Fast and Efficient Searches for Effective Optimization Phase Sequences

PRASAD A. KULKARNI, STEPHEN R. HINES, and DAVID B. WHALLEY
Florida State University
JASON D. HISER and JACK W. DAVIDSON
University of Virginia
DOUGLAS L. JONES
University of Illinois at Urbana-Champaign

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 possibly executed to evaluate each sequence's effectiveness. Consequently, evolutionary or iterative compilation schemes have been promoted for compilation systems targeting embedded applications where meeting strict constraints on execution time, code size, and power consumption is paramount and longer compilation times may be tolerated in the final stage of development when an application is compiled one last time and embedded in a product. Unfortunately, even for small embedded applications, the search process can take many hours or even days making the approach less attractive to developers. 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 62% 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 this approach decreases the average number of required generations by 59%. 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

Additional Key Words and Phrases: phase ordering, interactive compilation, genetic algorithms

1. INTRODUCTION

The phase ordering problem has long been known to be a difficult dilemma for compiler designers [Vegdahl 1982; Whitfield and Soffa 1997]. 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

A preliminary version of this research was described in the *ACM SIGPLAN '04 Conference on Programming Language Design and Implementation* under the title "Fast Searches for Effective Optimization Phase Sequences."
Authors' addresses: P. Kulkarni, S. Hines, and D. Whalley, Computer Science Department, Florida State University, Tallahassee, FL 32306-4530; e-mail: {kulkarni,hines,whalley}@cs.fsu.edu; phone: (850) 644-3506; J. Hiser and J. Davidson, Computer Science Department, University of Virginia, Charlottesville, VA 22904; e-mail: {hiser,jwd}@virginia.edu; phone: (434) 982-2209; D. Jones, Electrical and Computer Engineering Department, University of Illinois at Urbana-Champaign, Urbana, IL 61801; e-mail: dl-jones@uiuc.edu; phone: (217) 244-6823

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 phases, and the target architecture [Whitfield and Soffa 1997].

One approach to deal with this problem is to search for effective optimization phase sequences using genetic algorithms [Cooper, Schielke and Subramanian 1999; Kulkarni, Zhao et. al. 2003]. 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 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 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 [Zhao et. al. 2002]. VISTA allows a user to interact with a compiler backend to tune applications. For example, VISTA can obtain and present performance information which can be used by an application developer to make phase ordering decisions [Kulkarni, Zhao et. al. 2003]. 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 for tuning applications. Second, we give an overview of the VISTA framework in which our experiments are performed. Third, we describe and evaluate methods for reducing the overhead of the searches for effective sequences. Fourth, we discuss and evaluate techniques for finding effective sequences in fewer generations. Finally, we outline future work and present the conclusions of the paper.

2. RELATED WORK

Several groups have worked on the problem of attempting to find the best sequence of compiler optimization phases and/or optimization parameters in an attempt to reduce execution time, code size, and/or power consumption. Specifications of code-improving transformations have been automatically analyzed to determine if one type of transformation can enable or disable another [Whitfield and Soffa 1997]. This information can provide insight into how to specify an effective optimization phase ordering for a conventional optimizing compiler. While it was found that many pairs of optimization phases may not enable or disable another, there are still many instances where one phase can affect the optimization opportunities for another. In such cases, the determination of the order dependent phases cannot be automated and resolving the order requies detailed knowledge of the compiler. This work differs from ours in that we search for effective optimization sequences when compiling specific functions rather than one general optimization sequence to be used when compiling any program.

There has been some work on using iterative compilation to tune compilation heuristics. A neural network has been used to tune static branch predictions [Calder et. al. 1995]. Another system used genetic algorithms to derive improved compiler heuristics for hyperblock formation, register allocation, and data prefetching [Stephenson et. al. 2003]. While these searches are used to tune compiler heuristics instead of individual applications, compiler tuning searches can still be quite time consuming.

There has been much research investigating searches for optimization sequences to improve the code for individual applications. When the search space is relatively small, then exhaustive approaches can be used. Such techniques have been developed to search for optimal instruction sequences [Massalin 1987] or to eliminate branches [Granlund and Kenner 1992]. However, these approaches can only be used in very limited contexts.

When the search space is too large to be exhaustively searched, techniques are needed to intelligently probe or prune the space and/or to provide fast evaluation of a specific sequence. Rather than changing the order of optimization phases, there has been work done on attempting to find the best set of optimizations to turn on or off using optimization flags to a conventional compiler. User-supplied information, profile information, and static heuristics have been used to recommend optimization flag options [Granston and Holler 2001]. Searches for the best combination of optimization flags using fractional factorial design have also been investigated [Chow and Wu 1999]. In contrast, our system supports changing the order of the optimization phases rather than just determining whether or not an optimization should be applied.

Many researchers have developed automatic searches for finding efficient optimization sequences where the order and/or the parameters of the optimization phases have been varied. Some have concentrated more on optimization parameters rather than the order of optimization phases. Iterative techniques using actual execution times after each compilation have been applied to determine good optimization parameters, such as tile sizes and loop unroll factors, for specific programs or library routines [Kisuki et. al. 2000; Whaley et. al. 2001]. These researchers have used grid and line search based algorithms to attempt to find a combination of parameters that produces the most efficient code.

Other researchers have also varied how optimizations are applied and have instead used static estimations of performance to reduce the search time. One method searches through the different ways to apply loop fision, fusion, interchange, and outer loop unrolling in an attempt to optimize loop nests [Wolf et. al. 1996]. This method does not actually generate code, but instead uses an estimate based on the original loop nest and the potential benefit for a transformation. Thus, their approach is limited since the estimator only works on the set of optimizations being considered. The search space is pruned in different ways. The decisions regarding how to apply some optimizations, such as outer loop unrolling factors, are made independently from other optimizations since it was felt that it would not be affected by how inner loops would be optimized. The number of loops to be varied when tuning other optimizations, such as tile size and loop interchange, are also limited. Other methods for integrating different optimization phases were also studied [Irigoin and Triolet 1988; Gao et. al. 1992].

Another method, called Optimization-Space Exploration, also uses static performance estimators to reduced search time [Triantafyllis et. al. 2003]. This approach is very general since code for critical segments are actually generated and a static performance estimation is applied. Thus, any set of optimizations could be used in this approach. In order to prune the search space, they limited the number of configurations of optimization-parameter value pairs to those that are likely to contribute to performance improvements. The compiler also prunes other remaining configurations based on the success of the configurations it has already tried. The optimizations under consideration include coalescing multiple adjacent loads and stores, loop unrolling, software pipelining, and if-conversion. Beside the differences in the search and optimizations used, these systems also concentrate on critical segments of code, where our system attempts to tune entire functions.

The prior work that is most related to the VISTA system for finding effective optimization sequences is the low-level compilation system developed at Rice University, which also searches for efficient optimization phase sequences using a genetic algorithm to reduce code size and dynamic instruction count [Cooper, Schielke and Subramanian 1999; Cooper, Subramanian and Torczon 2002]. The Rice system uses a similar genetic algorithm as in VISTA for finding phase sequences and in fact was the inspiration for much of our work in this paper. They also attempted to characterize the space that such an adaptive compiler must search [Almagor et. al. 2004]. However, the Rice system is strictly 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 many of these systems. Our work on avoiding redundant sequences of optimization phases essentially guarantees when performance will be identical to a sequence that has been seen previously in the search. We believe our approach could be used when performing searches that attempt to determine compilation heuristics [Stephenson et. al. 2003] or effective sequences of optimization phases [Cooper, Schielke and Subramanian 1999; Cooper, Subramanian and Torczon 2002; Almagor et. al. 2004]. While the static estimators used in other search techniques [Wolf et. al. 1996; Triantafyllis et. al. 2003] will always be more efficient in terms of search time, obtaining more accurate performance information via direct execution or simulation may be desirable when tuning high performance kernels or embedded applications where longer compilation times are more likely to be tolerated. Our techniques for producing similar results in fewer generations are designed to work in the context of a search algorithm that randomly probes a subset of the search space in order to remember or predict when a particular phase will not be active. These techniques may be applicable to the Rice approach [Cooper, Schielke and Subramanian 1999; Cooper, Subramanian and Torczon 2002; Almagor et. al. 2004], but may be less useful when applying a search technique that enumerates the different parameters and explicitly prunes various portions of the space [Kisuki et. al. 2000; Whaley et. al. 2001; Triantafyllis et. al. 2003].

3. THE EXPERIMENTAL FRAMEWORK

This section provides a brief overview of the framework used for the experiments reported in this paper. This includes a description of VISTA, the candidate optimization phases, and the test programs used. 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. An optimization phase sequence is a sequence of optimization phases applied by the compiler. This paper describes techniques for achieving faster searches for effective optimization sequences.

VISTA is a low-level interactive compilation system. A more detailed description of VISTA's architecture can be found in prior publications [Zhao et. al. 2002; Kulkarni, Zhao et. al. 2003]. 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 to the program representation.

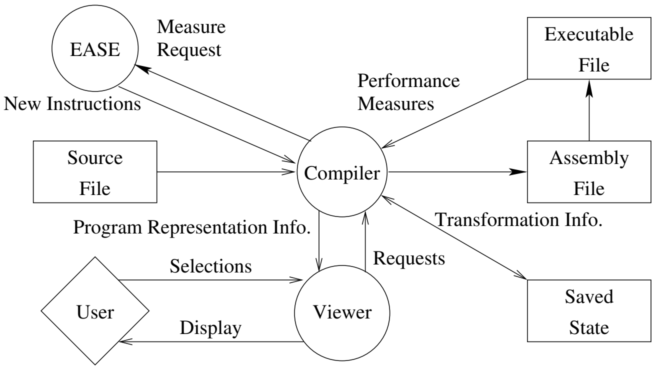

Figure 1: Interactive Code Improvement Process

The interactiveness of VISTA is an important aspect of the environment. A user can view the program representation after each phase or transformation along with performance feedback to gauge the improvement during the tuning process. In addition, a user can manually specify transformations, which is particularly useful when there are architectural features that the compiler cannot exploit. Note that traditional compiler optimization phases can be applied even after manually specifying transformations.

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) [Benitez and Davidson 1988; Benitez and Davidson 1994]. 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. Likewise, we could interface with a simulator to obtain improvements in cycle count and power consumption. 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.

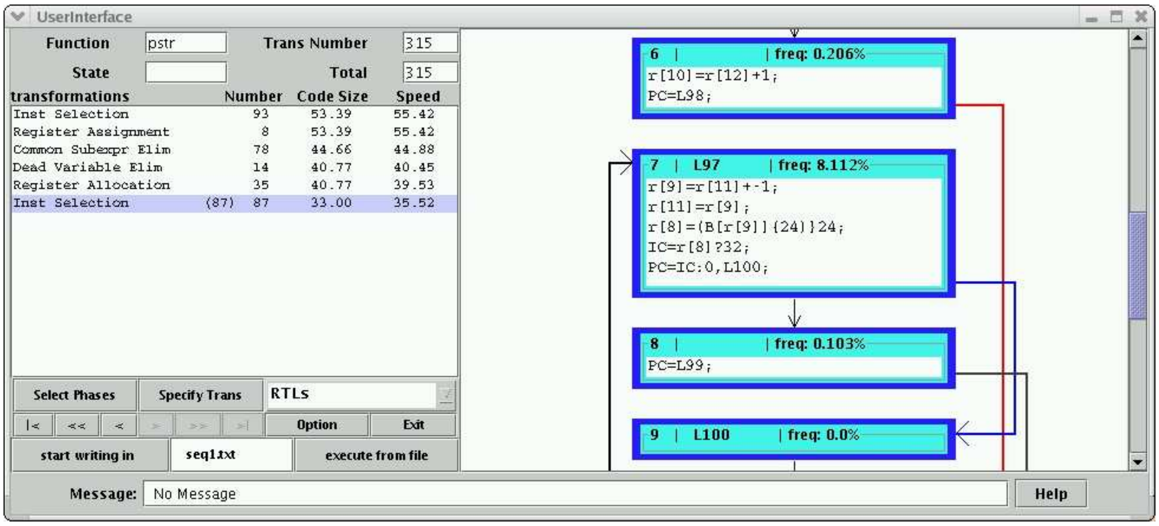

Figure 2: Main Window of VISTA Showing History of Optimization Phases

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 for a given function. Figure 3 shows the different options that VISTA provides 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 [Holland 1989]. A population is the set of solutions (sequences) 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 must be specified, which limits the total number of sequences evaluated. These terms are described in more detail later in the paper. 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.

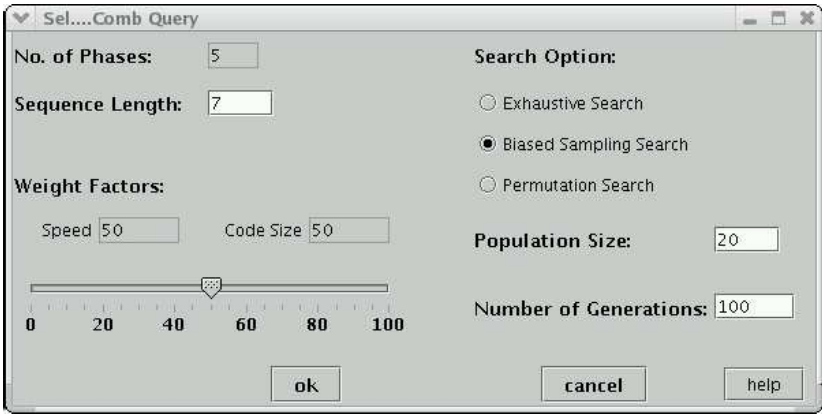

Figure 3: Selecting Options to Search for Possible Sequences

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.

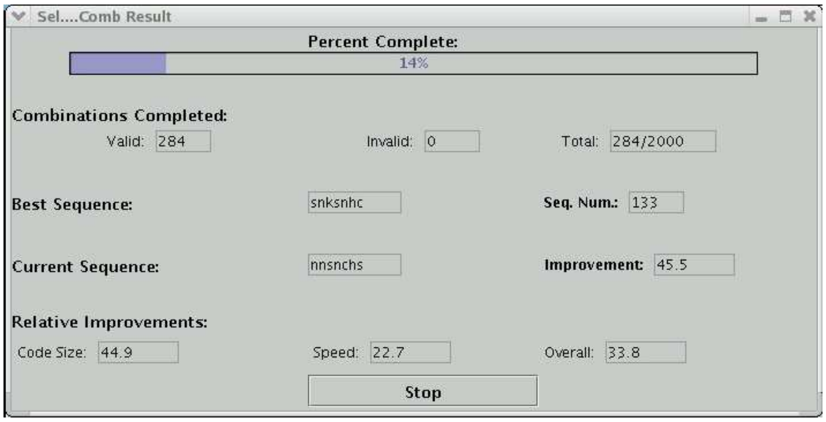

Figure 4: Window Showing the Search Status

Table 1 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, must 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.

Optimization Phase	Gene	Description
branch chaining	b	Replaces a branch or jump target with the target of the last jump in a chain.
common subexpression elimination	c	Performs global analysis to eliminate fully redundant calculations, which also includes global constant and copy propagation.
remove unreachable code	d	Removes basic blocks that cannot be reached from the function entry block.
remove useless blocks	e	Removes empty blocks from the control-flow graph.
dead assignment elimination	h	Uses global analysis to remove assignments when the assigned value is never used.
block reordering	i	Removes a jump by reordering basic blocks when the target of the jump has only a single predecessor.
minimize loop jumps	j	Removes a jump associated with a loop by duplicating a portion of the loop.
register allocation	k	Uses graph coloring to greedily replace references to a variable within a specific live range with a register.
loop transformations	l	Performs loop-invariant code motion, recurrence elimination, loop strength reduction, and induction variable elimination on each loop ordered by nesting level. Each of these optimizations can also be individually selected by the user.
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*.
evaluation order determination	o	Reorders RTLs within a single basic block in an attempt to use fewer registers.
strength reduction	q	Replaces an expensive instruction with one or more cheaper ones. For this version of the compiler, this means changing a multiply by a constant into a series of shifts, adds, and subtracts.
reverse branches	r	Eliminates an unconditional jump by reversing a conditional branch when it branches over the jump.
instruction selection	s	Combines pairs or triples of instructions together where the instructions are linked by set use dependencies. After combining the effects of the instructions, it also performs constant folding and checks if the resulting effect is a legal instruction before committing to the transformation.
remove useless jumps	u	Removes jumps and branches whose target is the following positional block.

Table 1: Candidate Optimization Phases in the Genetic Algorithm Experiments

We used a subset of the *mibench* benchmarks, which are C applications targeting specific areas of the embedded market [Guthaus et. al. 2001]. 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 2 contains descriptions of these programs.

We first perform experiments on an Ultra SPARC III processor so that the results could be obtained in a reasonable time. Thus, the experimental results reported to evaluate the effectiveness of the methods described in the next two sections were obtained for the SPARC. After ensuring that the techniques were sound, we obtained 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.

Our genetic algorithm search for obtaining the baseline measurements was accomplished in the following manner. Past studies using genetic algorithms to generate better code have performed searches on entire applications [Nisbet 1998; Cooper, Schielke and Subramanian 1999; Stephenson et. al. 2003]. In

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 2: MiBench Benchmarks Used in the Experiments

contrast, 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 [Kulkarni, Zhao et. al. 2003]. 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 that gives VISTA an opportunity to apply more active phases than what the batch compiler could accomplish. Note that this length is much less than the number of phases attempted during the batch compilation. The sequence lengths used in these experiments varied between 3 and 50 with an average of 14.15. We set the population size (fixed number of sequences or chromosomes) to 20 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 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, allowed us to obtain baseline measurements within a reasonable period of time, and it has been used in similar studies [Cooper, Schielke and Subramanian 1999; Kulkarni, Zhao et. al. 2003]. 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 (75%) sequences are then candidates for being 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.

Table 3 shows more detailed information about the functions in each of the benchmarks. The unexecuted functions are grouped together as one entry in the four benchmarks where not all of the functions were executed. Associated with each function are the number of basic blocks and instructions before performing any compiler optimizations. The sequence length is the number of optimization phases that are attempted for each sequence during the genetic algorithm run. We set the sequence length to be 1.25 times the number of active phases that were applied for the function by the batch compiler. The final two columns show the percentage of the remaining instructions in code size and dynamic instruction counts obtained by the batch compiler as compared to unoptimized code.

Benchmark	Function	Basic Blocks	Insts	Sequence Length	Remaining Instructions	
					Static	Dynamic
bitcount	AR_btbl_bitcount	2	85	12	23.5%	23.5%
	BW_btbl_bitcount	2	40	8	45.0%	45.0%
	bit_count	7	45	12	26.7%	18.6%
	bit_shifter	8	46	9	36.2%	30.9%
	bitcount	2	101	9	41.6%	41.6%
	main	15	208	24	43.9%	24.4%
	ntbl_bitcnt	4	47	14	34.7%	35.4%
	ntbl_bitcount	2	98	9	40.8%	40.8%
	unexecuted funcs	4.5	58.1	12.3	38.5%	
	average	5.2	80.9	12.1	36.8%	32.5%
dijkstra	dequeue	4	73	10	45.3%	45.3%
	dijkstra	19	284	27	47.3%	39.0%
	enqueue	10	112	14	35.6%	31.3%
	main	14	158	50	47.3%	25.8%
	print_path	4	56	13	46.7%	47.2%
	qcount	2	5	4	66.7%	66.7%
	average	8.8	114.7	19.7	48.2%	42.6%
fft	CheckPointer	4	28	10	58.1%	72.7%
	IsPowerOfTwo	6	31	9	51.5%	40.7%
	NumberOfBitsNeeded	8	51	19	52.7%	30.3%
	ReverseBits	6	49	13	32.0%	26.9%
	fft_float	31	661	30	40.0%	28.7%
	main	33	663	28	40.8%	38.0%
	unexecuted funcs	6	160	13	41.4%	
	average	13.4	234.7	17.4	45.8%	39.6%
jpeg	finish_input_ppm	1	1	3	100.0%	100.0%
	get_raw_row	4	64	12	37.3%	36.2%
	jinit_read_ppm	2	43	10	40.9%	40.9%
	main	25	417	22	44.5%	37.7%
	parse_switches	132	1275	20	41.5%	40.9%
	pbm_getc	13	144	13	27.8%	26.9%
	read_pbm_integer	17	141	15	40.2%	38.0%
	select_file_type	24	244	15	41.3%	42.0%
	start_input_ppm	49	884	27	32.3%	31.5%
	write_stdout	2	10	8	40.0%	40.0%
	unexecuted funcs	11.9	193.3	15.2	35.7%	
	average	25.5	310.5	14.6	43.8%	43.4%
sha	main	10	107	15	43.6%	39.6%
	sha_final	5	126	13	30.3%	27.5%
	sha_init	2	47	7	39.6%	39.6%
	sha_print	2	42	10	32.6%	32.6%
	sha_stream	5	70	14	54.7%	27.0%
	sha_transform	20	504	30	32.5%	28.0%
	sha_update	7	124	18	31.8%	32.5%
	average	7.3	145.7	15.3	37.9%	32.4%
stringsearch	init_search	10	87	24	53.3%	26.6%
	main	18	233	24	43.4%	30.5%
	strsearch	13	126	22	41.4%	38.3%
	unexecuted funcs	14	161	25	46.1%	
	average	13.8	151.8	23.8	46.0%	31.8%
average		12.3	173.0	17.1	43.1%	37.0%

Table 3: Function Information for the MiBench Benchmarks Used in the Experiments

Figures 5, 6, and 7 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 paper. 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 [Kulkarni, Zhao et. al. 2003]. The average benefits shown in the figure are slightly improved from previously published results since the searches now include additional optimization phases that were not previously exploited by the genetic algorithm [Kulkarni, Zhao et. al. 2003]. Note that the contribution of this paper is that the search for these benefits is more efficient, rather than the actual benefits obtained.

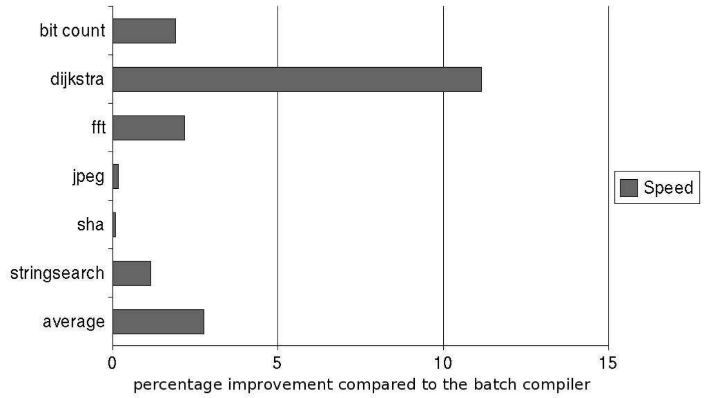

Figure 5: Speed Only Improvements for the SPARC

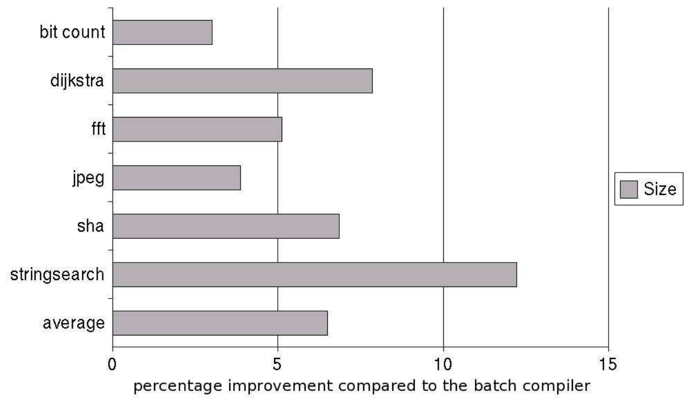

Figure 6: Size Only Improvements for the SPARC

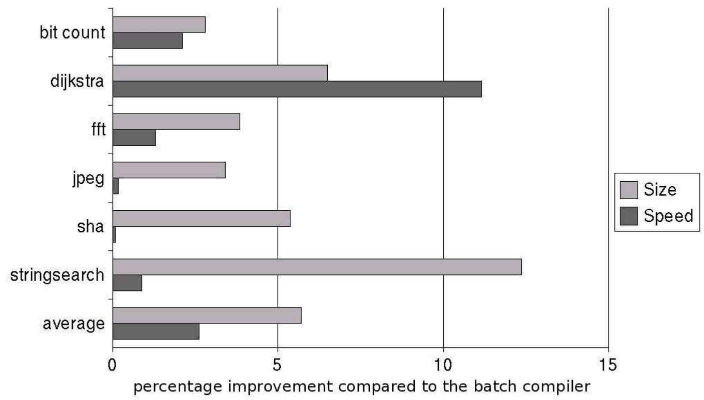

Figure 7: Size and Speed Improvements for the SPARC

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 instead of simulation to evaluate performance. 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. The following subsections describe methods to reduce the search overhead and the results of applying these methods.

4.1 Methods for Reducing the Search Overhead

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) To obtain performance measurements, the program is executed using a command in a configuration file, which may involve direct execution or simulation. (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.[1] 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 8 illustrates the order in which the different methods are attempted. Each optimization phase sequence generated by the genetic algorithm is checked by up to four methods. 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. The first method checks if the attempted sequence has been previously encountered for the function. If so, then the compilation by applying these phases is avoided. The second, third, and fourth methods are used to avoid the evaluation of the function, which comprise tasks 2-6 described earlier.

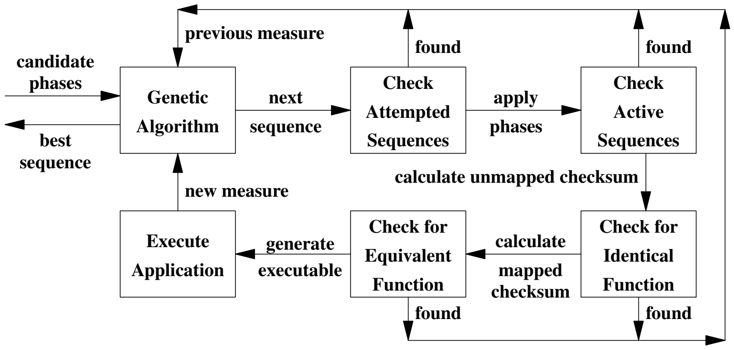

Figure 8: Methods for Reducing Search Overhead

[1] It is possible that a new optimization sequence can cause the generated code to produce incorrect output. In the rare case when this happens, we assign a poor fitness value to the sequence so that it will not be selected by the genetic algorithm.

4.1.1 Finding Redundant Attempted Sequences

Sometimes the same optimization phase sequence is reattempted during the search. Consider Figure 9, 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 prior attempted solutions has been previously used to reduce search time [Cooper, Schielke and Subramanian 1999; Stephenson et. al. 2003; Kulkarni, Zhao et. al. 2003].

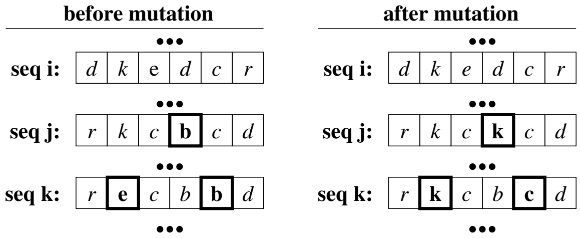

Figure 9: Example of Redundant Attempted Sequences

We realized that different sequences with the same attempted phases may generate the same code since some optimization phases are independent in that the order in which they are performed cannot affect the final code that is being generated. For instance, consider applying branch chaining before and after register allocation. Branch chaining does not change the live range of any variable that is a candidate for register allocation. Likewise, register allocation does not affect branch chaining since it does not affect conditional branches or unconditional jumps. Both branch chaining and register allocation will neither inhibit nor enable the other phase. Therefore, we identified for each optimization phase whether or not it is independent with each of the other phases. Rather than directly using the attempted sequence in the hash, we instead first sort the phases within the sequence so that two consecutively applied phases that are independent are always performed in the same order. We then use the sorted sequence of phases when accessing the hash table. Using the sorted sequence will allow more redundant sequences to be detected so more compilations can be avoided. The *x* entries in Table 4 indicate which optimizations are independent of one another in the VPO compiler. For instance, branch chaining (*b*) is independent of register allocation (*k*).

We used our experience and insight in deriving the information in this table indicating which optimization phases are independent of one another. We inserted sanity checks when running our experiments to ensure that this information was correct. We were surprised that our initial reasoning was often incorrect and corrected the independence information. This process is described in more detail in the Section 7, which appears later in the paper.

4.1.2 Finding Redundant Active Sequences

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. In other words, a dormant phase does not apply any transformations. 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 10 illustrates how different attempted sequences can map to the same active sequence, where the bold boxes represent active phases and the

Optimization Phase	Gene	b	c	d	e	h	i	j	k	l	m	o	q	r	s	u
branch chaining	b	x			x				x	x		x	x		x	
comm subexpr elim	c		x				x							x		x
remv unreach code	d			x		x		x				x	x		x	x
remv useless blocks	e	x			x			x	x		x	x	x		x	
dead asg elim	h			x		x	x	x			x		x			x
block reordering	i		x			x	x		x	x		x	x		x	
min loop jumps	j			x	x	x		x				x	x		x	
register allocation	k	x			x		x		x					x		x
loop trans	l	x					x			x		x				x
merge basic blocks	m				x	x					x		x			
eval order determ	o	x		x	x		x	x		x		x		x		x
strength reduction	q	x		x	x	x	x	x			x		x	x		x
reverse branches	r		x						x			x	x	x	x	
instruction selection	s	x		x	x		x	x						x	x	x
remv useless jumps	u		x	x		x			x	x		x	x		x	x

***x* indicates if the two phases are independent**

Table 4: Independent Optimization Phases

nonbold boxes represent dormant phases. A second hash table is used to record sequences where only the active phases are represented. As when accessing the attempted hash table, we also sort the phases in the active sequence so that two consecutive independent phases are always applied in the same order.

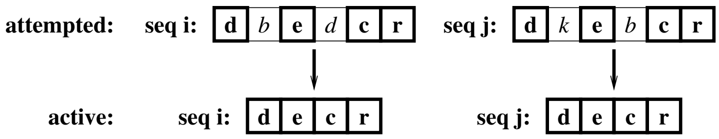

Figure 10: Example of a Redundant Active Sequence

4.1.3 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 11 can be merged together. Instruction selection symbolically merges the instructions

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 11: Different Optimizations Having the Same Effect

and checks to see if the resulting instruction is legal. The same effect in this case can be produced by constant propagation (actually part of common subexpression elimination in VPO) followed by dead assignment elimination.

We also found that while some optimization phases are not independent, the order in which they are applied often do not affect the generated code. For instance, branch chaining causes a transfer of control to go directly to the end of a chain of unconditional jumps. It is possible that one of those unconditional jumps in the chain can become unreachable code after performing branch chaining. However, this is unlikely to happen.

VISTA has to efficiently detect when different active sequences generate identical code to be able to lower the search overhead. A search may result in thousands of unique function instances, which may be too large to store in memory and be very expensive to access on disk. The key realization in addressing this issue was that while VISTA needs to detect when function instances are identical, it 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, VISTA calculates a CRC (cyclic redundancy code) checksum on the bytes of the RTLs and keeps 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 [Peterson and Brown 1961]. If the checksum has been generated for a previous function instance, then the performance results of that instance are used. We have verified that it is rare that the same checksum is generated for different function instances and we never observed that the best fitness value found was affected in our experiments.

4.1.4 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 situation can occur since different optimization phases compete for registers. For instance, consider the source code in Figure 12(a). Figures 12(b) and 12(c) show two possible translations given two different orderings of optimization phases that consume registers.

```
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 12: Different Functions with Equivalent Code

To detect this situation, VISTA identifies the live ranges of all of the registers in the function and maps each live range to a distinct pseudo register. Equivalent function instances become identical after mapping, which is illustrated for the example in Figure 12(d). The CRC checksum for the mapped function instance is computed and checked in a separate hash table of CRC checksums to see if the mapped function had been previously generated.

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 regarding a uniform register access time and no stalls due to anti or output dependences are true, if 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.

4.2 Experimental Results

We applied the techniques in Section 4.1 to each of the benchmarks shown in Table 2. Again we used a population size of 20 and 100 generations when attempting to find an effective optimization sequence using the genetic algorithm once for each function. Thus, 2000 optimization phase sequences are generated for each function.

Figure 13 shows the average number of sequences whose executions were avoided for each benchmark using the four different methods described in Section 4. Each function is weighted equally since the same number of sequences were applied for each function. The average bar is for the average of the percentages for the six benchmarks. 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. 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 38.2% of the sequences were detected as redundantly attempted using the technique in Section 4.1.1. 36.6% were caught as redundant active sequences using the technique in Section 4.1.2. 10.5% were discovered to produce identical code as generated by a previous sequence using the technique in Section 4.1.3 and 2.5% were found to produce unique, but equivalent code using the technique in Section 4.1.4. Thus, over 87.7% of the executions were avoided. We discovered that sorting the phases in a sequence, so that consecutively applied

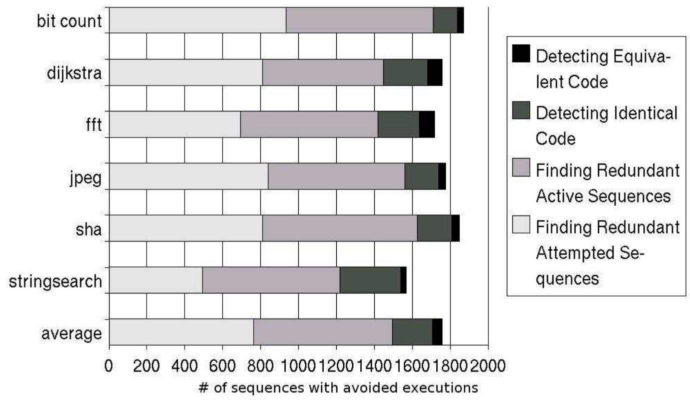

Figure 13: Number of Avoided Executions

independent phases are in the same order, increased the number of avoided executions by 1.15%. We found that sorting was more successful when hashing the active sequences than the attempted sequences since there was a greater chance of having a redundant sequence due to the sequence lengths being shorter after removing the dormant phases.[2]

Figure 14 shows the relative search time required when applying the methods described in Section 4 to not applying these methods. These methods reduced the search time by 62%. The average time required to evaluate each of the six benchmarks improved from 6.31 hours to 2.86 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.

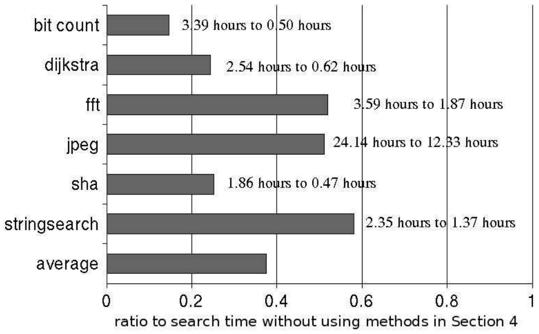

Figure 14: Relative Total Search Time on the SPARC

By observing the search status, as shown in Figure 4, we found that search progressed more quickly as the number of generations performed increased. Figure 15 shows the average number of redundant sequences, where execution was not required, for each of the 100 generations in the searches. The average number of redundant sequences generally increases as more generations are performed. This phenomenon is not surprising since there is a limited number of sequences that will produce different code. Thus, a user can double the number of generations to be performed with only a small increase in search time. Likewise, we could check for improvement for the last *n* generations and used this as a termination condition for the genetic algorithm.

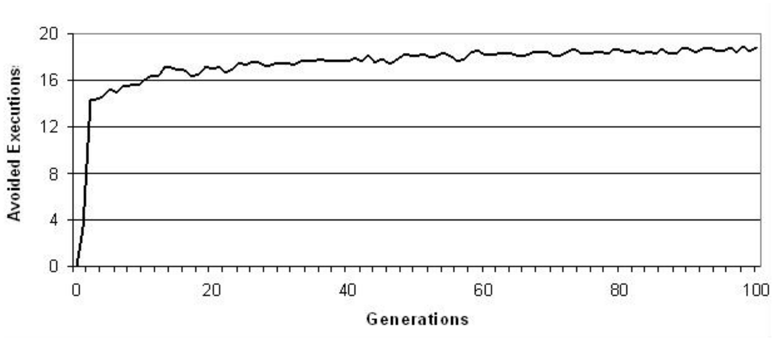

Figure 15: Number of Redundant Executions Avoided Per Generation

[2] The results presented in Figure 13 are slightly different than the results presented in a previous version of this paper [Kulkarni, Hines et. al. 2004]. The reasons for these differences include changes to the compiler, sorting of the independent phases within the attempted and active sequences before accessing the hash tables, and enhancements for detecting identical and equivalent functions.

We also found that searches performed with shorter sequences had a higher percentage of redundant executions that could be avoided. Note that the sequence length is established by the batch compilation. Smaller functions tended to have shorter sequence lengths due to fewer opportunities for optimization phases to be active. Figure 16 shows three plots with sequence lengths ranging from 3-10, 11-20, and 21-50. The shorter sequence lengths quickly become almost entirely redundant in a few generations. A sequence that has a shorter length is more likely to be redundant due to fewer active phases affecting the generated code. In addition, the likelihood of mutation is less when there are fewer phases in a sequence to mutate. In contrast, the longer sequences are on average much less redundant since longer sequence lengths yield more possible active sequences and more possible ways in which the final code can be generated. All three plots show that the search finds an increasing number of redundant sequences as the number of generations increases.

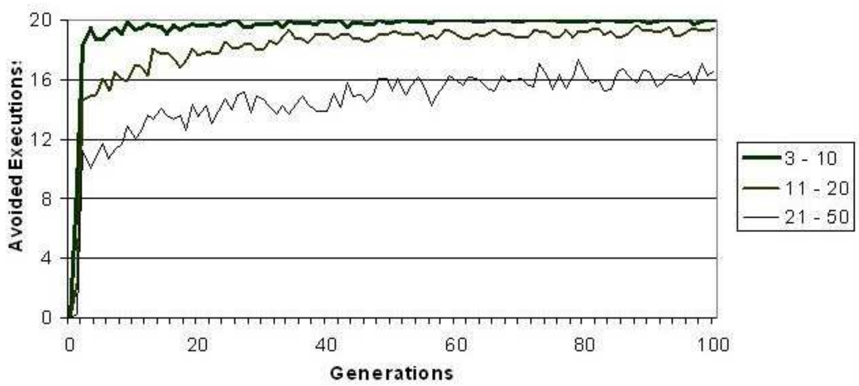

Figure 16: Number of Redundant Executions Avoided Per Generation for Different Sequence Lengths

Figures 17, 18, and 19 display information regarding the number of times an optimization phase was active. Figure 17 shows the average number of times that the different optimization phases were active for each sequence. One should realize that an optimization phase may not be active in a sequence since the genetic algorithm may simply not select that particular phase throughout the sequence. Also, this information does not depict the number of transformations that were applied in each active phase. However, the figure does illustrate that some optimization phases, such as instruction selection and common subexpression elimination, are much more likely to be active than other phases. In addition, some phases can be accomplished by a combination of other phases. For instance, common subexpression elimination and dead assignment elimination can often have the same effect as instruction selection. Finally, the success of phases is also affected by the code generation strategy. For instance, the front end that we used always generated intermediate code where a label preceded the epilogue code at the end of a function in case there were return statements in the source code from other locations in the function. For functions with no conditional control flow, this return block was always merged with the entry block. Thus, the *merge basic blocks* optimization phase was successful more frequently than if another code generation strategy was used.

Figure 18 shows how often an optimization phase will be active given that it was actually attempted. It is interesting to note that while instruction selection was the phase that was active the most often, common subexpression elimination was active a greater percentage of the time when it was selected. Instruction selection has a direct impact on both code size and speed. Sometimes common subexpression elimination does not reduce code size and may not be deemed as beneficial as instruction selection by the genetic algorithm. Likewise, evaluation order determination could often be applied successfully when attempted, but had little impact on performance. The phases that did not help performance are likely to be in sequences that are in the lower half of the population. These sequences could be replaced by the crossover operation and had a higher mutation rate applied to them. Thus, phases having little impact on performance

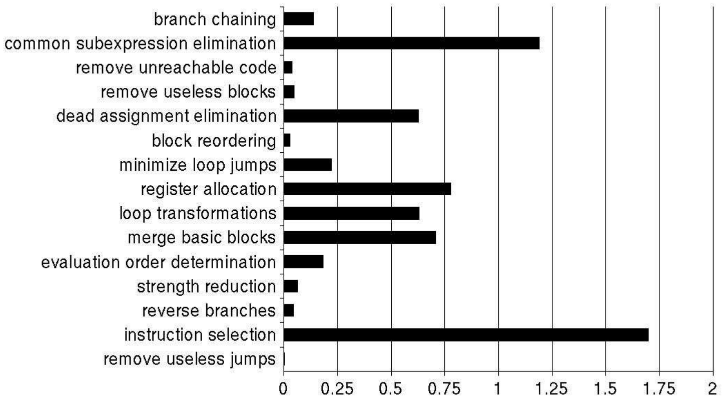

Figure 17: Average Times Each Phase Was Active

were applied less often. In addition, evaluation order determination could only be applied before assigning pseudo registers to hardware registers, which was implicitly performed before the first code-improving phase in the sequence that requires it.

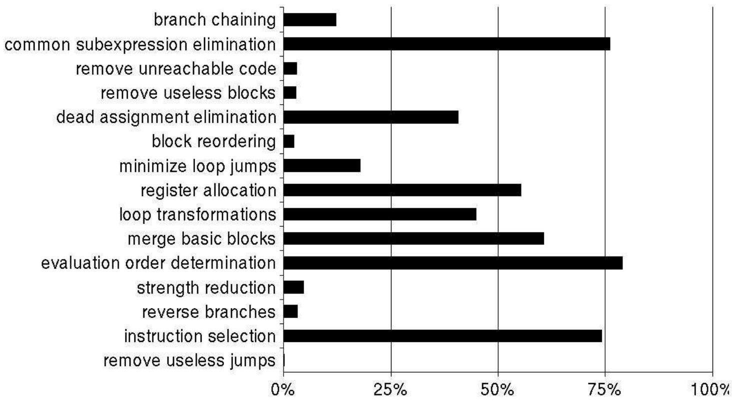

Figure 18: Percentage That Each Phase Was Active When Attempted

Figure 19 shows the average number of times an optimization phase was active in a sequence given that it was active at least once. There are several optimization phases, such as branch chaining, that were active at most a single time. This shows that perhaps these phases are typically not enabled by other phases.

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.

5.1 Methods for Producing Similar Results in Fewer Generations

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.

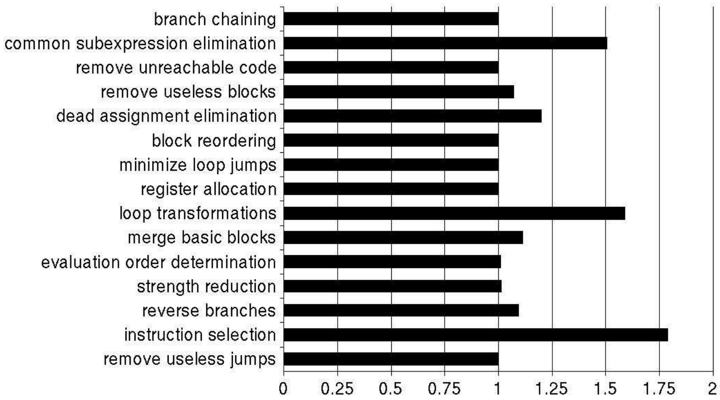

Figure 19: Number of Times Each Phase Was Active Given It Was Active at Least Once

5.1.1 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 [Kulkarni, Zhao et. al. 2003].

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 [Stephenson et. al. 2003].

5.1.2 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 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.

5.1.3 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, VISTA represents the active phases as nodes in a DAG, where each child corresponds to the next phase in an active sequence. For each node VISTA calculates the CRC checksum for the bytes of the RTLs at that point after applying the associated optimization phase. A node in the DAG has more than one parent when different prefixes produce identical RTLs. We also store at each node the set of phases that were found to be dormant for that prefix of active phases. Figure 20 shows an example DAG where the bold portions represent active prefixes and the nonbold boxes represent dormant phases given that prefix. The genetic algorithm finds that the prefixes **bcb** and **be** produce identical code. At that point the algorithm merges the prefixes so that they both point to the same **f** node in the DAG. For instance, *a* and *f* are dormant phases for the prefix **bac**. To prohibit applying a prior dormant phase, VISTA forces 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.

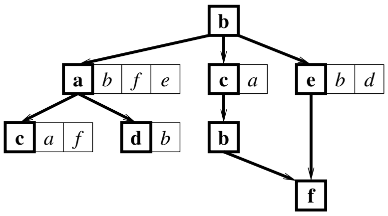

Figure 20: A DAG Representing Active Prefixes

5.1.4 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 21 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.

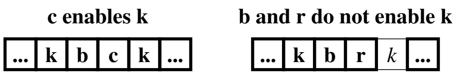

Figure 21: Enabling Previously Applied Phases

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 DAG of active prefixes, which was described in the previous subsection, to determine which phases are currently enabled. For example, consider again Figure 20. 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 DAG we track which phases cannot be enabled by just examining the subsequently attempted phases.

5.2 Experimental Results

In this section we determined the average number of generations that were evaluated for each of the functions before finding the best fitness value in the search.[3] The *baseline* result is without using any of the

[3] The results after applying the techniques in Section 5 also changed slightly from the results that were presented in a previous version of this paper [Kulkarni, Hines et. al. 2004]. These differences were due to not only changes in the compiler, but also to using a DAG instead of a tree, where a checksum is stored with each node, so that more redundant active prefixes can be detected.

techniques described in Section 5.1. The other results indicate the generation when the first sequence was found whose performance equaled the best sequence found in the baseline search. We did not include the results for the functions when the best fitness value found was not as good as the best fitness value in the baseline, which occurred on about 3% of the functions. Not including these results caused the baseline to vary since the functions with different fitness values were not always the same when applying each of the techniques. About 9.4% of the functions had improved fitness values and about 2.8% of the functions had worse fitness values when *all* of the techniques were applied. On average the best fitness values improved by 0.04% (by 0.30% for only the differing functions). The maximum number of generations before finding the best fitness value for any function was 98 out of a possible 100 when not applying any of the four techniques. The maximum was 89 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 22 shows the effect of using the batch sequence in the initial population, which in general was quite beneficial. The last three bars show the average effect when separating the benchmarks according to the sequence length used in the search. Note that sequence length for each function is established by multiplying the active sequence of the batch compiler by 1.25. 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.

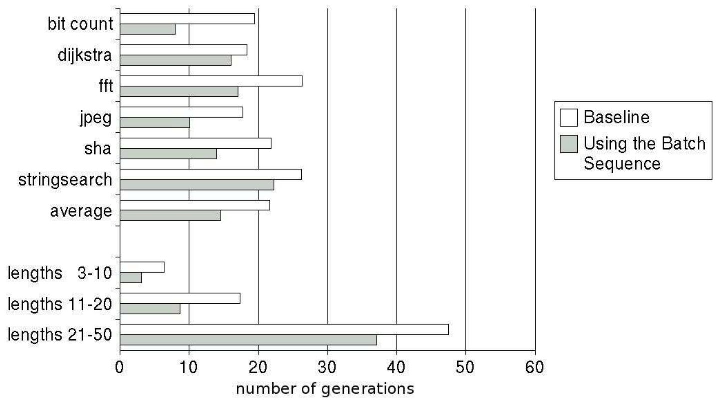

Figure 22: Number of Generations before Finding the Best Fitness Value When Using the Batch Sequence

The effect of prohibiting specific phases throughout the search was less beneficial, as shown in Figure 23. 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.

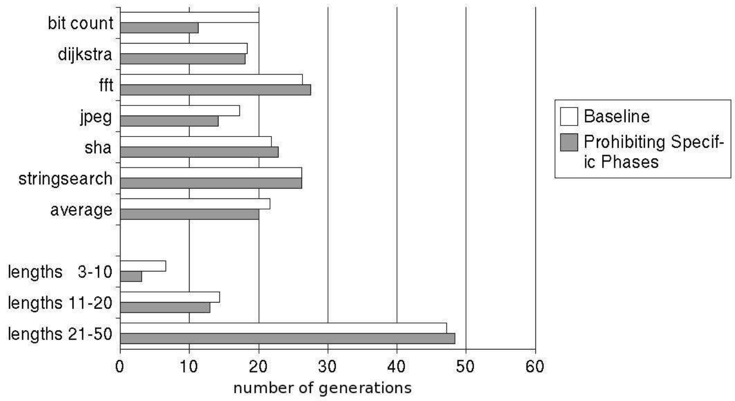

Figure 23: Number of Generations before Finding the Best Fitness Value When Prohibiting Specific Phases

Figure 24 shows how often each type of phase could be prohibited. Several transfer of control optimization phases could be prohibited when the function had no such instructions. Minimize loop jumps and loop transformations could be prohibited when there were no loops in a function. Register allocation could be prohibited for only very simple functions that referenced no local variables or arguments. Several optimization phases were never prohibited since these phases could either be commonly performed or the analysis to determine they could not be applied was difficult to accomplish.

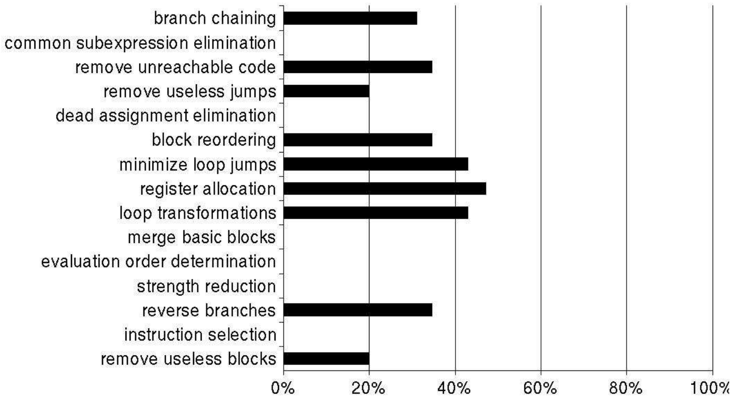

Figure 24: Percentage of Functions Where Each Phase Could be Prohibited

In contrast, prohibiting prior dormant and unenabled phases, which are depicted in Figures 25 and 26, 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 27. In fact, only about 41% of the generations on average (from 21.38 generations to 8.85 generations) 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.

Consider Figure 28, which depicts the number of avoided executions. The top bar shows the results given in Figure 13 from applying only Section 4 techniques. 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. No

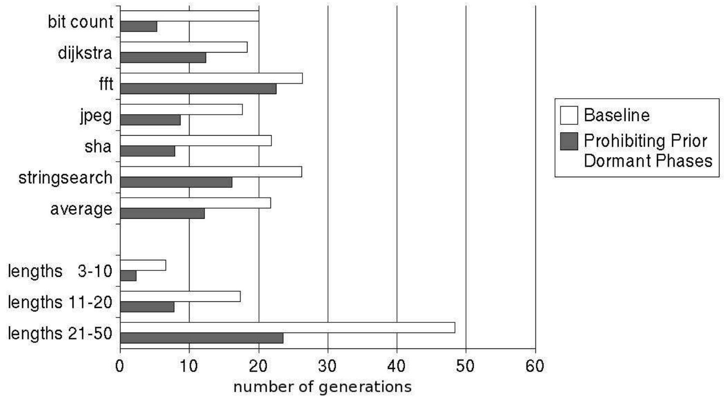

Figure 25: Number of Generations before Finding the Best Fitness Value When Prohibiting Prior Dormant Phases

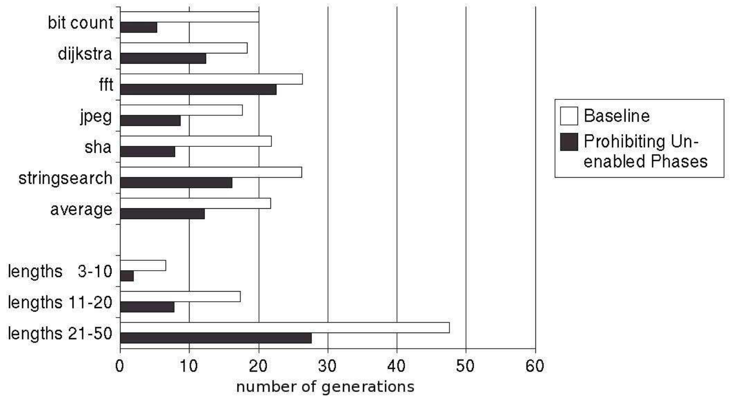

Figure 26: Number of Generations before Finding the Best Fitness Value When Prohibiting Unenabled Phases

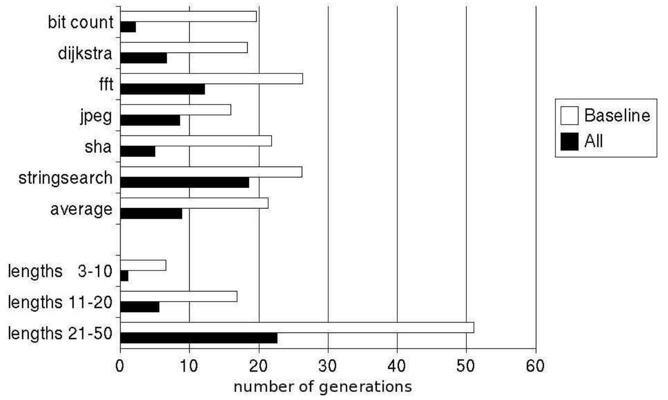

Figure 27: Number of Generations before Finding the Best Fitness Value When Applying All Techniques

active sequences were considered redundant after applying the technique described in Section 5.1.3 since we checked the checksums stored in the DAG of active prefixes to determine if the active sequences produced identical code. Thus, detecting sequences as identical also detects redundant active sequences. One can see that the number of redundantly attempted sequences decreased on average. We found that many of the smaller functions had more hash table hits for attempted sequences after applying the techniques in Section 5 and the larger functions typically had fewer hits. We believe this phenomenon is due to applying the techniques to prohibit prior dormant and unenabled phases. For the smaller functions with shorter sequence lengths, the possible phases to attempt were often exhausted and an active phase that was used before was often attempted. Likewise, the larger functions with longer sequence lengths and significantly larger search spaces tended to not reattempt previously dormant phases, but did not exhaust the possible phases and had fewer hits in the hash table. The average number of avoided executions decreases by about 1.4%, 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 27.

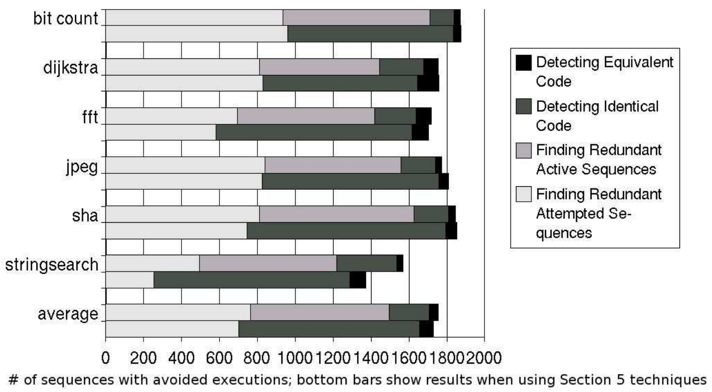

Figure 28: Number of Avoided Executions When Using Section 5 Techniques

Figure 29 shows the impact that applying all of the techniques in Section 5 had on the average performance of the code for each generation relative to the best fitness value found in the search. A significant improvement is obtained by performing the batch sequence in the initial generation. After a few generations, prohibiting prior dormant phases and prohibiting unenabled phases result in a greater benefit than using the batch sequence. Performing all of the techniques resulted in the best result. This graph shows that the number of generations could be reduced with a neglible loss in performance of the generated code.

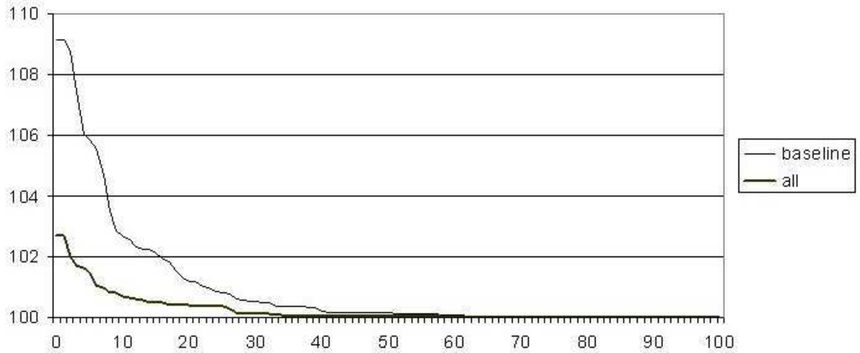

Figure 29: Average Benefit Relative to the Best Fitness Value Per Generation

Figure 30 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 65.0% of the time on average as compared to the baseline.

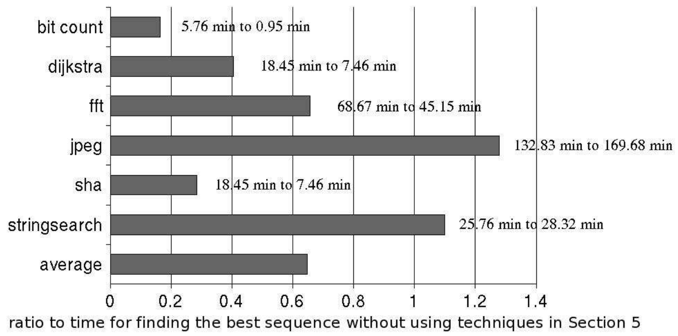

Figure 30: Relative Search Time before Finding the Best Fitness Value

6. APPLYING THE TECHNIQUES ON AN EMBEDDED PROCESSOR

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 31, 32, and 33 show the percentage improvement when optimizing for speed only, size only, and 50% for each factor, respectively.

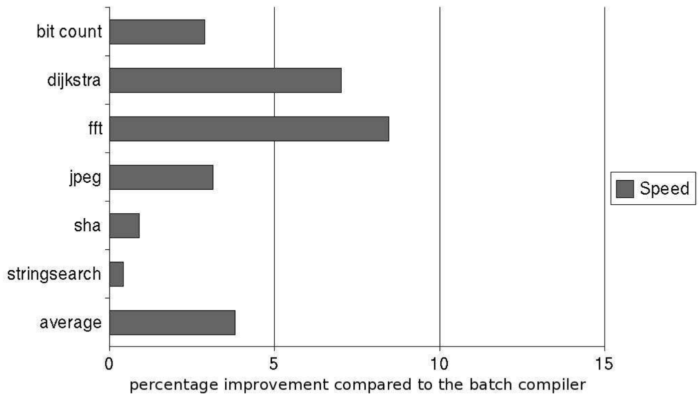

Figure 31: Speed Only Improvements for the ARM

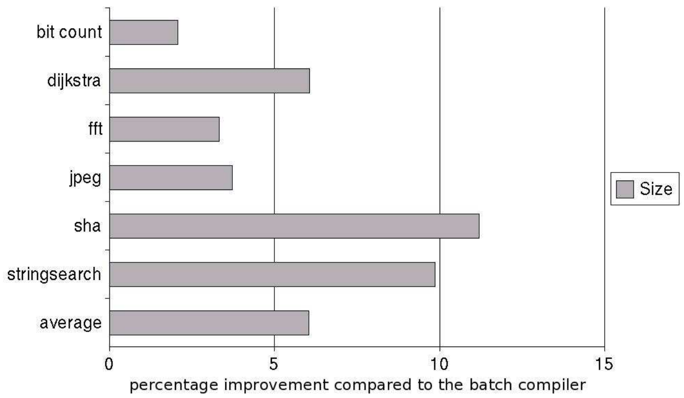

Figure 32: Size Only Improvements for the ARM

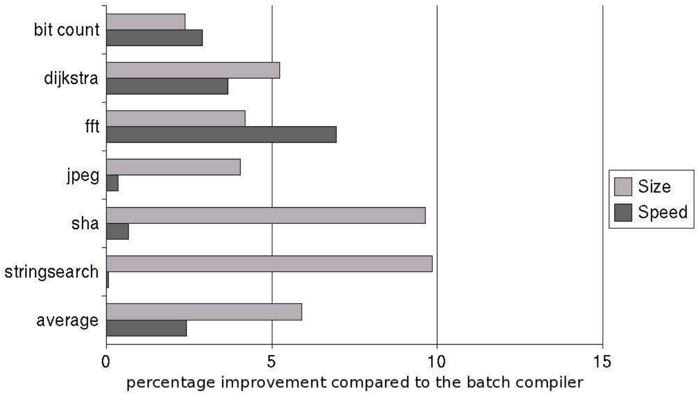

Figure 33: Size and Speed Improvements for the ARM

Figure 34 shows the relative time for running the genetic algorithm on the ARM when all of the techniques in Section 4 were applied. The search time using the Section 4 techniques required 35.9% of the time on average as compared to not applying these techniques. The average time required to obtain results for each of the benchmarks when optimizing for both speed and size on the ARM required 11.54 hours instead of 26.68 hours.

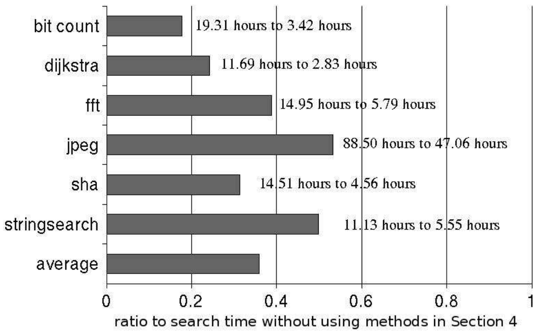

Figure 34: Relative Total Search Time on the ARM

7. IMPLEMENTATION ISSUES

During the process of this investigation, we encountered several implementation issues that made this work challenging. First, the VISTA framework was designed so that a user could interactively make selections using a mouse. We setup a mode in VISTA where selections could be specified in a file so that the experiments could be performed in a batch mode. Second, 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. Third, determining which phases were independent (see Table 4), prohibiting specific phases (see Section 5.1.2), and prohibiting unenabled phases (see Section 5.1.4) required analysis and judgement by the compiler writer to determine when optimization phases could be enabled or disabled. We inserted sanity checks when running experiments without using these methods to ensure that our assertions concerning the enabling of optimization phases were accurate. For instance, we checked that the attempted and active sequences for every function produced the same code when applied directly or when applied after sorting the independent phases. 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. Fourth, we sometimes found that dormant optimization phases did have unexpected side effects by changing the analysis information, which could enable or disable a subsequent optimization phase. These side effects can affect the results of the methods described in Sections 4.1.2, 5.1.3, and 5.1.4. 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. Finally, these experiments were quite time-consuming, particularly when obtaining a baseline without using our techniques to reduce the search overhead. We modified the system to log information during the search, such as each attempted sequence, the corresponding active sequence, the checksum of the function produced by the sequence, and the effect on speed and space. In order to reduce the time required to isolate problems when performing various sanity checks, we would process the log file rather than rerunning the entire search.

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 [Stephenson et. al. 2003]. 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 that 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 many helpful suggestions that improved the quality of the paper. In particular, we thank Keith Cooper and Tim Harvey for their insightful comments. This research was supported in part by National Science Foundation grants EIA-0072043, ACI-0203956, CCR-0208892, ACI-0305144, and CCR-0312493.

REFERENCES

Almagor, L., Cooper, K., Grosul, A. and Harvey, T., "Finding Effective Compilation Sequences," *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 231-239 (July 2004).

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

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

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

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

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

Cooper, K., Subramanian, D. and Torczon, L., "Adaptive Optimizing Compilers for the 21st Century," *Journal of Supercomputing* **23**(1) pp. 7-22 (2002).

Gao, G., Olsen, R., Sarkar, V. and Thekkath, R., "Collective Loop Fusion for Array Contraction," *Workshop on Languages and Compilers for Parallel Computing*, pp. 281-295 (1992).

Granlund, T. and Kenner, R., "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).

Granston, E. and Holler, A., "Automatic Recommendation of Compiler Options," *Proceedings of the 4th Workshop on Feedback Directed Optimization*, (December 2001).

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

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

Irigoin, F. and Triolet, R., "Supernode Partitioning," *Symposium on Principles of Programming Languages*, pp. 319-329 (1988).

Kisuki, T., Knijnenburg, P. and O'Boyle, M., "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).

Kulkarni, P., Hines, S., Hiser, J., Whalley, D., Davidson, J. and Jones, D., "Fast Searches for Effective Optimization Phase Sequences," *Proceedings of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, pp. 171-182 (June 2004).

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

Massalin, H., "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).

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

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

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

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

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

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

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

Wolf, M., Maydan, D. and Chen, D., "Combining Loop Transformations Considering Caches and Scheduling," *Proceedings of the International Symposium on Microarchitecture*, pp. 274-286 (December 1996).

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