Improving Both the Performance Benefits and Speed of Optimization Phase Sequence Searches

Prasad A. Kulkarni Michael R. Jantz
University of Kansas
Department of Electrical Engineering and Computer Science, Lawrence, KS 66045
{prasadk,mikejant}@ku.edu

David B. Whalley
Florida State University
Computer Science Department
Tallahassee, Florida 32306
whalley@cs.fsu.edu

Abstract

The issues of compiler optimization phase ordering and selection present important challenges to compiler developers in several domains, and in particular to the speed, code size, power, and cost-constrained domain of embedded systems. Different sequences of optimization phases have been observed to provide the best performance for different applications. Compiler writers and embedded systems developers have recently addressed this problem by conducting iterative empirical searches using machine-learning based heuristic algorithms in an attempt to find the phase sequences that are most effective for each application. Such searches are generally performed at the program level, although a few studies have been performed at the function level. The finer granularity of function-level searches has the potential to provide greater overall performance benefits, but only at the cost of slower searches caused by a greater number of performance evaluations that often require expensive program simulations. In this paper, we evaluate the performance benefits and search time increases of function-level approaches as compared to their program-level counterparts. We, then, present a novel search algorithm that conducts distinct function-level searches simultaneously, but requires only a single program simulation for evaluating the performance of potentially unique sequences for each function. Thus, our new *hybrid* search strategy provides the enhanced performance benefits of function-level searches with a search-time cost that is comparable to or less than program-level searches.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors- compilers, optimization

General Terms Performance, Measurements, Algorithms.

Keywords Phase Ordering, Genetic Algorithms.

1. Introduction

The optimization phase ordering and selection problems have been a long-standing and persistent dilemma for compiler writers [12, 25, 29]. The two problems are related in that phase ordering tries to find the best order in which optimization phases should be applied and phase selection focuses on determining whether or not phases should be applied. Optimization phases depend on machine resources (such as registers) and transformations performed by other phases for their successful application. Consequently, phases depend on and interact with each other, enabling and disabling opportunities for other optimization phases. It is widely recognized, and often reported, that a single sequence of optimization phases is unlikely to achieve the best performance for every application on a given machine [8, 17, 19, 28, 30]. Instead, the ideal phase sequence depends on the characteristics of the application being compiled, the compiler implementation, and the target architecture.

Since it is difficult to predict the optimal phase sequence for each application, compiler writers have recently investigated phase ordering/selection by employing genetic algorithms [8, 19] and other evolutionary techniques [1, 2, 15, 16, 24] during iterative compilation to search for the the most effective sequence. When the fitness criteria for such searches involves dynamic measures (e.g., cycle counts or power consumption), thousands of direct executions of an application may be required. The search time in such cases is significant, often needing hours or days for finding effective sequences for a single application, making it less attractive for developers.

However, there are application areas where long compilation times are acceptable. For example, such iterative search techniques are often suitable for compiling programs targeting embedded systems. Many embedded system developers attempt to construct systems with just *enough* compute power and memory as is necessary for the particular task. Most embedded systems are also constrained for power. Consequently, reducing the speed, memory (code size), and/or power requirements is extremely crucial for such embedded applications, as reducing the processor or memory cost can result in huge savings for products with millions of units shipped.

The search time problem is unfortunately further exacerbated for embedded systems because the software development environment is often different from the target environment. Obtaining performance measures on cross-platform development environments typically requires simulation. The advantages of using simulation include obtaining repeatable and more detailed information about the program's execution. For instance, simulation can easily provide performance information about each function in a program. Furthermore, simulators for embedded processors are in general much more accurate than those for general-purpose processors since embedded processors are inherently simpler. However, simulation 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. Searching for an effective optimization sequence in such environments can easily require significantly longer periods than even the hours or days reported when using direct execution on a general-purpose processor [2, 4]. Thus, reducing the search time while achieving the intended benefits is critical to make evolutionary searches feasible for embedded systems.

Iterative searches to find the most effective phase sequence can be performed at multiple levels of code granularity: (1) Typically, such searches are performed at the *program-level*, with the same set of optimization flags or phase sequence employed for the entire program [1, 8, 14, 16]. (2) *Function-level* searches attempt to find possibly distinct phase sequences that are most effective over individual functions at a time [19, 28]. Function-level searches are more expensive as they may require several times more program simulations/executions, depending on the number of functions in the program. For example, a simple genetic algorithm based function-level search over a program consisting of n functions, will require n times more program simulations/executions. However, by potentially achieving enhanced customization, function-level searches may result in more efficient executables.

In this paper, we study and quantify the potential benefits (in terms of execution cycles and code size) and relative costs (in term of search overhead) of function versus program-level evolutionary searches. We suggest a new *hybrid* search strategy that combines the best features of function and program level search techniques. Our approach works by performing multiple function-level searches over all functions in the program simultaneously, but does not simulate the program until each function in the program has a new sequence to evaluate. We show that our hybrid search strategy achieves the performance benefits of a function-level search, but with less cost than is required for even a program-level search. Thus, the main contributions of this paper are:

1. This is the first study of which we are aware to compare the costs and benefits of performing searches for effective optimization phase sequences for individual functions versus an entire program as a single search unit.
2. We also introduce and evaluate a new *file*-level search strategy, which is the finest search granularity that can be achieved by algorithms implemented outside most compilers.
3. We describe and show the results for a hybrid search strategy that produces code with the effectiveness of individual function-level searches and has search costs that match or are more efficient than program-level search costs.

The rest of the paper is organized as follows. In the next section, we discuss previous research related to automatically finding effective phase sequences in optimizing compilers. In Section 3, we describe the three search strategies being compared in this paper, individual function, file, and entire program searches. We then outline our compiler framework and experimental setup in Section 4. In Section 5, we present the configuration of the genetic algorithm based evolutionary search technique used for the searches compared in this paper. We give the results of the comparisons of function, file, and program level searches in Section 6. We present and evaluate our new hybrid search technique in Section 7. Finally, we detail our thoughts regarding future work and conclusions in Sections 8 and 9, respectively.

2. Related Work

Prior work in optimization phase ordering and selection has investigated both analytical and empirical approaches to address the problem. Specifications of code-improving transformations have been automatically analyzed to determine their enabling and disabling interactions [30], as well as other phase properties such as the impact [31] and profitability [32] of optimizations. Such analytical information can provide insight into constructing a single *compromise* phase ordering for a conventional optimizing compiler.

The empirical search community acknowledges that it is unlikely that a single sequence of phases will achieve the best performance for all programs, and instead employs empirical techniques to iteratively search for the most effective phase sequence over programs or individual functions [1, 4, 5, 7, 8, 16, 18–20, 28]. Exhaustive enumeration of the phase application search space, although feasible in some cases [21, 22], has been reported to take many days to several months, depending on the compiler, application programs, and search strategy [2]. Consequently, researchers have primarily focused on heuristic search approaches and aggressive pruning of the search space [1, 20, 28] to address the phase ordering and selection issues.

A number of systems have been developed that use iterative or evolutionary algorithms to find the most effective phase combination. Such searches generally operate on an entire program or a per-function basis. A technique called *Optimization Space Exploration* uses a function-based strategy to search a statically pruned space of 2^{29} optimization parameters for *hot* functions, and uses static performance estimators (instead of program execution) to limit the search time. Kulkarni et al. employed evolutionary algorithms on individual functions to search for effective phase orderings in a search space of up to 15^{44} phase sequences [19, 24]. Cooper et al. were the first to employ genetic algorithms during iterative searches over entire programs to find the best phase ordering to reduce program code size in a solution space size of 10^{12} possible sequences [8]. In addition to finding custom phase orderings for individual programs, they were also able to construct a fixed sequence that generated up to 40% smaller codes than the default sequence used in their compiler. Other approaches used aggressive pruning of the search space to avoid evaluating sequences that are not likely to lead to improved benefits.

Researchers have also investigated the problem of finding the best set of compiler optimization flags (phase selection) for each program. Chow and Wu applied a technique called *fractional factorial design* [5], Haneda et al. used the *Mann-Whitney* test [15], Pan and Eigenmann employed three different feedback-driven orchestration algorithms [27], and Hoste and Eeckhout proposed a multi-objective evolutionary technique called *Compiler Optimization Level Exploration* [16] to effectively select a single setting of compiler optimization flags for the entire application. All such studies attempted to find a single distinct set of optimization flag setting for entire programs, and demonstrated that different programs do achieve their best performance with distinct flag settings.

Researchers have explored various schemes to limit the time of iterative searches. For example, researchers have used static estimation techniques to avoid expensive program simulations for performance evaluation [9, 22, 28]. Agakov et al. characterized programs using static *features* and developed adaptive mechanisms using statistical correlation models to focus their iterative search to those areas of the phase ordering space most likely to provide the greatest performance benefit [1]. A separate study by Kulkarni et al. employed several innovative search space pruning techniques to avoid over 84% of program executions, thereby considerably reducing the search overhead [20, 23]. Fursin et al. devised a novel strategy to evaluate the relative benefit of multiple per-function phase sequences during a single program execution by maintaining different versions of the same function in one executable and switching between them during execution [10]. Thus, this approach can substantially reduce the search overhead when the paths taken within the function are guaranteed to be the same each time. However, to our knowledge, there is no previous work that compares the performance benefit and search time overhead of function-based iterative search approaches over a program-based approach.

3. Entire Program, File, and Individual Function Searches

Empirical searches for finding the most effective optimization phase sequence can be conducted at different levels of code granularity. It is possible to perform these searches at either the function, file, or entire program level. Finer levels of search granularities enable greater flexibility in selecting distinct best sequences for different code segments in a program. This flexibility can potentially produce better-performing code than that achievable by a single phase sequence for the entire program.

At the same time, the search implementation strategy can also affect the levels of code granularities available for conducting the search. Search algorithms can either be implemented inside the compiler, or as a separate external program that invokes the compiler for each new phase sequence. Mechanisms implemented inside the compiler offer the most flexibility in terms of choosing the code granularity for the search. Moreover, such searches can proceed more quickly since the compiler need only be invoked once, and the search technique can have access to many internal compiler data structures for performance evaluation and function instance equivalence matching. However, this approach requires familiarity with the compiler, is more difficult to implement since it involves modifications to the compiler, and may need to be ported to each investigated compiler, which is a substantial development task.

In contrast, a search program to find the best phase sequence may be easier to implement outside the compiler, by invoking the compiler with different phase sequences (as determined by the search technique), and evaluating the performance of the resulting program each time. Moreover, such a search framework will be portable by allowing different compilers to be plugged into the same search program. However, most compilers provide no way to apply different phase sequences to individual functions in a single source file, thus eliminating the option of function-based search approaches. [1] Also, conventional compilers only permit turning optimizations on or off using the provided command-line flags, and do not support reordering optimizations phases, thus preventing investigations into the phase ordering problem.

To allow maximum flexibility in exploring phase order search strategies at different code granularities, we have implemented our search strategy inside the compiler for the techniques evaluated in this paper. This allows us to perform searches for the most effective optimization phase orderings at the function, file, and program levels. These three search algorithms are illustrated using the pseudo code in Figure 1. These algorithms avoid simulating the program when functions are redundant, which will be described later in the paper.

As previously mentioned, empirical search at lower levels of code granularity are desirable since they can potentially produce better performing code for the entire application. However, current implementations of function-based search techniques, [2] conduct their searches for the most effective optimization sequence for each function individually, and in isolation of the searches performed on the remaining functions in the program. For search strategies that employ *wall-time* or *execution cycles* for evaluating the merit of each phase sequence, program execution/simulation time (as compared to compilation time) is typically the dominant factor in the overall search time. For example, Cooper et al. reported execution time comprised 76% of their total search time, in spite of conducting their performance evaluations via native program executions [9]. For evaluation environments that require simulation

entire program approach:
```
DO
   determine next compilation settings;
   compile entire program with these settings;
   IF any function is not redundant THEN
      get entire program performance results
        by simulating the program;
UNTIL number of iterations completed;
```

individual file approach:
```
FOR each file in program DO
   DO
      determine next compilation settings;
      compile all functions in file with
        these settings;
      IF any function is not redundant THEN
         get performance of functions in file
           by simulating the program;
   UNTIL number of iterations completed;
END FOR
```

individual function approach:
```
FOR each function in program DO
   DO
      determine next compilation settings;
      compile function with these settings;
      IF function is not redundant THEN
         get function performance
           by simulating the program;
   UNTIL number of iterations completed;
END FOR
```

Figure 1. Pseudo-code for the search algorithms at three different program granularity levels, *entire program* level, *individual file* level, and *individual function* level

(a common scenario for embedded systems), the search overhead will be further dominated by the simulation time. In such cases, *naive* function-based search techniques that require up to *n* times more program executions/simulations can be up to *n* times more expensive than the program-based search approach, where *n* is the number of functions in the program.

Empirical searches at the file-level will typically require more program evaluations than a program-based approach at the cost of potentially losing some performance benefits as compared to the function-based approach. These searches are nonetheless important since they provide the lowest granularity for search algorithms that are implemented outside the compiler. Surprisingly, although several earlier investigations into the phase selection problem designed iterative search algorithms outside the compiler [1, 8, 14, 16], we are not aware of any prior work that suggested or studied the potential of file-based search approaches. Finally, we integrate the best features of function-based (higher performance potential for generated code) and program-based (typically lower search overhead) search approaches into a new *hybrid* search strategy. We provide a detailed description of this new search strategy along with its evaluation in Section 7.

4. Compiler Framework

The research in this paper uses the Very Portable Optimizer (VPO) [3], which is a compiler back end that performs all its optimizations on a single low-level intermediate representation called RTLs (Register Transfer Lists). This strategy allows VPO to apply most analyses and optimization phases repeatedly and in an arbi-

[1] Compilation frameworks, such as GCC 4.5 and MILEPOST GCC are notable exceptions [11].

[2] We are unaware of any work in file-based search strategies.

trary order. VPO compiles and optimizes one function at a time, allowing us to perform function-level phase order searches. Different functions may require very different phase orderings to achieve the best results, so a strategy that allows functions in a file to be compiled differently may achieve significant benefits [19].

The usual interface to the VPO compiler allows the user to input a single source file at a time to the compiler. Since the phase order search algorithm is implemented within the compiler, the above interface suffices for searching for the most effective optimization phase sequence at the individual function and file levels. However, to implement the search algorithm at the program level, we modified the VPO interface to accept multiple source files at the same time. While the VPO compilation process keeps track of the individual file boundaries, this distinction is transparent to the search algorithm. At the end of the search, VPO produces a separate assembly file corresponding to each source file to avoid static name conflicts.

For our experiments in this paper, the VPO compiler has been targeted to generate code for the StrongARM SA-100 processor using Linux as its operating system. The ARM is a simple 32-bit RISC instruction set. The relative simplicity of the ARM ISA combined with the low-power consumption of ARM-based processors have made this ISA dominant in the embedded systems domain. We used the SimpleScalar set of functional simulators [6] for the ARM to get dynamic performance measures. However, invoking the *cycle-accurate* simulator for evaluating the performance of every distinct phase sequence produced by the search algorithm is prohibitively expensive during our experimentation process. Therefore, we used a measure of estimated performance based partly on static function properties. [3] Our performance estimate accounts for stalls resulting from pipeline data hazards, but does not consider other penalties encountered during execution, such as branch misprediction and cache miss penalties. This approach has the additional advantage of evaluation of each function or file independent wrt to performance evaluation. Interestingly, we have shown that this measure of dynamic performance has a strong correlation with simulator cycles for an embedded processor [22]. For every unique function instance generated by the search process, our compiler instruments the assembly code and links it to produce an executable. We then use the fast SimpleScalar *functional* simulator on our instrumented executable to produce a count of the number of times each basic block is reached. This information is used by our performance estimator to provide our dynamic performance measures.

Table 1 describes each of the 15 candidate code-improving phases that were used during search algorithms. Unlike the other candidate phases, loop unrolling is applied at most once. Our search algorithm randomly selects one from among three different unroll factors (2, 4, and 8) whenever loop unrolling is present in the search sequence. The default VPO compiler is tuned for generating high-performance code while managing code-size for embedded systems, and hence uses a constant loop unroll factor of 2. In addition, *register assignment*, which is a compulsory phase that assigns pseudo registers to hardware registers, must be performed. VPO 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, VPO performs another compulsory phase that inserts instructions at the entry and exit of the function to manage the activation record on the run-time stack. Finally, the compiler also performs *predication* and *instruction scheduling* before the final assembly code is produced. These last two optimizations should only be performed late in the compilation process in the VPO compiler, and so are not included in the set of re-orderable optimization phases.

[3] Note that reducing this expensive simulation cost without affecting the performance benefits of the generated code is the goal we are attempting to achieve in this research.

For the experiments described in this paper we used a subset of the benchmarks from the *MiBench* benchmark suite, which are C applications targeting specific areas of the embedded market [13]. We selected two benchmarks from each of the six categories of applications present in MiBench. Table 2 contains descriptions of these programs. VPO compiles and optimizes individual functions at a time. The 12 benchmarks selected contained a total of 251 functions, out of which 90 were executed (at least once) with the standard input data provided with each benchmark.

Category	Program	Files/ Funcs.		Description
auto	bitcount	10	18	test proc. bit manipulation abilities
	qsort	1	2	sort strings using the quicksort algo.
network	dijkstra	1	6	Dijkstra's shortest path algorithm
	patricia	2	9	construct patricia trie for IP traffic
telecomm	fft	3	7	fast fourier transform
	adpcm	2	3	compress 16-bit linear PCM samples to 4-bit samples
consumer	jpeg	7	62	image compression and decomp.
	tiff2bw	1	9	convert color *tiff* image to b&w
security	sha	2	8	secure hash algorithm
	blowfish	6	7	symmetric block cipher with variable length key
office	search	4	10	searches for given words in phrases
	ispell	12	110	fast spelling checker

Table 2. MiBench Benchmarks Used

5. Search Algorithm Configuration

We adopt a variant of a popular genetic algorithm (GA) based search technique for the experiments in this paper [8, 19]. We also employ the latest techniques available in the literature to avoid redundant program compilations and executions during the genetic algorithm search process in order to make the experiments feasible within a reasonable amount of time [20]. At the same time, we also believe that the conclusions of this work are, most likely, independent of the heuristic algorithm that is employed during the empirical search process. In this section, we present the details of our search algorithm, and the techniques employed to avoid redundant program compilations and executions.

5.1 Base Genetic Algorithm

Genetic algorithms are adaptive algorithms based on Darwin's theory of evolution [26]. There are several parameters of a genetic algorithm that can be varied for different search configurations. *Genes* in the genetic algorithm correspond to optimization phases, and *chromosomes* correspond to optimization phase sequences. The set of chromosomes currently under consideration constitutes a *population*. The number of *generations* is how many sets of populations are to be evaluated. Previous studies have indicated that genetic algorithm based searches generally produce better phase sequences faster than a pure random sampling of the search space [8]. Additionally, it has also been revealed that genetic algorithms are competitive with most other *intelligent* phase sequence search techniques, and minor modifications in the configuration of the GA search parameters do not significantly affect their performance [24].

For our current study we have fixed the number of chromosomes in each population at 20. Chromosomes in the first generation are randomly initialized. After evaluating the performance of each chromosome in the population, they are sorted in decreasing order of performance. During crossover, 20% of chromosomes

Optimization Phase	Description
branch chaining	Replaces a branch or jump target with the target of the last jump in the jump chain.
common subexpression elimination	Performs global analysis to eliminate fully redundant calculations, which also includes global constant and copy propagation.
remove unreach. code	Removes basic blocks that cannot be reached from the function entry block.
loop unrolling (unroll factors 2, 4, and 8)	To potentially reduce the number of comparisons and branches at runtime and to aid scheduling at the cost of code size increase.
dead assign. elim.	Uses global analysis to remove assignments when the assigned value is never used.
block reordering	Removes a jump by reordering 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	Uses graph coloring to replace references to a variable within a 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.
code abstraction	Performs cross-jumping and code-hoisting to move identical instructions from basic blocks to their common predecessor or successor.
eval. order determ.	Reorders instructions within a single basic block in an attempt to use fewer registers.
strength reduction	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 shift, adds, and subtracts.
reverse branches	Removes an unconditional jump by reversing a cond. branch when it branches over the jump.
instruction selection	Combines pairs or triples of instructions together where the instructions are linked by set/use dependencies. Also performs constant folding and checks if the resulting effect is a legal instruction before committing to the transformation.
remove useless jumps	Removes jumps and branches whose target is the following positional block.

Table 1. Candidate Optimization Phases

from the poorly performing half of the population are replaced by repeatedly selecting two chromosomes from the better half of the population and replacing the lower half of the first chromosome with the upper half of the second and vice-versa to produce two new chromosomes each time. During mutation we replace a phase with another random phase with a small probability of 5% for chromosomes in the upper half of the population and 10% for the chromosomes in the lower half. The chromosomes replaced during crossover are not mutated. During all our experiments, we iterate the genetic algorithm for 200 generations.

The sequence length of each chromosome should be sufficiently long to provide the genetic algorithm maximum opportunity to find the most effective phase sequence. However, too long a sequence can potentially increase the compilation overhead with no benefit to the best generated performance. An *active* phase is one that is able to successfully apply one or more transformations to the program representation. To find the right balance between compilation cost and performance opportunity, we first find the sequence length of active phases applied by our default (*batch*) compiler. Since all phases in a chromosome are not guaranteed to be active, the sequence length is selected to be twice the batch sequence length. For function and file-level searches, all functions in a single file use a sequence length that is twice the maximum batch length over all functions in that file. Similarly, program-level searches choose twice the maximum batch length over all functions in the entire program.

All heuristic-based search algorithms attempt to either maximize or minimize a cost function or *fitness criteria*. In the domains of desktop or high-performance computing, the fitness criteria is typically just the runtime speed of the resulting program. However, in the embedded system domain, memory consumption (measured by the size of the generated code) is often as important as the speed of execution in several cases. Therefore, the fitness criteria employed by our search algorithm attempts to maximize a performance measure that is an equally weighted function of dynamic performance and code size. These weights can be easily modified to meet the constraints of a specific embedded system. Moreover, the fitness criteria of the function and hybrid approaches for each function are relative to the unoptimized performance numbers over the entire program. Consequently, for a frequently executed function, an intelligent search process would progressively select phase sequences that emphasize reducing execution time at the expense of increasing code size (for example, by selecting more aggressive unroll factors for loop unrolling). In contrast, the search would primarily select optimization phases that reduce code size for an infrequently executed function.

5.2 Techniques to Remove Redundancy During the GA Search

Researchers have observed that several sequences found during a heuristic algorithm search are *similar* to sequences seen earlier in the search. In such cases, if we store the performance results of previous phase sequences, then we can use various redundancy detection schemes to avoid the compilation of several phase sequences and the execution/simulation for similar sequences found later. We have used several techniques derived from our previous studies to detect and eliminate redundancy during our search process [20]. We briefly describe these techniques in this section, and quantify the redundancy found during our experiments.

We employ the following redundancy detection techniques during our experiments:

Identical attempted sequence: If the current phase sequence to be attempted is identical to some chromosome seen earlier, then the algorithm can avoid having the compiler even apply the optimization phases. Performance measures of all distinct attempted phase sequences are maintained in fast access hashtables.

Identical active sequence: After applying the phases in the current chromosome, if the sequence of active phases is identical to some earlier active sequence, then we can avoid program simulation for performance evaluation. All active phase sequences are stored in hashtables.

Identical function instance: Different sequences of active phases can sometimes produce identical code. Our algorithm detects such cases using multi-variable function hash-values, and prevents program simulation in such cases.

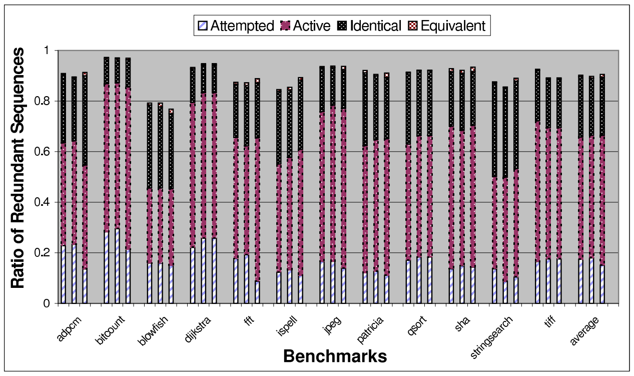

Figure 2. Number of phase sequences that are found to be redundant by various techniques during the GA-based search algorithm. Each benchmark has three bars, one for each of the following configurations: (a) function, (b) entire file, (c) entire program.

Equivalent function instance: Earlier studies reveal that it is also possible for function instances produced by different active phase sequences to not be completely identical, but only differ cosmetically in the register number used or in the labels of basic blocks. Our algorithm avoids program simulation in such cases as well.

The redundancy detection schemes are performed in the order specified above. Figure 2 presents the number of redundant instances detected by each detection technique, and for each of our three search strategies. We can see that close to 90% of the sequences generated by the genetic algorithm are detected to be redundant. Additionally, the redundancy ratio remains about the same for each search strategy. One reason for the large amount of redundancy is the large number of generations (iterations) computed during the genetic algorithm. As the various redundancy detection tables are populated, later generations tend to produce significantly more redundant sequences than earlier generations. However, note that even the evaluation of 10% of the 4000 phase sequences results in 400 program simulations on average for each function, which depending on the search strategy can result in a prohibitively long search time. For example, for our set of benchmarks, the current function-based search strategy requires about ($400 * 90 = 36{,}000$) different program simulations, where 90 is the total number of executed functions across all benchmarks.

6. Evaluation of Function, File, and Program Level Search Algorithms

As mentioned earlier, we have implemented our genetic algorithm based search strategy at three different program granularities: function, file, and program. A function-based search approach has the potential of the greatest performance benefit by finding a customized phase sequence for each individual function, but at the cost of more program evaluations via expensive simulations. In contrast, program-based searches lose flexibility by attempting to determine the most effective phase sequence for the entire program, thus overlooking the finer grained program characteristics. However, by conducting a single search (instead of *n* searches conducted by function-based approaches), program-based searches require less time. A file-based search strategy can provide a compromise between the two alternatives. In this section, we compare the search cost and the performance of the best code generated by each of our three search strategies: (a) function, (b) file, and (c) program level searches. To save space, the graphs in this section also show

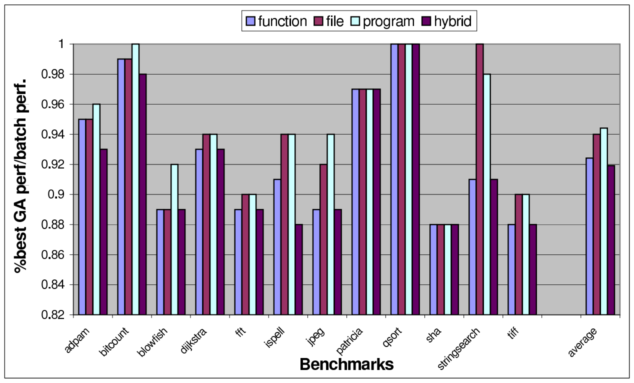

Figure 3. Entire program performance benefit achieved by all four indicated search strategies

results for our hybrid search strategy. The hybrid search strategy and results are described in Section 7.

6.1 Comparing Performance of Generated Code

Figure 3 plots the improvement in performance (50% speed and 50% code size) for each of our three search strategies, as compared to the batch compiler results. The default VPO batch compiler applies the 15 optimization phases in a fixed order, but attempts them repeatedly until the last iteration is unable to make any further changes to the program representation. Thus, the batch compiler provides a very aggressive baseline for the search algorithms. In spite of this aggressive baseline, the function search strategy achieves an average performance improvement of 8%, and up to 12% in the best case. There was also about an 8% improvement in both execution cycles and code size on average for the function-level search strategy.

As expected, the additional flexibility inherent in function-level search strategies enables them to produce code that is optimized by different customized phase sequences for each function. Thus, function-level searches can select a distinct best phase sequence over smaller program units than a file or entire program-level approach. This advantage allows function-based searches to produce the best overall code, surpassing that obtained by the file and program level searches. Correspondingly, file-level searches are also able to leverage the same advantage of optimizing over smaller program units to produce code that performs slightly better than the program-level approach for several benchmarks.

The results in Figure 3 allow us to make several other interesting observations as well. The effectiveness of file-based search in producing efficient code depends on the distribution of functions across the different files in a program. Thus, for benchmarks like *blowfish* that contain only a single function in most files, the performance of the code generated by file-based searches is close to that delivered by function-based searches. In contrast, for single file programs, like *dijkstra*, *qsort*, and *tiff*, a file-based search generates code that is equivalent in performance to that produced by a program-based approach.

6.2 Comparing Search Progress

Another important measure of the effectiveness of a particular iterative search algorithm is the number of iterations required to reach the best or the *steady-state* performance for each benchmark. More adept search algorithms typically reach their steady-state performance during earlier iterations of the search process. For a genetic algorithm based search process, each *generation* of the search is considered to be one iteration.

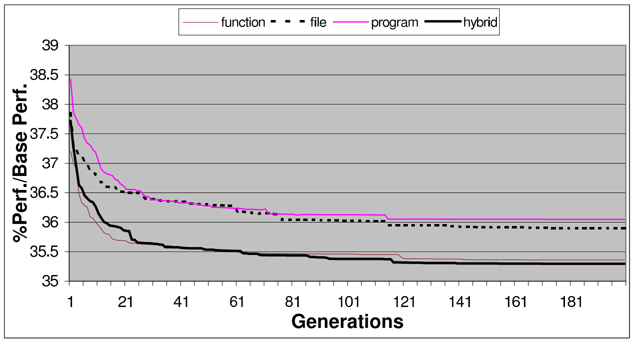

Figure 4. Progress of the GA search during *function*, *program*, *file* and *hybrid* mode, averaged over all 12 MiBench benchmarks. *Function* and *hybrid* level searches are using variable sequence lengths per function.

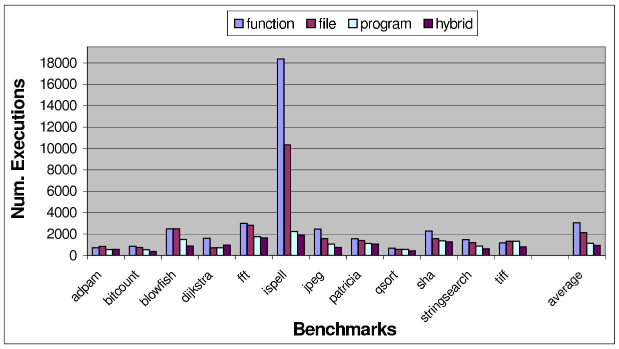

Figure 5. Number of program simulations during genetic algorithm searches at all four indicated search strategies

Figure 4 plots the average performance of the code produced by the best sequence during each generation of the genetic algorithm. The performance number is averaged across all the 12 selected MiBench benchmarks. Since all three search strategies (function, file, program) employ the same genetic algorithm, they all achieve their respective best, or close to best, performances at about the same time, and relatively early, during the search process. However, more importantly, the finer granularity search algorithms start generating better code than their higher granularity counterparts almost immediately after the search starts. Therefore, a function-based search strategy will likely outperform the entire file and program level strategies even if the search is performed for fewer number of generations.

6.3 Comparison of Search Costs

Although finer-granularity search strategies achieve better performing code, current implementations do so at the cost of increased search time. Our experiments were performed on a number of processors and required a period of over a month to complete. We found that the search times varied significantly depending on the processor that was used and the load of the machine when the search was performed. In general, the search times did improve as we expected, but there were a few abberations in specific benchmarks. The search time of an iterative search algorithm is comprised of the following two main components.

Compilation Time: This is the time spent by the compiler as it applies optimization phases to the current function, file, or program to generate the output code. Although often a minor component of the total search time in the general case, compilation time becomes significant for very large functions due to both the time required to apply each phase and typically longer phase sequences. In particular, compilation time is the only component of the search time for unexecuted functions, which the search algorithm optimizes purely for reduced code size.

Execution/Simulation Time: This is the component of the search time that is spent executing or simulating the program to measure the dynamic performance of the code produced by the current phase sequence. As mentioned earlier, in typical cases, this is the major portion of the search time.

We decided to instead report repeatable counts for these two components since we believe these counts provide more meaningful information than the actual wall clock times.

Current implementations of function-level searches explore the space of each function individually and independently of the remaining functions in the program. Therefore, each program execution/simulation is only able to reveal performance information regarding the current function. Thus, for a program with *n* distinct functions, a function-based search strategy may result in up to *n* times more program executions than a program-based approach.

Figure 5 shows the number of program executions/simulations required during individual function, file, and program level search algorithms. To maintain a uniform compilation time for the current experiments, we use a constant sequence length during our genetic algorithms for all functions in a single application. This length is selected to be twice the maximum length of *active* phases applied by the batch compiler over all functions in the program. Note that the selected sequence length for any benchmark is still about 5-10 times smaller than the number of phases *attempted* by the batch compiler for most functions. The phase sequence evaluated during our genetic algorithm may contain unsuccessful (dormant) phases interspersed with active phases, and the selected sequence length allows maximum opportunity to the genetic algorithm to construct more effective sequences.

The number of program executions shown in Figure 5 was directly affected by the number of executed functions in each application. It is easy to see that the number of program-based executions was less than the number of file-based executions. Likewise, the number of file-based executions was exceeded by the number of function-based executions. However, one can see that the average was skewed by the results for *ispell*, which had a significantly greater number of executed functions and hence a greater number of program executions.

A more meaningful measure is to compare against the number of executions for the function-level approach without avoiding executions for redundant phase sequences. The baseline number of executions in this case would be 4000**n*, where *n* is the number of functions invoked one or more times during the program's execution and 4000 represents the maximum number of unique instances of functions, which is 20 chromosomes (sequences) for 200 generations. This measure allows each benchmark result to be weighted the same, regardless of the number of functions executed in the program. Thus, we can see from Figure 6 that a program-based search strategy requires only about 59% of the number of executions, on average, compared to those required for function-based searches. Again, file-based approaches achieve a middle ground and perform close to 84% as many program executions, on average, as those performed by an individual function approach. The results for the benchmark *tiff* are an exception, where the function-based search actually requires fewer executions than a corresponding program-based approach. Further analysis reveals that *tiff* has very few executed functions (four) with the largest among them dominating

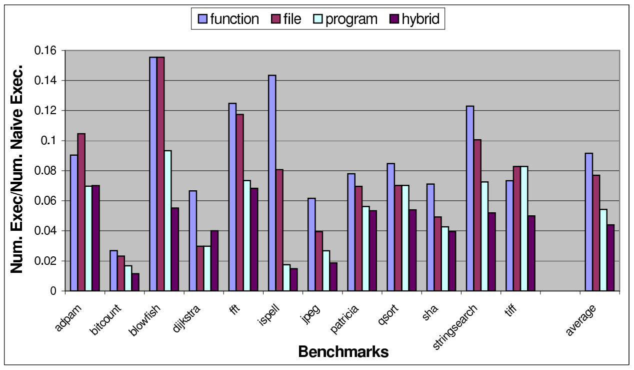

Figure 6. Ratio of program simulations during genetic algorithm searches to number of simulations using the function approach with no redundant sequences

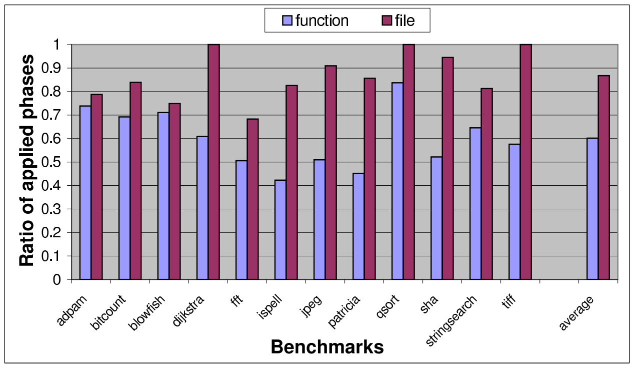

Figure 7. Ratio of (number of phases applied during *function* and *file* level search)/(number of phases applied during *program*-level search)

the number of executions count. In such cases, the inherent randomness in the GA search can produce such anomalies. For similar reasons, the *adpcm* requires the most executions with its file-based search approach. *adpcm* has only one executed function in each of its two files, and this slight increase in the number of executions during the file-based search can also be attributed to the genetic algorithm taking a different path through the search space. We can also see that a file-based strategy degenerates to a program-level approach (and performs the same number of executions) for single file benchmarks, namely, *dijkstra*, *qsort*, and *tiff*.

The other, and typically smaller, component of the search time during an iterative algorithm is the compilation time required to apply the optimization phases. In our earlier results comparing the number of program executions presented in Figures 5 and 6, all algorithms use a uniform sequence length for each benchmark, which is twice the maximum batch sequence length. However, the additional flexibility available in function (and file) based approaches can allow the genetic algorithm to use a distinct sequence length for each function (or file) in the program. Thus, searches for smaller functions can now work with smaller sequence lengths, thereby reducing the compilation overhead is such cases.

To quantify the reduction in the number of applied phases, we conducted a set of experiments that employ custom sequence lengths for each function during a function-based search, and each file during file-based search. The sequence length selected is twice the batch active sequence length of each function for function-level searches, twice the length of the longest active sequence among the functions in each file during file-level searches, and twice the length of the longest active sequence for all functions during program-level searches. The results of this study, illustrated in Figure 7, show that function-based searches only need to apply about 60% of the number of phases, on average, compared to the number applied during a program-based search. A similar comparison for file-based searches reveals a drop in the number of phases applied to 87%, on average, as compared to the program-based searches. The performance numbers of the best code generated by function and file level searches with variable sequence lengths remained the same.

Thus, the results described in this section enable us to make the following conclusions that were never reported earlier (to the best of our knowledge):

1. Function-level searches for the most effective optimization phase sequence produce better performing code than a program-level search. Additionally, based on the program layout, file-based searches are also more effective at producing better code than a program-based approach.
2. The finer granularity of function (and file) based search strategy, allows this approach to typically reach better performance even during the earlier iterations of the search process.
3. The major drawback of finer granularity searches is the large number of additional program executions/simulations required over a program-based strategy, which often dominate the search time. However, function-based approaches can save some compilation time by using variable sequence lengths per function, and thus applying fewer number of phases than a program-based approach that needs to employ a single sequence length corresponding to the largest function in each program.

7. Hybrid Search Strategy

In the previous section we showed that both function and program based searches have their respective advantages and drawbacks. A file-based strategy can provide a middle ground, and, most importantly, may be the finest granularity available for search strategies implemented outside the compiler. In this section we propose and evaluate a new search strategy that encompasses and leverages the best features of both function and program based iterative search approaches.

Current implementations of function-based search approaches isolate the evaluation of each function from the remaining functions in the program. As a consequence of this isolated evaluation, a function-based search can require up to n times more program executions than a corresponding search over the entire program, where n is the number of functions in the program. Instead, our new *hybrid* search strategy is a function-based approach that performs the searches for all n functions simultaneously. Our hybrid search strategy delays program simulations for the performance estimation of individual functions, until all (executable) functions in the entire program also require a performance evaluation or have completed their search. The individual function performance achieved by distinct phase sequences for each function can now be evaluated in a single program simulation. Although obvious in hindsight, it should be noted that this search strategy, to the best of our knowledge, has never been attempted in the past.

This hybrid search strategy can be best described using the pseudo code in Figure 8. The outlined hybrid search strategy achieves all the advantages of both function and program-based search techniques. In fact, the hybrid strategy, in most cases, requires even fewer program simulations than the program-based approach. Both the hybrid and program-based strategies conserve search time by overlapping the evaluations of multiple individual functions in one program execution. However, a program-based approach applies the same phase sequence to all functions in a *synchronized* manner. Therefore, it requires program execution whenever *any* function updated by the current phase sequence needs to be evaluated for performance. As the algorithm in Figure 8 shows, the searches for individual functions in a hybrid search strategy proceed unsynchronized, and a function is passed only when it requires evaluation. Thus, in contrast to a program-based approach, a hybrid search strategy performs its executions only when *all* functions whose searches have not yet been completed in the program need performance evaluation.

```
DO
   FOR each function in program DO
      IF function search still incomplete THEN
         DO
            determine next compilation settings
              for this function;
            compile function with these settings;
         UNTIL function is not redundant OR
               function search is complete
      ENDIF
   END FOR
   get results of each function by
     simulating program once;
UNTIL number of search generations completed
  for all functions in program;
```

Figure 8. Algorithm for hybrid search approach

This further saving in the number of executions/simulations can be observed from Figure 6, which shows the number of program simulations relative to the number required for the function-level approach when not avoiding redundant sequences during each of four different search strategies, *function*, *file*, *program*, and *hybrid*. Thus, a function-level hybrid strategy requires about 48% fewer program simulations, on average than the program-level approach and only about 4% of the total simulations as compared to a naive function-level approach.

Most importantly, the hybrid approach is able to leverage all the advantages that are inherent in the flexibility allowed by a finer granularity search approach. Thus, our hybrid strategy uses customized sequence lengths for individual functions, producing the savings in compilation times illustrated earlier in Figure 7. Figure 3 compares the performance of the hybrid search approaches using variable sequence lengths with the performance of the earlier three search strategies. As we had observed earlier, the reduced sequence lengths for smaller functions do not produce any degradation in the performance of the code generated by the best sequence. [4] Finally, Figure 4 compares the average performance over all benchmarks during each iteration of a hybrid search, function-level search, file-level search, and program-level search. This figure again confirms that a hybrid-based search shows performance characteristics similar to a function-level approach.

8. Future Work

In the future, we plan to further investigate three issues related to the study presented in this paper. First, our current results demonstrate that customizing the phase sequence over finer program levels can lead to greater performance benefits. In this study we selected an individual function as the finest granularity for conducting the phase sequence search. Instead, in the future, we would like to lower the granularity further, and explore the most effective phase sequence for individual *loops* within a single function, and quantify the resulting performance benefit over the entire program. Similar to our current work, we will also devise additional search strategies to limit the number of program executions during loop-level searches for effective phase sequences.

In addition to genetic algorithms, researchers investigating the phase ordering and phase selection problems have incorporated various other heuristic, evolutionary, and statistical mechanisms during their phase sequence searches, including simulated annealing, hill-climbing, orthogonal arrays, fractional factorial design, logistic regression, as well as other custom approaches. Likewise, we plan to conduct our experiments using other search mechanisms to assess if our current results regarding the performance benefits and search time improvements transcend other heuristic and statistical mechanisms.

Finally, we also plan to evaluate the use of a cluster of processors to reduce the search time. It will also be interesting to study how the various search mechanisms lend themselves to parallelism on multi-core or multi-processor machines. The most effective and fastest search strategy will likely be one where individual phase sequences or individual functions are able to be evaluated independently on separate processors. We believe that the techniques presented in this paper can be extended to further enhance the search time on multi-processor machines.

[4] The slight average performance improvement of hybrid search over function-based search is, most likely, due to a different random path selected by the genetic algorithm during the hybrid approach.

9. Conclusions

Phase ordering and phase selection are important problems in compiler optimization research, and are especially relevant to the area of performance and cost-constrained embedded systems. Iterative searches for the most effective phase sequences are typically conducted at the *function* or *entire program* level. This paper describes the first study to compare the performance benefits and costs of searches conducted at these two levels. We conclude that the finer granularity of function-level searches allows the search algorithm to find better customized phase sequences over smaller code units, resulting in enhancing the overall program performance in most cases, but at a significant cost in search overhead. We further introduced and evaluated a *file*-level search strategy that can provide the finest granularity searches for mechanisms implemented outside the compiler.

Previously, a major concern with function-level searches was the additional search overhead due to the number of program executions/simulations increasing linearly with the number of functions in the program. To alleviate this concern, we introduced a new *hybrid* search strategy that conducts all function-level searches simultaneously, and reduces the number of program executions/simulations to the number required for the function having the most nonredundant sequences in the program. We demonstrated that a hybrid search strategy can reduce the number of program executions to be even less than the number required by a program-based approach, while retaining the performance benefits and compilation time savings of the function-based approach. Thus, our hybrid search strategy using a variable sequence length achieves the best advantages of function and program level searches.

Acknowledgments

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by NSF grants CNS-0615085 and CNS-0915926.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA, 2006.

[2] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 231–239, 2004. ISBN 1-58113-806-7.

[3] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 329–338. ACM Press, 1988. ISBN 0-89791-269-1.

[4] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, , and E. Rohou. Iterative compilation in a non-linear optimisation space. Proc. Workshop on Profile and Feedback Directed Compilation.Organized in conjuction with PACT'98, 1998.

[5] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*. John Wiley & Sons, 1 edition, June 1978. isbn:0471093157.

[6] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[7] J. Cavazos and M. F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 229–240, 2006. ISBN 1-59593-348-4.

[8] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, May 1999. URL `citeseer.ist.psu.edu/cooper99optimizing.html`.

[9] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Acme: adaptive compilation made efficient. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 69–77, 2005. ISBN 1-59593-018-3.

[10] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. Quick and practical run-time evaluation of multiple program optimizations. pages 34–53, 2007.

[11] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, C. Williams, and M. O. Boyle. Milepost gcc: machine learning based research compiler. GCC Summit, 2008. http://www.milepost.eu/.

[12] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 442–452, 1988. ISBN 0-89791-272-1.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[14] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Generating new general compiler optimization settings. In *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*, pages 161–168, 2005.

[15] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2429-X.

[16] K. Hoste and L. Eeckhout. Cole: Compiler optimization level exploration. In *accepted in the International Symposium on Code Generation and Optimization (CGO 2008)*, 2008.

[17] T. Kisuki, P. Knijnenburg, M. O'Boyle, F. Bodin, , and H. Wijshoff. A feasibility study in iterative compilation. In *Proceedings of ISHPC'99, volume 1615 of Lecture Notes in Computer Science*, pages 121–132, 1999.

[18] T. Kisuki, P. Knijnenburg, , and M. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Internation Conference on Parallel Architectures and Compilation Techniques*, pages 237–246, 2000.

[19] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 12–23. ACM Press, 2003. ISBN 1-58113-647-1.

[20] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 171–182, Washington DC, USA, June 2004.

[21] P. Kulkarni, D. Whalley, G. Tyson, and J. Davidson. Exhaustive optimization phase order space exploration. In *Proceedings of the Fourth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 306–308, March 26-29 2006.

[22] P. Kulkarni, D. Whalley, G. Tyson, and J. Davidson. In search of near-optimal optimization phase orderings. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers and tool support for embedded systems*, pages 83–92, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-362-X.

[23] P. A. Kulkarni, S. R. Hines, D. B. Whalley, J. D. Hiser, J. W. Davidson, and D. L. Jones. Fast and efficient searches for effective optimization-phase sequences. *ACM Transactions on Architecture and Code Optimization*, 2(2):165–198, 2005. ISSN 1544-3566.

[24] P. A. Kulkarni, D. B. Whalley, and G. S. Tyson. Evaluating heuristic optimization phase order search algorithms. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 157–169, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2764-7.

[25] B. W. Leverett, R. G. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz, and W. A. Wulf. An overview of the production-quality compiler-compiler project. *Computer*, 13(8):38–49, 1980. ISSN 0018-9162.

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

[27] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 319–332, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0.

[28] S. Triantafyllis, M. Vachharajani, N. Vachharajani and D. I. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 204–215. IEEE Computer Society, 2003. ISBN 0-7695-1913-X.

[29] S. R. Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. In *Proceedings of the 15th Annual Workshop on Microprogramming*, pages 125–133. IEEE Press, 1982.

[30] D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. In *Proceedings of the second ACM SIGPLAN symposium on Principles & Practice of Parallel Programming*, pages 137–146. ACM Press, 1990. ISBN 0-89791-350-7.

[31] M. Zhao, B. Childers, and M. L. Soffa. Predicting the impact of optimizations for embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN Conference on Language, compiler, and tool for embedded systems*, pages 1–11, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-647-1.

[32] M. Zhao, B. R. Childers, and M. L. Soffa. A model-based framework: An approach for profit-driven optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 317–327, Washington, DC, USA, 2005. ISBN 0-7695-2298-X.