Exploiting Phase Inter-Dependencies for Faster Iterative Compiler Optimization Phase Order Searches

Michael R. Jantz Prasad A. Kulkarni
Electrical Engineering and Computer Science, University of Kansas
{mjantz,kulkarni}@ittc.ku.edu

Abstract

The problem of finding the most effective set and ordering of optimization phases to generate the best quality code is a fundamental issue in compiler optimization research. Unfortunately, the exorbitantly large phase order search spaces in current compilers make both exhaustive as well as heuristic approaches to search for the ideal optimization phase combination impractical in most cases. In this paper we show that one important reason existing search techniques are so expensive is because they make no attempt to exploit well-known independence relationships between optimization phases to reduce the search space, and correspondingly improve the search time. We explore the impact of two complementary techniques to prune typical phase order search spaces. Our first technique studies the effect of implicit application of cleanup phases, while the other partitions the set of phases into mutually independent groups and develops new multi-stage search algorithms that substantially reduce the search time with no effect on best delivered code performance. Together, our techniques prune the exhaustive phase order search space size by 89%, on average, (96.75% *total* search space reduction) and show immense potential at making iterative phase order searches more feasible and practical. The pruned search space enables us to find a small set of distinct phase sequences that reach near-optimal phase ordering performance for all our benchmark functions as well as to improve the behavior of our genetic algorithm based heuristic search.

Keywords optimization ordering, iterative compilation, search space pruning

1. Introduction

Finding the best set and combination of optimization phases to apply is a fundamental, long-standing and persistent problem in compiler optimization research [11, 20, 25]. Each optimization phase in a compiler applies a series of code transformations to improve program quality with respect to performance, code size and/or power. Successful application of an optimization phase often depends on the availability of registers and the existence of specific instruction and control-flow patterns in the code being optimized. The problem of optimization phase ordering is caused by the interactions between optimization phases that create and destroy the conditions for the successful application of successive phases. Consequently, different phase orderings can generate different program representations with distinct performances.

Researchers have also discovered that there is no single ordering of phases that will produce the best code for all applications [6, 24, 26]. Instead, the ideal phase sequence depends on the characteristics of the code being compiled, the compiler implementation, and the target architecture. Earlier research works uniformly find that applying customized per-function or per-program optimization phase sequences can significantly improve performance – by up to 350% with GCC [10], up to 85% with SUIF / GCC [1], up to 47% with their research compiler [2], and up to 33% with VPO, compared to the fixed sequence used in their respective compilers. It is also crucial in certain application areas, like embedded systems, for the compiler to generate the best possible output code. Small improvements in program speed, memory (code size), and/or power consumption in such areas can result in huge savings for products with millions of units shipped.

An *exhaustive* solution to the phase ordering problem attempts to evaluate the performance of codes generated by all possible optimization sequences to find the *optimal* solution for each function / program. However, current compilers implement many optimizations, causing exhaustive phase order searches to operate in extremely large search spaces. Researchers have shown that, depending on the compiler employed, exhaustive phase ordering searches in existing compilers may need to evaluate up to 10^{12} [6], 2^{29} [24], or 15^{44} [19] different phase combinations, making such an approach extremely time-consuming in most cases. In spite of such high costs, exhaustive searches are still *critical* to: (a) understand the nature of the phase order search space, (b) determine the performance limits of the phase ordering problem for a compiler, and (c) evaluate the effectiveness of faster, but imprecise, *heuristic* solutions to the problem.

Researchers are also developing heuristic machine-learning and statistical algorithms that can result in a more focused search, and have been shown to often provide good phase ordering solutions. Unfortunately, to be effective, even these heuristic algorithms still need to evaluate hundreds of different phase sequences in most cases. All search approaches that generate and evaluate several function instances are called *iterative* search algorithms.

Hypothesis: In this paper we hypothesize that one reason for the high search-time cost of existing iterative algorithms for phase order search space exploration is that they do not take into account intuitive and well-known relationships between specific groups of optimization phases. Although the issue of phase ordering exists due to unpredictable phase interactions, each phase may not interact with every other phase in a compiler. In this work, we focus on two important categories of phase independence relationships that exist in our compiler. First, some optimization phases in a compiler, such as *dead assignment elimination* and *dead code elimi-*

nation, can be designated as *cleanup* phases that do not affect any useful instructions in the program, and are thus unlikely to have many interactions with other phases. Removing such phases from the phase order search space and applying them implicitly after every relevant phase may substantially prune the phase order search space. Second, it may be possible to partition the set of optimization phases into distinct branch and non-branch sets since these phases are most likely to interact with phases within their sets, but show minimal interactions with phases in the other set. Such phase set partitioning may allow the phase order search to be conducted as a series of smaller searches, each over fewer optimization phases. Just as $(m! + n!)$, for larger values of m and n, is much smaller than $(m+n)!$, such *staged* phase order searches over smaller partitioned optimization sets can result in huge search space reductions without any loss in the performance of the best generated code.

Goal: The goal of this work is to validate and test this hypothesis for our compiler and measure the resulting reduction in phase order search space size and effect on best phase ordering performance. Such *innocuous* search space reductions can make exhaustive and heuristic phase order searches much more feasible and practical in existing and future compilers. Thus, the primary contributions of this work are:

1. This is the first work that employs *independence* characteristics of optimization phases to prune the phase order search space.
2. This paper presents the first exploration of the effect of implicitly applying cleanup phases during the phase order search on the size of the search space and best achieved performance.
3. This is the first investigation of the possibility and impact of partitioning optimizations and performing phase order searches in multiple stages over smaller disjoint subsets of phases. We also design and implement the first algorithm to conduct exhaustive (multi-stage) phase order searches over partitioned sets of phases.
4. We show how to use our observation regarding phase independence to find a common set of near-optimal phase sequences, and to improve heuristic genetic algorithm-based phase order searches.

The rest of the paper is organized as follows. We describe related work in the next section. Our compiler setup and experimental framework are presented in Section 3. In Section 4, we describe our observations regarding the pair-wise independence interactions between optimization phases to validate our hypothesis. In Section 5 we explore the effect of implicit application of cleanup phases on the search space size and best performance delivered by exhaustive phase order searches. In Section 6 we investigate the effect of partitioning the set of optimization phases, and develop new algorithms for exhaustive and heuristic phase order searches. We list future work in Section 7 and present our conclusions in Section 8.

2. Related Work

Compiler researchers have investigated several strategies to explore and address the phase ordering problem. Despite suggestions that the phase ordering problem is theoretically *undecidable* [23], exhaustive evaluation of the compiler optimization phase order search space has been conducted in some existing compilers, but is extremely time consuming [19]. Researchers have devised aggressive techniques to detect redundancies in the phase order search space to allow exhaustive search space enumeration for most of their benchmark functions [17, 18]. Enumerations of search spaces over subsets of available optimization phases have also been attempted before. One such work exhaustively enumerated a 10-of-5 subspace (optimization sequences of length 10 from 5 distinct optimizations) for some small programs [2]. Another study evaluated different orderings of performing loop unrolling and tiling with different unroll and tiling factors [16]. In this work, we employ an algorithm for exhaustive phase order search space exploration discussed in earlier works [18, 19]. However, our techniques to prune the phase order search space are novel and complementary to existing techniques.

Research for addressing the phase ordering/selection problem has also focused on designing and applying heuristic algorithms during iterative compilation to find good (but, potentially suboptimal) phase sequences relatively quickly. Cooper et al. were among the first to apply machine-learning techniques to find good phase orderings to reduce the code size of programs for embedded systems [6]. Genetic algorithms with aggressive pruning of identical and equivalent phase orderings was employed by Kulkarni et al. to make searches for effective optimization phase sequences faster and more efficient [17]. Hoste and Eeckhout proposed a multi-objective evolutionary technique to select a single setting of compiler optimization flags for the entire application [14]. Successful attempts have also been made to use predictive modeling and code context information to focus search on the most fruitful areas of the phase order search space for the program being compiled [1]. Additionally, researchers have used static estimation techniques to avoid expensive program simulations for performance evaluation [7, 24]. Our techniques described in this paper are complementary to all existing approaches and can further improve the solutions delivered by heuristic search algorithms.

Several researchers have investigated the effect of the interdependence between specific pairs of optimization phases on the performance of generated code. For example, researchers have studied the interactions between *constant folding* & *flow analysis*, and *register allocation* & *code generation* in the PQCC (Production-Quality Compiler-Compiler) project [20], between *code generation* & *compaction* for VLIW-like machines [25], as well as between *register allocation* & *code scheduling* [3, 13, 22]. Many of these studies suggested combining specific pairs of phases, if possible. Another complementary study found that implicit application of *register remapping* and *copy propagation* during the phase order search removes phase interactions that do not contribute to quality code and effectively reduces the search space [15]. Also related to our approach are studies that attempt to exploit phase interaction relationships to address the phase ordering problem. These include strategies to use enabling and disabling interactions between optimization phases to automatically generate a single static *default* phase ordering [26, 27], as well as to automatically adapt default phase ordering at runtime for each application [18]. In contrast, in this work we evaluate the potential of phase independence relationships to prune the phase order search space to enable faster exhaustive and heuristic phase order searches.

3. Experimental Framework

In this section we describe our compiler framework, benchmark and experimental setup, and the algorithm we employ for our default exhaustive phase order searches.

3.1 Compiler Framework

The research in this paper uses the Very Portable Optimizer (VPO) [4], which was part of the DARPA and NSF co-sponsored National Compiler Infrastructure project. VPO is a compiler backend that performs all its optimizations on a single low-level intermediate representation called RTLs (Register Transfer Lists). The 15 *reorderable* optimization phases in VPO are listed in Table 1. For each optimization listed in column 1, column 2 presents the code we use to identify that phase in later sections, and column 3 provides a brief description of each phase. Most phases in VPO can be applied repeatedly and in an arbitrary order. Unlike the other

Optimization Phase	Code	Description
branch chaining	b	Replaces a branch/jump target with the target of the last jump in the chain.
common subexpression elimination	c	Performs global analysis to eliminate fully redundant calculations, which also includes global constant and copy propagation.
dead code elimination	d	Removes basic blocks that cannot be reached from the function entry block.
loop unrolling	g	Reduces the number of comparisons and branches at run time and aids scheduling at the cost of code size increase.
dead assignment elim.	h	Uses global analysis to remove assignments when the assigned value is never used.
block reordering	i	Removes a jump by reordering blocks when the target of the jump has only a single predecessor.
loop jump minimization	j	Removes a jump associated with a loop by duplicating a portion of the loop.
register allocation	k	Uses graph coloring to replace references to a variable within a 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 loop nesting level.
code abstraction	n	Performs cross-jumping and code-hoisting to move identical instructions from basic blocks to their common predecessor or successor.
evaluation order determ.	o	Reorders instructions 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 shift, adds, and subtracts.
branch reversal	r	Removes an unconditional jump by reversing a conditional branch when it branches over the jump.
instruction selection	s	Combines pairs or triples of instructions that are are linked by set/use dependencies. Also performs constant folding.
useless jump removal	u	Removes jumps and branches whose target is the following positional block.

Table 1. Candidate Optimization Phases Along with their Designations

Category	Program	Files/	Funcs.	Description
auto	bitcount	10	18	bit manipulation operations
	qsort	1	2	sort strings using quicksort
network	dijkstra	1	6	Dijkstra's shortest path algorithm
	patricia	2	9	patricia trie for IP traffic
telecomm	fft	3	7	fast fourier transform
	adpcm	2	3	compress 16-bit PCM samples
consumer	jpeg	7	62	image compress/decompress
	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 words in phrases
	ispell	12	110	fast spelling checker

Table 2. MiBench Benchmarks Used

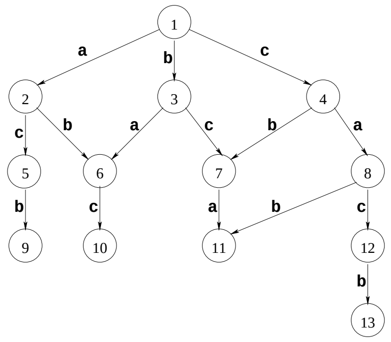


Figure 1. DAG for Hypothetical Function with Optimization Phases `a`, `b`, and `c`

VPO phases, *loop unrolling* is applied at most once in each phase sequence. Our current experimental setup is tuned for generating high-performance code while managing code-size for embedded systems, and hence we use a loop unroll factor of 2. In addition, *register assignment*, which is a compulsory phase that assigns pseudo registers to hardware registers, is implicitly performed by VPO 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 performs *instruction scheduling* before generating the final assembly code.

For our experiments in this paper, VPO is targeted to generate code for the ARM processor running the Linux operating system. We use a subset of the benchmarks from the *MiBench* benchmark suite, which are C applications targeting specific areas of the embedded market [12]. We randomly selected two benchmarks from each category of applications present in MiBench. Table 2 contains descriptions of these programs. More importantly, VPO compiles and optimizes *individual functions* at a time. The 12 selected benchmarks contain a total of 246 functions, out of which 87 are executed (at least once) with the standard *small* input data set provided with each benchmark.

3.2 Setup for Exhaustive Search Space Enumeration

Our research goal is to investigate the impact of techniques that exploit phase independence to prune the phase order search space size, while achieving the best phase ordering performance. As such, we implemented the framework presented by Kulkarni et al. to generate per-function exhaustive phase order search spaces [19]. Our exhaustive phase order searches use all of VPO's 15 reorderable optimization phases. In this section we briefly describe this algorithm.

The algorithm to evaluate per-function exhaustive phase order search spaces generates all possible function instances that can be produced for that function by applying any combination of optimization phases of any possible sequence length (to account for repetitions of optimization phases in a single sequence). This interpretation of the phase ordering problem allows the phase order search space to be viewed as a directed acyclic graph (DAG) of *distinct* function instances. Each DAG is function or program specific. For instance, the example DAG in Figure 1 represents the search space for a hypothetical function and for the three optimization phases, `a`, `b`, and `c`. Nodes in the DAG represent function instances,

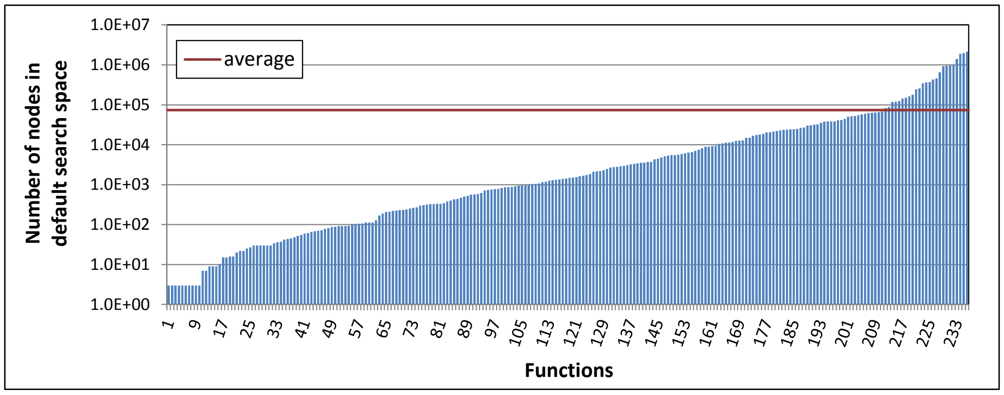


Figure 2. Exhaustive phase order search space size (number of distinct function instances) for all enumerated benchmark functions

and edges represent transition from one function instance to another on application of an optimization phase. The unoptimized function instance is at the root. Each successive level of function instances is produced by applying all possible phases to the distinct nodes at the preceding level. This algorithm uses various redundancy detection schemes to find phase orderings that generate the same *function instance* as the one produced by some earlier phase sequence during the search. Such detection enables this algorithm to prune away significant portions of the phase order search space, and allows exhaustive search space enumeration for most of the functions in our benchmark set with the default compiler configuration. The algorithm terminates when no additional phase is successful in creating a new distinct function instance at the next level.

When restricting each *individual* search time to a maximum of *two weeks*, this algorithm allows exhaustive phase ordering search space enumeration for 236 of our 246 benchmark functions (including 81 of 87 executed functions). We measure the *size* of the phase order search space in terms of the number of *distinct function instances* produced by the exhaustive search algorithm. Figure 2 plots the number of distinct function instances found by the exhaustive search algorithm for each of our 236 enumerated benchmark functions. In this graph, and in similar later graphs, the functions along the X-axis are sorted by the size of their phase order search space and a horizontal line marks the average. Thus, we can see that, for our benchmark functions, the number of distinct function instances found by the exhaustive search algorithm range from only a few to several millions of instances. One of the primary goals of this work is to uncover redundancy in the phase ordering search space and reduce the time for exhaustive phase order searches, while still producing the original best code.

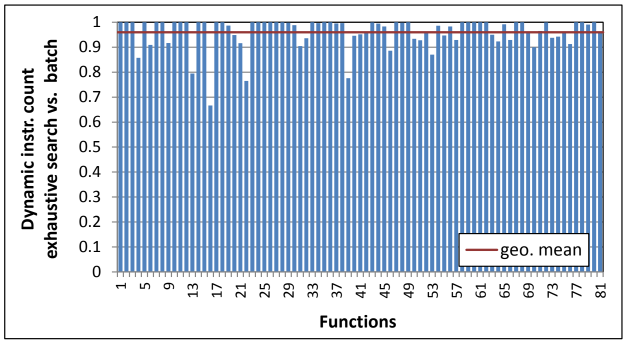
3.3 Setup for Evaluating Code Performance

Our exhaustive search framework uses function-level *dynamic instruction counts* supplemented with whole-program simulated cycle counts (using ARM-SimpleScalar [5]) and actual program run-times (on ARM 'pandaboard' processors) to validate generation of the original best codes with the pruned search spaces. In this section we describe our setup for evaluating generated code performance.

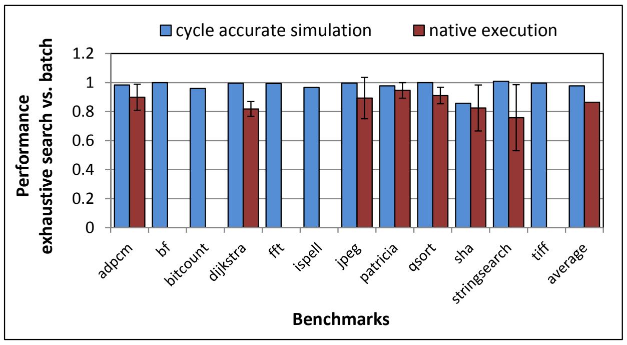
Each per-function exhaustive phase order search space experiment requires the algorithm to evaluate the performance of all the (possibly, millions of) generated distinct function instances to find the best one. It is impractical to perform these evaluations using slow cycle-accurate simulations or with our available ARM hardware resources. Additionally, the validated SimpleScalar cycle-accurate simulator and the native program runs (without intrusive noise-inducing instrumentations) only provide measures over the whole program, and not at the granularity of individual functions. Therefore, our framework employs a technique that generates quick dynamic instruction counts for all function instances, while only requiring a program simulation on generating an instance with a yet unseen *control-flow* [7, 19]. Such function instances with unseen control-flows are instrumented and simulated to determine the number of times each basic block in that control-flow is reached during execution. Then, the dynamic instruction count is calculated as the sum of the products of each block's execution count times the number of static instructions in that block. Researchers have previously shown that dynamic instruction counts bear a strong correlation with simulator cycles for simple embedded processors like the ARM SA-11xx [19]. Figure 3(a) plots the ratio of dynamic instruction count of the best function instance found by exhaustive phase order search to the default (batch) VPO compiler. The default VPO optimization sequence aggressively applies optimizations in a loop until there are no more code changes. In spite of this aggressive baseline, we find that over our 81 executed benchmark functions the best phase ordering sequences improve performance by as much as 33.3%, and over 4.0% on (geometric) average. Throughout the rest of this article, we report the arithmetic mean to summarize raw values and the geometric mean to summarize normalized values [9].

Next, to validate the dynamic instruction count benefit of per-function phase ordering customization, we compile each benchmark program such that individual *executed* program functions are optimized with their respective best phase ordering sequence found by the exhaustive search using dynamic instruction count estimates. We then employ the cycle-accurate SimpleScalar simulator and native execution on the ARM Cortex A9 processor to measure the whole program performance gain of function customization over an executable compiled with the VPO batch compiler. The SimpleScalar cycle-accurate simulator models the ARM SA-1 core that emulates the pipeline used in Intel's StrongARM SA-11xx processors. SimpleScalar developers have validated the simulator's timing model against a large collection of workloads, including the same MiBench benchmarks that we also use in this work. They found that the largest measured error in performance (CPI) to be only 3.2%, indicating the preciseness of the simulators [12]. Our OMAP4460 based pandaboard contains a 1.2GHz dual-core ARM chip implementing the Cortex A9 architecture. We installed a recent release of the Ubuntu Linux operating system (version 10.10) on this board.

The VFP (Vector Floating Point) technology used by the ARM Cortex A9 to provide hardware support for floating-point operations is not opcode-compatible with the SA-11xx's FPA (Floating Point Accelerator) architecture that is emulated by SimpleScalar.Therefore, we were only able to run the seven (out of 12) integer benchmarks on the ARM A9 hardware platform – *adpcm*, *dijkstra*, *jpeg*, *patricia*, *qsort*, *sha*, and *stringsearch*. Additionally,



(a)



(b)

Figure 3. Performance of the default exhaustive phase order search space size. (a) shows the dynamic instruction count of the best function instance found by exhaustive phase order search compared to the default batch VPO compiler, and (b) shows the whole program performance benefit with the SimpleScalar cycle-accurate simulator counts and native execution on ARM A9 processor. The following benchmarks contain floating-point instructions and do not execute on the ARM A9 with VPO generated codes: *blowfish*, *bitcount*, *fft*, *ispell*, and *tiff*.

the Cortex A9 implements an 8-stage pipeline (as opposed to the 5-stage pipeline used by the SA-11xx ARM cores) as well as different different instruction/data cache configurations than those simulated by SimpleScalar. Therefore, it is hard to directly compare the benefit in program execution cycles provided by SimpleScalar with the run-time gains on the ARM Cortex A9 hardware. However, our techniques in this work intend to prune the size of the phase order search spaces without negatively affecting the performance of the best generated codes. Therefore, the native performance results in this paper are still valuable for providing such validation of our techniques on the latest ARM micro-architecture.

The exhaustive phase order search space exploration may find multiple phase sequences (producing distinct function instances) that yield program code with the same *best* dynamic instruction counts for each function. Therefore, for each of these whole program experiments, VPO generates code by randomly selecting one of its best phase sequences for each executed function, and using the default batch sequence for the other compiled program functions. We perform 100 such runs and use the *best* cycle-count / run-time from these 100 runs for each benchmark and experiment. Additionally, while simulator cycle counts are deterministic, actual program run-times are not due to unpredictable hardware, timer, and operating system effects. Therefore, for all the native ARM experiments, we run each generated binary file 61 times (including one startup run), and gather the average run-time and 95% confidence interval over the final 60 runs.

Figure 3(b) plots the ratio of best simulator cycles and program run-time of code generated using the best customized per-function phase sequences over the batch compiler generated code for all our benchmarks. In this graph, and in similar later graphs, the two rightmost bars show the (geometric) average performance with simulator counts and native execution, respectively. Thus, we can see that using customized optimization sequences, whole program processor cycles reduce by up to 16.3%, and 2.3% on average. We emphasize that whole-program cycles / run-time includes the portion spent in (library and OS) code that is not compiled by VPO and not customized using our techniques. This is likely the reason our average whole-program processor cycle-count benefit is lower than the average per-function benefit of customizing optimization phase orderings. We also find that native program run-time on the ARM Cortex A9 processor improves by up to 22% (with *stringsearch*) and 13.9%, on average, with our custom per-function phase sequences over the batch VPO compiler. Thus, customizing optimization phase orderings results in performance gains for many functions and programs in our set of benchmarks.

4. Phase Independence

Our techniques to prune the phase order search space are based on exploiting the independence relationship between optimization phases. Phase independence measures the probability of interdependence between each pair of optimization phases. Two phases can be considered to be independent if their application order does not matter to the final code that is produced. Thus, if an optimization phase is detected to be completely independent of all other phases, then removing it from the set of reorderable phases used during the phase order search space algorithm, and applying it implicitly at any point will make no difference to the final code generated. Our goal for this work is to use this observation to achieve substantial pruning of the phase order search space.

While the interactions between several optimization phases are hard to predict, there are some phases that seem unlikely to interact with each other. For example, *cleanup* phases, such as *dead code elimination* and *dead assignment elimination*, only remove instructions that are either unreachable or calculate useless values. Therefore, it seems reasonable that interchanging the application order of such cleanup phases with any other phase should not affect the final code that is produced. Similarly, interchanging the order of phases that work on distinct code regions and do not share any resources can also be expected to produce identical codes. For example, most phases (in VPO) can be partitioned into control-flow changing phases (*branch* phases) and phases that do not affect the control-flow of the function (*non-branch* phases). Thus, phases such as *branch chaining* or *branch reversal* only affect the branch instructions and no other computational instruction. In contrast, it is uncommon for non-branch phases, such as *instruction selection* and *common subexpression elimination*, to affect the function control flow. Consequently, applying the set of branch and non-branch phases in either order should typically produce the same code. Note that branch phases can still be *enabled* by other non-branch phases by producing code patterns that facilitate their application. However, unlike the issue of phase independence, a purely enabling relationship between phases can be more easily resolved by applying a single ordering of the involved phases in a loop until no phase finds any further code transformation opportunities.

To confirm our intuition regarding the independence of cleanup phases, and the mutual independence of branch and non-branch phases, we calculate pair-wise phase independence relationship [18] between all VPO phases and over all our 236 benchmark functions. Table 3 illustrates this independence relationship for all 15 VPO phases, where each row and column is labeled by an optimization

Phase	b	u	d	r	i	s	o	j	h	k	c	q	l	n	g
b		0.99		0.86	0.95					0.99	0.96		0.98	0.94	0.08
u	0.99				0.68					0.97			0.99		
d															
r	0.86				0.22			0.86			0.99		0.98		
i	0.95	0.68		0.22				0.60		0.98	0.99		0.98	0.99	0.46
s							0.71			0.92	0.80	0.75	0.93	0.93	0.92
o						0.71									
j				0.86	0.60					0.97			0.75		
h										0.85	0.98	0.99	0.92		0.93
k	0.99	0.97			0.98	0.92		0.97	0.85		0.75		0.23	0.96	0.93
c	0.96			0.99	0.99	0.80			0.98	0.75		0.98	0.92	0.79	0.85
q						0.75			0.99		0.98		0.99		0.87
l	0.98	0.99		0.98	0.98	0.93		0.75	0.92	0.23	0.92	0.99		0.85	0.79
n	0.94				0.99	0.93				0.96	0.79		0.85		0.94
g	0.08				0.46	0.92			0.93	0.93	0.85	0.87	0.79	0.94	

Table 3. Independence relationship among optimization phases. Blank cells indicate an independence probability of greater than 0.995.

phase code as indicated in Table 1. The relationship between each pair of phases a and b is represented in Table 3 as a fraction of the number of times application orders a–b and b–a at any function node produced identical codes to the number of times phases a and b were both simultaneously *active* at any node, on average. We call applied phases active when they produce changes to the program representation. Thus, the closer a value in this table is to 1.00, the greater is the independence between that pair of phases.

The phase independence results in Table 3 reveal some interesting trends. First, most phases are highly independent of most other phases. This result is expected and is the primary reason that in spite of the prohibitively large phase order search space size, the search algorithm is able to exhaustively enumerate the search space in most cases. In other words, the high phase independence causes the number of distinct function instances in the search space to be a small fraction of the number of possible phase sequences. Second, very few phases are completely independent of all others. Thus, except for *dead code elimination* (phase 'd') all remaining phases at least interact with four other phases. Third, our intuition regarding *cleanup* phases is also mostly correct. Thus, *dead code elimination* is completely independent, while *dead assignment elimination* (phase 'h') only has small dependencies with a few other phases. Fourth, the table also reveals that the branch and non-branch phases mostly interact with other phases in their own groups, but do have small dependencies with phases in the other group. Most prominently, phases such as *loop unrolling* (phase 'g') in VPO performs both control-flow and non-control-flow changes to the code. The implications of such cross-group phase interactions on our ability to partition the set of reorderable phases is explored in Section 6. In the next two sections we develop various algorithms to exploit our observations regarding phase independence in this section to more effectively prune the phase order search space.

5. Implicit Application of Cleanup Phases During Exhaustive Phase Order Search

Dead code elimination in VPO eliminates unreachable basic blocks, while *dead assignment elimination* removes instructions that generate values not used later in the program. We term these two phases as *cleanup* phases. In this section we perform experiments that exclude cleanup phases from the optimization list used during exhaustive phase order searches. Instead, these phases are now *implicitly* applied after each phase that can potentially produce dead code or instructions. We study the search space size and best code performance delivered by these new search configurations.

Figure 4 compares the number of distinct function instances (search space size) generated with our new exhaustive search configuration (that applies the cleanup phases implicitly) to the default exhaustive phase order search space algorithm. The functions along the horizontal axis in Figure 4 are sorted in ascending order of their original exhaustive search space size. Additionally, the horizontal line in this graph plots the overall geometric mean. Thus, we can see that the phase order search space size with our new search configuration is less than 45% of the original exhaustive search space size, on average. Additionally, we find that functions with larger search spaces witness a greater reduction in the search space size. Indeed, if we aggregate the search space sizes over all benchmark functions and compare this *total* search space size, then we find that our new configuration reduces the space by more than 78% over the default exhaustive configuration. At the same time, our function-level dynamic instruction count measurements reveal that the phase order search space generated for 79 (out of 81) functions by our new search configuration includes at least one function instance that reaches the same best performance as achieved by the original exhaustive search. The best dynamic instruction count reduces for two functions, by 9% and less than 1% respectively, with an average performance loss of 0.1% over the 81 executed benchmark functions. We used our performance validation setup described earlier in Section 3.3 and confirmed that, except for *bitcount* cycle counts, all our benchmark programs achieve the same *best* phase ordering performance (both for program-wide simulator cycles and statistical native runtime on the ARM processor) with and without this configuration of implicit application of cleanup phases. The simulated cycle counts for the benchmark *bitcount* degrade by 1.28%. Over all benchmarks, the *geometric mean* of best program performance ratios with and without implicit application of cleanup phases for simulator cycles and native program run-time is 1.001 and 1.006 respectively. Thus, implicit application of cleanup phases allows massive pruning of the exhaustive phase order search space while realizing the ideal phase ordering performance in most cases.

6. Exploiting Phase Independence Between Sets of Phases

Our second strategy to make exhaustive phase order searches more practical is to partition the set of optimization phases into disjoint sub-groups such that phases in each sub-group interact with each other, but not with any phases in the other sub-group(s). Our knowledge of compiler optimizations in general, and the VPO phases in particular, lead us to believe that *branch* optimization phases should only have minimal interactions with *non-branch* phases. Branch phases (in VPO) only affect the control-flow instructions in the program and do not require registers for successful operation. On

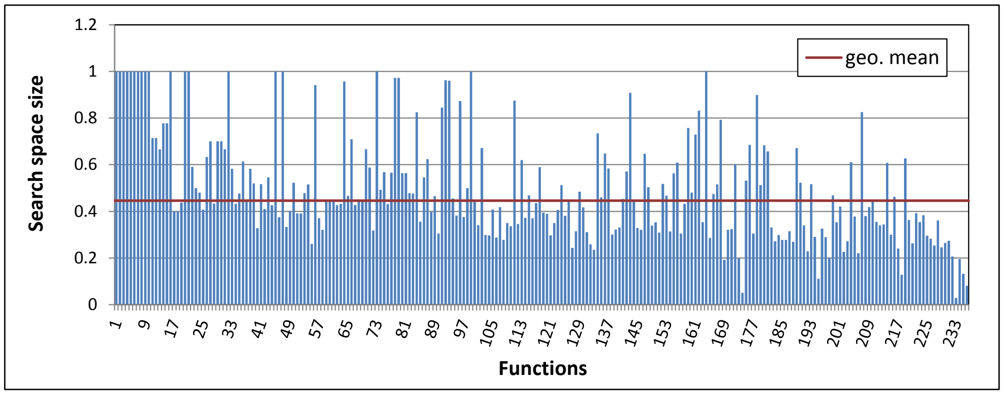


Figure 4. Comparison of search space size (over 236 benchmark functions) achieved by our configuration with cleanup phases implicitly applied to the default exhaustive phase order search space.

the other hand, most non-branch phases do not alter the function control-flow structure, even when they enable more opportunities for branch phases. Our intuition is also backed by the data in Table 3, which shows a high degree of independence between most branch and non-branch optimization phases. *Loop unrolling* is an anomaly since it performs control-flow changes, may require registers, and also updates other computation instructions in the program. In this section, we present our novel *multi-stage* exhaustive search algorithm that employs a partitioned phase set and describe the impact of this algorithm on the search space size and performance of the best code. We then develop techniques that employ partitioned sets of optimization phases to construct a minimal set of good phase orderings and to improve our genetic algorithm based heuristic search.

6.1 Faster Exhaustive Phase Order Searches

For this study, we first partition the VPO optimization phases into two sets, six *branch* phases (*branch chaining*, *useless jump removal*, *dead code elimination*, *branch reversal*, *block reordering* and *loop jumps minimization*), and the remaining nine (*non-branch*) phases. Then, our multi-stage exhaustive search strategy applies the default exhaustive search algorithm over only the branch phases in the first stage, and finds all function instances that produce the best code in this stage. The next stage takes the the best code instances generated by the preceding stage and applies the default search algorithm over only the non-branch phases. Our algorithm is unaffected by the interaction between the branch phases and *loop unrolling*, since *loop unrolling* in VPO is only activated after register allocation, which is applied in the second stage. Since we need to evaluate the function instances to determine the best instances in the first stage to input to the next stage, we conduct these experiments only on our 81 executed benchmark functions. We measure the search space size as the sum of the distinct function instances produced by the algorithm at each stage.

Figure 5 plots the impact of our multi-stage exhaustive search algorithm on search space size as compared to the default exhaustive phase order search space approach. Thus, we can see that the multi-stage configuration reduces the search space size by 76%, on (geometric) average. Additionally, only two out of our 81 executed benchmark functions notice any degradation in the best generated dynamic instruction counts (by 3.47% and 3.84%) for an average degradation by less than 0.1%. Similar to our observations in Section 5, our new configuration has a greater impact on functions with larger default search spaces, reducing the *total* search space size by about 90%.

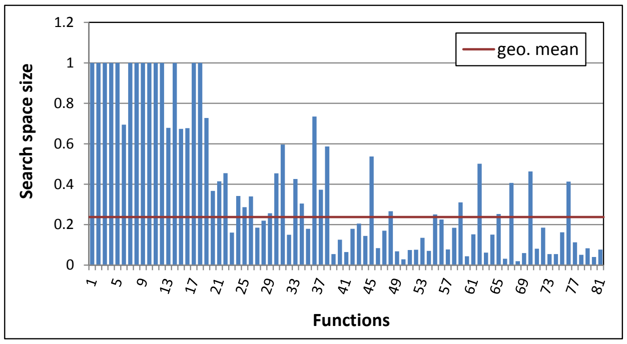


Figure 5. Comparison of search space size achieved by our multi-stage search algorithm as compared to the default exhaustive phase order search space.

We further analyzed the degradation in performance witnessed by the two functions with our multi-stage configuration. We discovered that the degradations occur due to branch phases being enabled during the second stage of our multi-stage configuration. Since our new configuration only applies the branch phases in the first stage, our existing algorithm is unable to resolve these additional opportunities. To address this issue, we changed our multi-stage configuration to perform a search over all phases in the second-stage. The first stage of this new algorithm still performs the search over only the branch phases. Interestingly, this change removes *all* dynamic instruction count degradations, while achieving almost identical (75% per function geometric mean, 88.4% total) search space size reductions. Using our performance validation setup we found that, except for *bitcount*'s simulated cycle counts, all other benchmark programs achieve the same best phase ordering performance (both simulator cycle counts and (statistical) native ARM runtime) with and without this technique of partitioning branch and non-branch optimization phases. The simulator cycle counts for the benchmark *bitcount* degrade by 7%. Over all programs, the *geometric mean* of best program performance ratios (both simulator cycles and native program run-time) with and without this technique of phase partitioning is 1.006 in both cases.

Furthermore, combining our two complementary search space pruning techniques (implicit application of cleanup phases and multi-stage search over partitioned sets of phases) achieves an 89% and 96.75% reduction in the average and total exhaustive search space size, respectively, with the same average dynamic instruction count loss of 0.1% due to the implicit application of cleanup

Seq. #	Covered	Sequence
1	33	b u i r d r o s k s n c s h g
2	11	u r u b r i s k s l c h l s g
3	8	u r u b r i o c s k l s c s h
4	5	b j d i r j c l c s k s h c h
5	4	b u i r d r s h k l s k l c g
6	3	u r u b r i o h s k s l s h g
7	3	b j d i r j s o s c h k s l g
8	2	u r u b r i s o h k s k c s h
9	2	b u i r d r s o k s c l s l c
10	1	u r u b r i o l s c h k c s h
11	1	u r u b r i o c s c s k c s g
12	1	b u i r d r s k s l s l s l s
13	1	b u i r d r s k l c h k c s g
14	1	b u i r d r s k c g l h c s c
15	1	b u i r d r s o s h s k s l c
16	1	b u i r d r s o k l s h l c g
17	1	b u i r d r c s k c l g h c s
18	1	b u i r d r o h l s h k s k s
19	1	b j d i r j s k c k s l c l c

Table 4. Covering Set of Phase Sequences. Optimization phases corresponding to the codes shown here are in Table 1.

phases as seen in Section 5. Thus, we can conclude that exploiting optimization phase independence can have a huge impact in making exhaustive phase order searches to find the best phase ordering more feasible and practical in existing compilers.

6.2 Finding a Covering Set of Phase Sequences

Although earlier works have shown that no single order of optimization phases can achieve the best performance for all functions [6, 24, 26], it is not yet clear if a *small* number of phase orderings can obtain near-optimal performance for every function. In this section we present novel schemes that use our observations regarding independent sets of phases to investigate this question.

The typical phase order search space is exorbitantly large, and prohibits any naïve attempt at evaluating all possible optimization phase orderings of any length. For example, the longest possible *active* sequence length during the exhaustive phase order search over our benchmark functions is 37 phases. Thus, in the worst case, any algorithm to determine the set of best phase sequences must consider 15^{37} sequences. Our exhaustive search algorithm prunes away much of the redundancy in this search space. Every path in each phase order search space DAG corresponds to a phase sequence that produces a distinct function instance (corresponding to the final node in the path). Unfortunately, even with this reduction, the combined set of paths in the search space for our benchmark set is still too large (over $1.8 * 10^{16}$ total paths) to search completely.

Interestingly, in the previous section, we show that partitioning the phase order search space over sets of branch and non-branch phases does not impact the best performance achieved by the exhaustive phase order search. In this section, we use our reduced search space from Section 6.1 to develop techniques that construct a small set of phase sequences that achieve close-to-optimal performance for all our functions.

The first step in our algorithm is to find the best branch-only phase orderings over our benchmark functions. We express the problem of finding the minimal number of the best branch-only phase orderings as the classical set-cover problem [8]. Our compiler includes six branch phases, so we apply every possible branch-only sequence of length six to each of our benchmark functions and evaluate the code produced. Our technique generates a set of functions for each branch-only sequence, where a function is in a

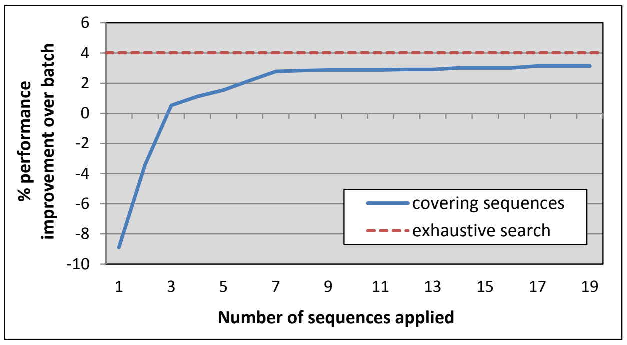


Figure 6. Average performance improvement over the default compiler after applying *n* of the 19 sequences listed in Table 4 to each benchmark function. The dashed line shows the average improvement achieved by the individual best phase ordering sequences (found by our exhaustive search algorithm) over the default compiler performance.

particular sequence's set if that sequence achieves the best *branch-only solution* for that function. The set-cover problem is to find the minimum number of branch-only phase orderings such that their corresponding sets cover all benchmark functions. Since this problem is NP-hard, we use a greedy algorithm to approximate the set cover. We find that only three branch sequences are needed to reach the best branch-only performance in all our 81 functions (a single sequence covers 85% of the functions).

The next step is to combine these three branch-only sequences (of length 6) with sequences consisting of the nine non-branch phases. We generate all non-branch sequences of length nine and append each to the three branch-only sequences (for a total of $3 * 9^9$ candidate sequences *of length 15*). We then generate sets of benchmark functions that reach the best performance with each sequence (similar to our approach with the branch-only sequences). Applying the greedy set-cover algorithm to these sets yields 19 sequences that are needed to reach the best phase-ordering performance for all 81 functions. It is important to note that this algorithm restricts each sequence length to 15 phases. These 19 sequences are shown in Table 4. The first column ranks the sequence, the second lists the number of functions covered during the set covering algorithm by each sequence, and the third column presents the actual sequence.

We evaluate the performance delivered by our covering sequences by incrementally applying them in descending order of the number of functions they cover. We also employ the standard *leave-one-out cross validation* scheme, which ensures that sequences that cover only a single benchmark function are not used to evaluate that function. This experiment applies a certain number (n) of our covering sequences *one at a time* in the order shown in Table 4, and reports the best performance of each program for any of the n sequences. For each point along the X-axis, the plot in Figure 6 averages and compares this best performance (over all 81 executed functions) achieved by the n covering sequences with the average performance obtained by the default (batch) compiler. In the batch mode, VPO applies a fixed ordering of optimization phases in a loop repeatedly until no phase is able to make any further changes to the code. Thus, the batch compiler sequence is not restricted to a length of 15, and therefore achieves better performance than any of our individual 19 phase sequences. However, even with this aggressive batch sequence, only three covering sequences are able to improve performance (by 0.5%) over the batch compiler. Applying all 19 sequences yields an average speedup of 3.1%. Additionally, we use our performance validation setup (described in Section 3.3) to evaluate our covering sequences. Figure 7 compares the full pro-

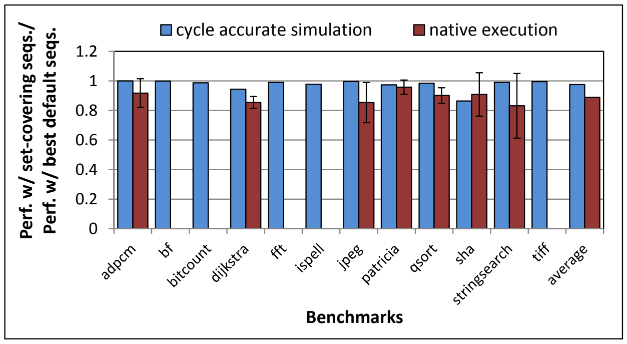


Figure 7. Whole program performance benefit of using the best code generated by the 19 sequences listed in Table 4 over VPO batch compiler with SimpleScalar cycle-accurate simulator counts and native execution on ARM A9 processor.

gram simulation and native performance of code compiled with the batch sequence to the best run with randomly chosen (leave-one-out) covering sequences for each benchmark function (out of 100 such runs). Thus, the covering sequences either improve or achieve the same full-program simulation cycle count as the batch compiler sequence, with a maximum improvement of 13.5% (*sha*) and an average improvement of 2.5%. The native program run-time on the ARM Cortex A9 improves as well by up to 16.8% (with *stringsearch*), and yields an 11.2% average improvement. Thus, the covering sequences significantly improve performance over the default compiler sequence for our benchmark programs.

6.3 Better Genetic Algorithm Searches

Finally, we explore if our observations regarding search space pruning by exploiting phase independence relationships can improve heuristic search techniques. Machine-learning based genetic algorithms (GA) [21] are a popular mechanism to develop such heuristic search techniques. Unlike the exhaustive searches, GA-based searches do not guaranty reaching the optimal phase ordering performance, but often produce quite effective phase ordering sequences. Therefore, we adapt a variant of a popular GA-based search technique for our experiments in this section [6]. *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. 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 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. Our genetic algorithm uses 20 sequences (*chromosomes*) per generation, and iterates the GA for 200 generations. The *batch* VPO compiler aggressively applies hundreds of optimization phases to the code, a fraction of which are actually successful (or *active*) in changing the code. Thus, the number of active batch phases can vary for every function. We use 1.25 times this active batch sequence length as the number of phases in each chromosome of the GA with a minimum chromosome length of 20. The

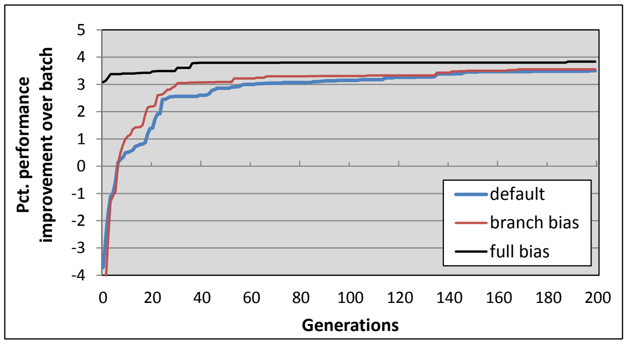


Figure 8. Improvement achieved by the default and biased GA configurations over the batch VPO performance in each generation

fitness criteria used by the GA searches is the dynamic instruction counts of the code generated by each chromosome.

Our aim here is to employ the covering sets of phase sequences (from Section 6.2) to expedite and assist the GA-based search. We develop two new configurations that use the covering sets to bias chromosomes in the first generation. In each configuration, we again employ cross-validation so that the branch and covering sequences are only generated from the training data that uses all programs except the one to be optimized. In the *branch bias* configuration, chromosomes in the first generation of our modified GA are produced by prepending one of the three set-covering branch phase orderings (of length six) to a randomly generated phase sequence of length *n* - 6, where *n* is the length of each chromosome in the default GA configuration. Each of the set-covering branch sequences is used at least once, and the prefixes to the remaining chromosomes are chosen depending on the ratio of benchmark functions each branch sequence covers. In the *full bias* configuration, chromosomes are produced by prepending each of the 19 set-covering sequences consisting of both branch and non-branch phases (shown in Table 4) to randomly generated phase sequences of length $n - 15$. The last chromosome in the initial population is randomly generated. The rest of the GA proceeds as in the original algorithm.

Figure 8 compares the improvement achieved by the best chromosome in each generation over the batch VPO performance for each of the default and modified genetic algorithms, averaged over all 81 executed benchmark functions. Thus, we can see that the branch-bias configuration is able to focus the search to the more useful sections of the search space sooner and allows the GA to converge on its final solution faster than the original algorithm. The full bias GA shows clear improvements over the other configurations because the set-covering sequences used to bias this configuration achieve very good performance by themselves. At the same time, it is also interesting to note that the genetic algorithm is powerful enough to further improve the performance of the full bias configuration after the first generation. Of course, several other changes to the GA (and to other heuristic algorithms) that attempt to exploit the phase independence relationship are possible and may show varying improvements. However, our results show that exploiting phase interactions is a fruitful direction to improve future heuristic algorithms for addressing the phase ordering problem.

7. Future Work

There are several avenues for future work. First, we plan to investigate new algorithms that will employ the independence and other phase relationships to *automatically* partition the phase set into smaller subsets. Second, we will explore the impact of our techniques on reducing the search time for heuristic algorithms besides GAs. We also plan to study the benefits of combining our techniques with the strategy of using function characteristics to focus heuristic searches. Third, we wish to implement our techniques in other compilers to validate their broader applicability and understand the effect of phase implementation on the resulting search space size and performance benefits. Finally, we will attempt to devise algorithms to find the minimal set of optimization sequences, one of which will achieve optimal phase ordering performance for all functions.

8. Conclusions

The long-standing problem of optimization phase ordering exists due to hard-to-predict interactions between phases. The primary contribution of this work is suggesting and validating the hypothesis that not all optimization phases interact with each other, and that it is possible to develop mechanisms that exploit this observation to substantially improve the exhaustive and heuristic phase order search times, while almost always delivering the same best phase ordering solutions. Our first technique employs the independence of cleanup phases to reduce the exhaustive search space by a total of 78% and over 55%, on average. Only 2 of 81 benchmark functions do not generate the same best code instance, and the average performance loss is 0.1%. Our second technique develops a novel multi-stage exhaustive search strategy over independent sub-groups of branch and non-branch phases. This technique prunes the search space size by about 75%, on average, (88.4% total) with no loss in best phase ordering performance. *Together, our techniques achieve a 89% reduction in average search space size, and a 96.75% reduction in the total search space, combined over all functions.*

We also develop new algorithms that employ our phase independence observations to find a small set of near-optimal phase sequences and to improve genetic algorithm performance. Our techniques presented in this paper are simple and general enough to be applicable and effective in several other compiler frameworks. Most compilers are likely to have at least a few "cleanup" phases. Similarly, we also expect branch and non-branch phases in several other compilers to display similar independence relationships as seen in VPO. Thus, our results show that the phase ordering search overhead can be significantly reduced, perhaps to a tolerable level, by exploiting specific relationships between optimization phases.

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 *Symposium on Code Generation and Optimization*, pages 295–305, 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 *Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 231–239, 2004.

[3] W. Ambrosch, M. A. Ertl, F. Beer, A. Krall, M. Anton, E. Felix, and B. A. Krall. Dependence-conscious global register allocation. In *In proceedings of PLSA*, pages 125–136. Springer LNCS, 1994.

[4] 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, 1988.

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

[6] 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, 1999.

[7] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Acme: adaptive compilation made efficient. In *Conference on Languages, compilers, and tools for embedded systems*, pages 69–77, 2005.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, September 2001.

[9] P. J. Fleming and J. J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, Mar. 1986.

[10] G. Fursin, Y. Kashnikov, A. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. Williams, and M. OBoyle. Milepost GCC: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39:296–327, 2011.

[11] 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.

[12] 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.

[13] J. L. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, 1983.

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

[15] M. R. Jantz and P. A. Kulkarni. Analyzing and addressing false interactions during compiler optimization phase ordering. *Software: Practice and Experience*, 2013.

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

[17] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Conference on Programming Language Design and Implementation*, pages 171–182, June 2004.

[18] 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.

[19] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson. Practical exhaustive optimization phase order exploration and evaluation. *ACM Transactions on Architecture and Code Optimization*, 6(1):1–36, 2009.

[20] 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.

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

[22] S. S. Pinter. Register allocation with instruction scheduling. In *Conference on Programming language design and implementation*, pages 248–257, 1993.

[23] S.-A.-A. Touati and D. Barthou. On the decidability of phase ordering problem in optimizing compilation. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 147–156, 2006.

[24] 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, 2003.

[25] 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.

[26] D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. In *Symposium on Principles & Practice of Parallel Programming*, pages 137–146, 1990.

[27] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6): 1053–1084, 1997.