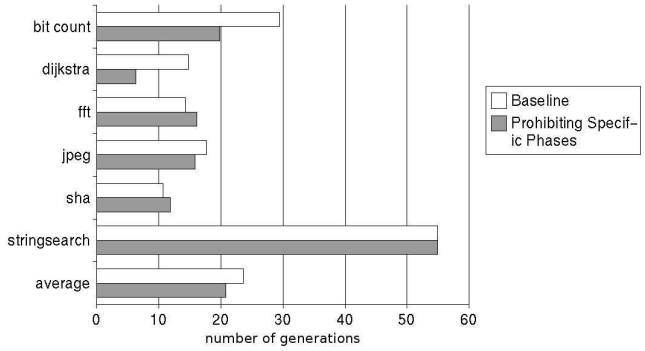

Figure 18: Number of Generations before Finding the Best Fitness Value When Prohibiting Specific Phases

In contrast, *prohibiting prior dormant* and *unenabled phases*, which are depicted in Figures 19 and 20, had a more significant impact since these techniques could be applied to all functions. Without using these two techniques, it was often the case that many phases were reattempted when there was no opportunity for them to be active.

Applying *all* the techniques produced the best overall results, as shown in Figure 21. In fact, only about 32% of the generations on average (from 25.74 to 8.24) were required to find the best sequence in the search as compared to the baseline. As expected, applying all of the techniques did not result in the sum of the benefits of the individual techniques since some of the phases that were prohibited would be caught by multiple techniques.

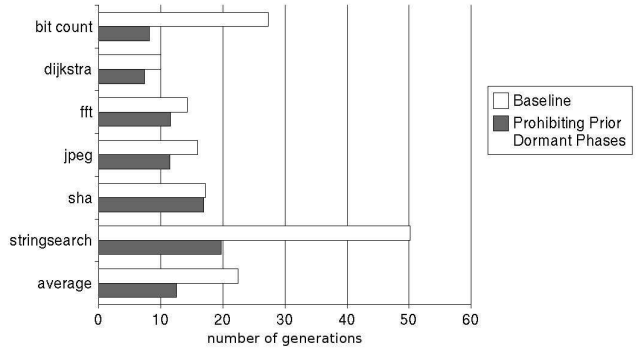

Figure 19: Number of Generations before Finding the Best Fitness Value When Prohibiting Prior Dormant Phases

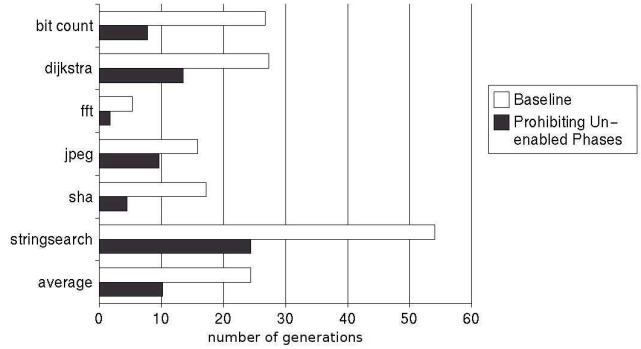

Figure 20: Number of Generations before Finding the Best Fitness Value When Prohibiting Unenabled Phases

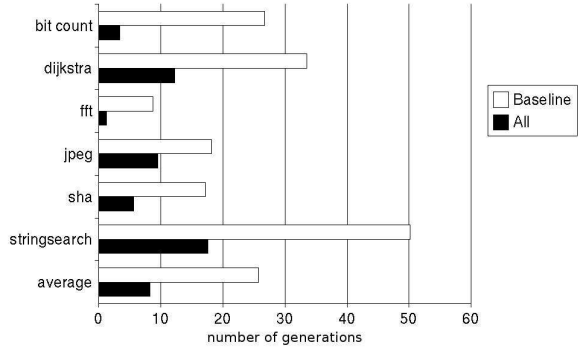

Figure 21: Number of Generations before Finding the Best Fitness Value When Applying All Techniques

Consider again Figure 15, which depicts the number of avoided executions. The bottom bar for each benchmark shows the number of executions that are avoided when all of the techniques described in Section 5 are applied. One can see that while the number of redundantly attempted sequences decrease, the number of sequences caught by the three other techniques increase. The remaining redundantly attempted sequences were the sequences created by the crossover operation and the best sequence in the population, which were not subject to mutation, and the redundant sequences with only active phases. The average number of avoided executions decreases by about 10%, which means a greater number of functions with unique code were

generated. However, the decrease in avoided executions is much less than the average decrease in generations required to reach the best sequence found in the search, as shown in Figure 21.

Figure 22 shows the relative time for finding the best fitness value when all of the techniques in Section 5 were applied. The actual times are shown in minutes since finding the best sequence is accomplished in a fraction of the total generations performed in the search. Note the baseline for finding the best fitness value includes all of the methods described in Section 4 to avoid unnecessary executions. The best fitness value was found in 53.0% of the time on average as compared to the baseline.

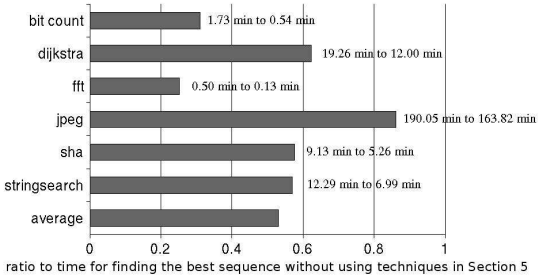

Figure 22: Relative Search Time before Finding the Best Fitness Value

After ensuring that the techniques we developed to improve the search time for effective sequences were sound, we obtained results on the Intel StrongARM SA-110 processor. Figures 23, 24, and 25 show the percentage improvement when optimizing for speed only, size only, and 50% for each factor, respectively. The average time required to obtain results for each of the benchmarks when optimizing for both speed and size on the ARM required 12.67 hours. Using the average ratio shown in Figure 16, we estimate it would have taken over 36.19 hours without applying the techniques in Section 4.

7. IMPLEMENTATION ISSUES

During the process of this investigation, we encountered several implementation issues that made this work challenging. First, producing code that always generates the correct output for different optimization phase sequences is difficult. Even implementing a conventional compiler that always generates code that produces correct output when applying one predefined sequence of optimization phases is not an easy task. In contrast, generating code that always correctly executes for thousands of different optimization phase sequences is a severe stress test. Ensuring that all sequences in the experiments produced valid code required tracking down many errors that had not yet been discovered in the VISTA system. Second, the techniques presented in Sections 5.2 and 5.4 required analysis and judgement by the compiler writer to determine when optimization phases will be enabled. We inserted sanity checks when running experiments without using these methods to ensure that our assertions concerning the enabling of optimization phases were accurate. We found several cases where our reasoning was faulty after inspecting the situations uncovered by these sanity checks and we were able to correct our enabling assertions. Third, we sometimes found that dormant optimization phases did have unexpected side effects by changing the analysis information, which could enable or disable a subsequent

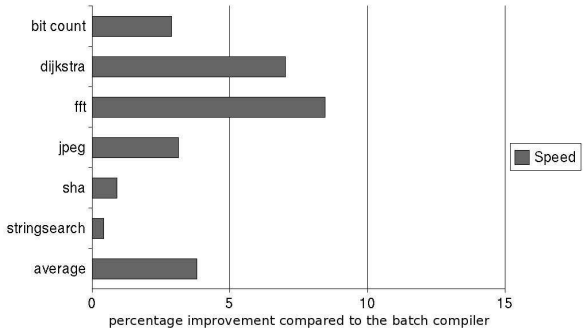

Figure 23: Speed Only Improvements for the ARM

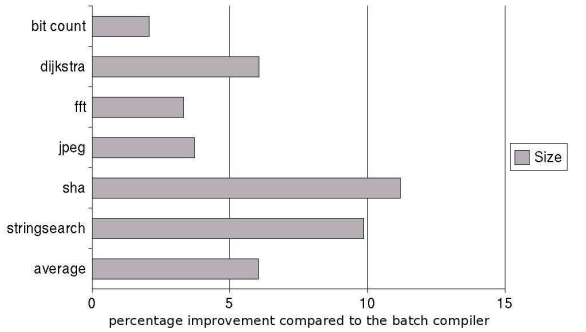

Figure 24: Size Only Improvements for the ARM

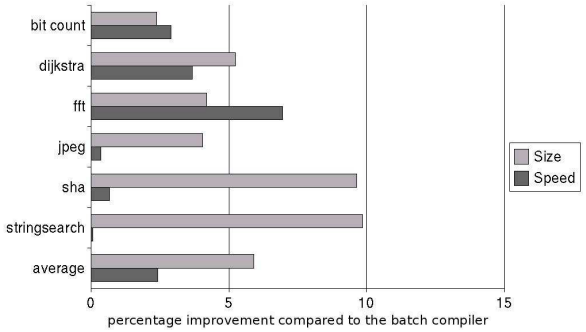

Figure 25: Size and Speed Improvements for the ARM

optimization phase. These side effects can affect the results of the methods described in Sections 4.2, 5.3, and 5.4. We also inserted sanity checks to ensure that different dormant phases did not cause different effects on subsequent phases. We detected when these situations occurred, properly set the information about what analysis is required and invalidated by each optimization phase, and now rarely encounter these problems.

8. FUTURE WORK

There is much future research that can be accomplished on providing fast searches for effective optimization sequences. We have shown that detecting when a particular optimization phase will be dormant can result in fewer generations to converge on the best sequence in the search. We believe it is possible to estimate the likelihood that a particular optimization phase will be active

given the active phases that precede it by empirically collecting this information. This information could be exploited by adjusting the mutation operation to more likely mutate to phases that have a better chance of being active with the goal of converging to a better fitness value in fewer generations.

Another area of future work is to vary the characteristics of the search. It would be interesting to see the effect on a search as one changes aspects of genetic algorithm, such as the sequence length, population size, number of generations, etc. We may find that certain search characteristics may be better for one class of functions, while other characteristics may be better for other functions. In addition, it would be interesting to perform searches involving more compiler optimizations and benchmarks.

Finally, the use of a cluster of processors can reduce the search time. Certainly different sequences within a population can be evaluated in parallel [15]. Likewise, functions within the same application can be evaluated independently. Even with the use of a cluster, the techniques we have presented in our paper would still be useful since they will further enhance the search time. In addition, not every developer has access to a cluster.

9. CONCLUSIONS

There are several contributions that we have presented in this paper. First, we have shown there are effective methods to reduce the search overhead for finding effective optimization phase sequences by avoiding expensive executions or simulations. Detecting when a phase was active or dormant by instrumenting the compiler was very useful since many sequences can be detected as redundant by memoizing the results of active phase sequences. We also discovered that the same code is often generated by different sequences. We demonstrated that using efficient mechanisms, such as a CRC checksum, to check for identical or equivalent functions can also significantly reduce the number of required executions of an application. Second, we have shown that on average the number of generations required to find the best sequence can be reduced by over two thirds. One simple, but effective technique is to insert the active sequence of phases from the batch compilation as one of the sequences in the initial population. We also found that we could often use analysis and empirical data to determine when phases could not be active. These techniques result in faster convergence to more effective sequences, which can allow equally effective searches to be performed with fewer generations of the genetic algorithm.

An environment to tune the sequence of optimization phases for each function in an embedded application can be very beneficial. However, the overhead of performing searches for effective sequences using a genetic algorithm can be quite significant and this problem is exacerbated when performance measurements for an application are obtained by simulation or on a slower embedded processor. Many developers are willing to wait for tasks to run overnight to improve a product, but are unwilling to wait longer. We have shown that the search overhead can be significantly reduced, perhaps to a tolerable level, by using methods to avoid redundant executions and techniques to converge to the best sequence it can find in fewer generations.

ACKNOWLEDGEMENTS

Clark Coleman and the anonymous reviewers provided helpful suggestions that improved the quality of the paper. This research was supported in part by National Science Foundation grants EIA-0072043, ACI-0203956, CCR-0208892, ACI-0305144, and CCR-0312493.

10. REFERENCES

[1] 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).

[2] 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).

[3] B. Calder, D. Grunwald, and D. Lindsay, "Corpus-based Static Branch Prediction," *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 79-92 (June 1995).

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

[5] 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).

[6] K. Cooper, D. Subramanian, and L. Torczon, "Adaptive Optimizing Compilers for the 21st Century," *Journal of Supercomputing* **23**(1) pp. 7-22 ().

[7] 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).

[8] 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).

[9] J. Holland, *Adaptation in Natural and Artificial Systems*, Addison-Wesley (1989).

[10] 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).

[11] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan, "Finding Effective Optimization Phase Sequences," *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 12-23 (June 2003).

[12] 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).

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

[14] W. Peterson and D. Brown, “Cyclic Codes for Error Detection,” *Proceedings of the IRE* **49** pp. 228-235 (January 1961).

[15] M. Stephenson, S. Amarasinghe, M. Martin, and U. O’Reilly, “Meta Optimization: Improving Compiler Heuristics with Machine Learning,” *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 77-90 (June 2003).

[16] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August, “Compiler Optimization Space-Exploration,” *ACM SIGMICRO International Symposium on Code Generation and Optimization*, (March 2003).

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

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

[19] 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).

[20] 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).