SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2011; **00**:1–36
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Analyzing and Addressing False Interactions During Compiler Optimization Phase Ordering*

Michael R. Jantz and Prasad A. Kulkarni

Department of Electrical Engineering and Computer Science, University of Kansas, Lawrence, Kansas, USA

SUMMARY

Compiler optimization phase ordering is a fundamental, pervasive and longstanding problem for optimizing compilers. This problem is caused by interacting optimization phases producing different codes when applied in different orders. Producing the best phase ordering code is very important in performance-oriented and cost-constrained domains, such as embedded systems. In this work we analyze the causes of the phase ordering problem in our compiler, VPO, and report our observations. We devise new techniques to eliminate, what we call, *false* phase interactions in our compiler. We find that reducing such false phase interactions significantly prunes the phase order search space. We also develop and study algorithms to find the best average performance that can be delivered by a single phase sequence over our benchmark set, and discuss the challenges in resolving this important problem. Our results show that there is no single sequence in VPO that can achieve the optimal phase ordering performance across all functions. Copyright © 2011 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Phase ordering, False phase interaction, Exhaustive search

1. INTRODUCTION

Finding the best set and ordering of optimization phases is a fundamental, pervasive and long-standing problem in optimizing compilers. Current compilers contain several different optimization phases. Every phase attempts to apply a series of *transformations*, that consist of a sequence of changes that preserve the semantic behavior of the program while typically improving its efficiency. Many of these optimization phases use and share resources (such as machine registers), and may also need specific patterns in the code to be applicable. As a result, optimization phases interact with each other by enabling and disabling opportunities for successively applied phases. In earlier compiler optimization research, such phase interactions were widely studied [2, 3, 4, 5]. These studies show that phase interaction can often cause different orders of applying optimization phases to produce different output codes, with significant performance variations. The best phase application sequence depends on the program being optimized, the characteristics of the target machine, and the manner in which the optimizations are implemented in the compiler. At the same time, the potentially large performance difference between the codes produced by different phase sequences can have major implications on the cost (e.g., memory size) or power requirements of the application. Such

*This work extends our earlier conference submission, titled *Eliminating False Phase Interactions to Reduce Optimization Phase Order Search Space*, published in the ACM conference on Compilers, architectures and synthesis for embedded systems (CASES), 2010 [1]. We extend this earlier work by conducting a more thorough investigation with additional benchmarks, validation for many of our earlier experiments with SimpleScalar cycle-accurate simulation and native runs of the latest ARM hardware, more detailed analysis of the results, and re-evaluating and updating our description and conclusions. We also perform new studies (entire Section 7) in this paper that have never been reported.

implications make finding a good ordering of optimization phases very attractive in performance-critical application domains such as embedded systems.

Most conventional compilers are plagued with this fundamental problem of determining the ideal sequence of optimization phases to apply to each function or program so as to maximize the gain in either speed, code-size, power, or any combination of these performance constraints. With more and more compiler optimization phases being implemented in conventional compilers, it is becoming increasingly difficult to keep track of all possible ways in which different phases might interact, and their effects on program performance. Most existing approaches to address this phase ordering problem are based on the premise that the task of deciphering and eliminating the actual phase interactions is highly daunting, and may be impossible in the presence of the large number of phase combinations, programs, program inputs, and architectural contexts that might need to be considered. Thus, increasingly, individual phases are designed with minimal regard to the phase ordering problem, and the issue of the most appropriate phase sequence is dealt with as an afterthought at the end of the compiler design process. Empirical search algorithms are commonly used to iterate over possible phase sequences or orderings for each input program, applying, evaluating, and comparing each sequence with all others to find the best one [6, 7, 8, 9, 10, 11, 9, 12]. Unfortunately, optimization phase ordering/selection search spaces in current compilers have been reported to consist in excess of 15^{32} [13], or 16^{10} [14], or 2^{60} [15] unique phase sequences, making exhaustive phase order searches highly time-consuming and impractical. Consequently, current research to resolve the phase application problem is mainly focused on studying novel machine-learning-based algorithms and search heuristics to more intelligently traverse the large phase ordering search spaces.

We believe that such existing closed-box approaches have several drawbacks. Lack of proper consideration of the phase ordering problem during the design of optimization phases at compiler construction time may result in implementations that further increase the phase interactions. Additionally, as compilers keep increasing their set of optimization phases, search algorithms may be required to operate with increasing search space sizes in the future. Empirical searches are already time-consuming, even with probabilistic and machine-learning heuristics. Their high cost restricts the applicability of iterative searches to certain domains that permit time-consuming static compilation, and larger search spaces only exacerbate this problem. Finally, treating compiler phases as black boxes provides no clue or guidelines to compiler developers on how to implement future optimizations so as to minimize their interactions with other phases. Availability of such guidelines will strike at the root of the phase ordering problem, and likely benefit all current and future approaches to address this problem in compilers. These factors motivated us to study, understand and, if possible, resolve the most prominent optimization phase interactions in our compiler, VPO (Very Portable Optimizer) [3]. Eliminating the most important phase interactions can prune the phase order search space, allowing existing and new search techniques to more quickly find the best phase ordering solutions. In this paper, we report our observations from this study.

Phase interactions due to the limited number of available registers and high register pressure in many programs has been observed in earlier works, and is considered as one of the most prominent causes of the phase ordering problem [3, 16]. Our analysis of the most significant phase interactions confirms the importance of architectural registers during optimizations in our compiler backend. However, deeper analysis revealed that many phase interactions are not caused by register contention, but exist due to the dependences between phase transformations that reuse the same register *numbers*. We term such dependences as *false* phase interactions. We explore the extent and impact of phase interactions due to false register dependences on our phase order search space size and generated code performance. We then devise novel mechanisms to minimize the false register dependences and evaluate their impact on the phase order search space and quality of generated code during conventional compilation.

Several researchers also believe that it is unlikely for a single set or ordering of optimization phases to produce optimal code for every application [17, 10, 11, 9, 6]. However, no previous study has attempted to evaluate the performance of *all* possible phase sequences over a large benchmark set to validate this belief, or to find the single best optimization phase sequence over a

DOI: 10.1002/spe

large benchmark set. Therefore, in this work, we also explore this problem of automatically finding the best average performance that can be achieved by any single compiler phase sequence in our compiler, and discuss the challenges in addressing this issue. Our study empirically shows the non-existence of a single phase sequence that can achieve optimal code for all functions.

Although phase ordering is a pervasive issue across all or most compilers, we must necessarily restrict this study to a single compiler. The presence of an exhaustive framework for phase ordering research, along with our intimate knowledge of its internal organization made VPO the best choice for us to conduct this research. However, we believe that our motivation and methodology for exploring and addressing phase interaction issues should be more generally applicable to other compilers. Most of all, we believe that even though different compilers may have different optimization phases, implementations and phase interactions, this study shows the potential and benefit of understanding such interactions to guide future implementation decisions that minimize the phase ordering problem and generate higher-quality code in compilers. The major contributions of this research work are as follows:

1. This is the first research that provides the methodology and presents the benefits of analyzing and alleviating the optimization phase interactions to reduce the phase order search space and improve code quality.
2. We show that the problem of false phase interactions is a significant contributor to the size of the phase order search space.
3. We develop and evaluate techniques that reduce the discovered false phase interactions to substantially prune the phase order search space in our compiler VPO.
4. We study techniques to find empirical limits on the best performance that can be achieved by any single phase sequence for any benchmark set.

The rest of the article is organized as follows. We present our experimental framework in the next section. We explain our observations regarding the effect of register availability on phase interactions in Section 3. In Section 4 we show that the effects of false register dependence are often independent of register pressure issues. In Section 5 we develop mechanisms to reduce false register dependence for a hypothetical ARM-based machine with virtually unlimited registers. We adapt these techniques for use on real embedded ARM architectures in Section 6. In Section 7 we develop new mechanisms that employ our analysis of the phase order search space to improve code quality of conventional compilation. We describe related work in Section 8. We list avenues for future research in Section 9, and present our conclusions in Section 10.

2. EXPERIMENTAL SETUP

In this section we describe our compiler framework and the setup employed to perform our studies.

2.1. Compiler Framework

The research in this paper uses the Very Portable Optimizer (VPO) [3], which was a part of the DARPA and NSF co-sponsored National Compiler Infrastructure project. VPO is a compiler back end that performs all its optimizations on a single low-level intermediate representation called RTLs (Register Transfer Lists). Since VPO uses a single representation, it can apply most analysis and optimization phases repeatedly and in an arbitrary order. VPO compiles and optimizes one function at a time. This is important for the current study since restricting the phase ordering problem to a single function, instead of the entire file, helps to make the optimization phase order search space more manageable. At the same time, customizing optimization phase sequences to individual functions instead of the entire program has been observed to provide better performance results [18]. VPO has been targeted to produce code for a variety of different architectures. For this study we used the compiler to generate code for the ARM 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

DOI: 10.1002/spe

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.
unreachable code elimination	d	Removes basic blocks that cannot be reached from the function entry block.
loop unrolling	g	To potentially reduce the number of comparisons and branches at run time and to aid scheduling at the cost of code size increase.
dead assignment elimination	h	Uses global analysis to remove assignments when the assigned value is never used.
block reordering	i	Removes a jump by reordering 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 determination	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 I. VPO Optimization Phases

with the low-power consumption of ARM-based processors have made this ISA dominant in the embedded systems domain.

The 15 *reorderable* optimization phases currently implemented in VPO are listed in Table I. Most of these phases can be applied repeatedly and in an arbitrary order. Unlike the other VPO phases, loop unrolling is applied at most once. The VPO compiler is tuned for generating high-performance code while managing code-size for embedded systems, and hence uses a loop unroll factor of 2. In addition, *register assignment*, which is a compulsory phase that assigns pseudo registers to hardware registers, is automatically 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 also performs *instruction scheduling* before generating the final assembly code.

For this work we use a subset of the benchmarks from the *MiBench* benchmark suite, which are C applications targeting specific areas of the embedded market [19]. We randomly selected two benchmarks from each of the six categories of applications present in MiBench. The same benchmarks were also used in previous phase ordering studies with VPO [20, 13]. Table II contains descriptions of these programs. As noted earlier, VPO compiles and optimizes individual functions at a time. The 12 selected benchmarks contain a total of 244 functions, out of which 87 are executed with the standard input data provided with each benchmark.

2.2. *Algorithm for Exhaustive Search Space Enumeration*

Our goal in this research is to understand the effect of false phase interactions on the size of the phase order search space. This effect can be most clearly demonstrated by the reduction in the exhaustive phase order search space that can be achieved for each benchmark. Additionally,

DOI: 10.1002/spe

Category	Program	#Lines	Description
auto	bitcount	584	test processor bit manipulation abilities
	qsort	45	sort strings using the quicksort sorting algorithm
network	dijkstra	172	Dijkstra's shortest path algorithm
	patricia	538	construct patricia trie for IP traffic
telecomm	fft	331	fast fourier transform
	adpcm	281	compress 16-bit linear PCM samples to 4-bit samples
consumer	jpeg	3575	image compression and decompression
	tiff2bw	401	convert color *tiff* image to b&w image
security	sha	241	secure hash algorithm
	blowfish	97	symmetric block cipher with variable length key
office	string-search	3037	searches for given words in phrases
	ispell	8088	fast spelling checker

Table II. MiBench Benchmarks Used in the Experiments

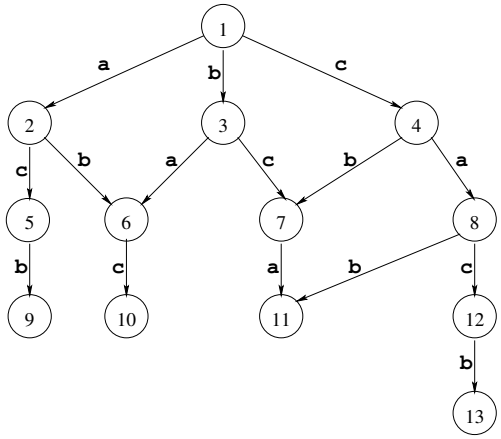


Figure 1. DAG for Hypothetical Function with Optimization Phases `a`, `b`, and `c`

exhaustive phase order searches are crucial to: (a) better explore and understand the properties of the search space [21, 14, 12], (b) determine the *ideal* optimization phase ordering(s) for a given benchmark-input pair [22, 13], and (c) quantify the effectiveness of machine-learning and model-driven search approaches in realizing this *ideal* performance [20, 21, 14]. We implement the framework presented by Kulkarni et al. [13] to generate per-function exhaustive phase order search spaces over all of VPO's 15 reorderable optimization phases. In this section we describe this algorithm to generate exhaustive phase order search spaces.

A simple approach to enumerate the exhaustive phase order search space would be to generate (and evaluate the performance of) all possible combinations of optimization phases. This approach is clearly intractable and naïve since it does not account for the fact that many such sequences may produce the same code (also called *function instance*). Another way of interpreting the phase ordering problem is to enumerate all possible function instances that can be produced by any combination of optimization phases for any possible sequence length (to account for repetitions of optimization phases in a single sequence). Such an interpretation makes the problem of exhaustive phase order enumeration much more practical by exploiting the redundancy of several phase sequences generating identical code.

The phase order search space can now be viewed as a directed acyclic graph (DAG) of *distinct* function instances. [†] Each DAG is function or program specific, and may be represented as in

[†]In theory, compiler optimizations can undo changes made by preceding phases and introduce back-edges in the directed graph. However, for our set of 244 benchmark functions, the exhaustive search algorithm only produced "acyclic" graphs

DOI: 10.1002/spe

Figure 1 for a hypothetical function and for the three optimization phases, `a`, `b`, and `c`. Nodes in the DAG represent function instances, 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. It is assumed in Figure 1 that no phase can be successful multiple times consecutively without any intervening phase(s) in between. This algorithm computes multiple hash-values for each function instance, including CRC (cyclic-redundancy code) checksum on the bytes of the RTLs in that function. On generating a new function instance, these hash-values are used to quickly compare that instance with all previously generated code instances to find and eliminate phase orderings that generate the same function instance as the one produced by some earlier phase sequence during the search [13]. This comparison for detecting redundant (previously-seen) function instances check for *identical* as well as *equivalent* instances. Two function instances are deemed *identical* only if their codes match in every respect. Since this is a stricter requirement than necessary, the algorithm also checks for function instances that are *equivalent*, in regard to speed and size, but not identical. We detect this situation by mapping all register live ranges in a function to distinct pseudo-registers. Thus, equivalent function instances become identical after mapping. Eliminating identical and equivalent function instances enables this algorithm to prune away large 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.

Thus, this approach can make it possible to generate/evaluate the entire search space, and determine the *optimal* phase ordering solution. It is also interesting to note that this exhaustive solution to the phase ordering problem subsumes the related issue of *phase selection*, which deals with deciding what transformations to apply without considering their order [23, 7, 15]. Any phase sequence of any length from the phase order/selection search space can be mapped to a node in the DAG of Figure 1. This space of all possible *distinct function instances* for each function is, what we call, the *actual* optimization phase order search space, and the *size of each search space is measured as the number of nodes in this DAG*. We restrict the exhaustive phase order search for each individual function to a maximum of *two weeks*. With this restriction, the algorithm allows exhaustive phase ordering search space enumeration for 234 of our 244 benchmark functions (including 81 of 87 executed functions). Note that our techniques to prune the phase order search space discussed in this paper will enable more effective and quicker resolution of all (including the larger) search spaces. The number of distinct function instances found by the exhaustive search algorithm range from a few tens to several millions of instances. The time to explore and evaluate the entire exhaustive phase order search space is directly proportional to the number of distinct nodes in the search space DAG. All our search space comparisons in this paper evaluate the reduction in the number of nodes of the exhaustive search space DAG of the unmodified compiler that is accomplished by each technique. Thus, the goal of all our phase order search space reduction techniques is to enable distinct phase order sequences to converge to the same node (function instance) in the search space DAG by eliminating (false) interactions between corresponding phases.

2.3. Dynamic Performance Measurements

Each per-function exhaustive phase order search space experiment requires the algorithm to evaluate the performance of all generated distinct function instances to find the best one. As noted, such exhaustive experiments can generate millions of distinct function instances in several cases. Thus, even though native execution of the benchmarks on actual ARM processors to measure the dynamic run-time for each distinct function instance would be ideal, we did not have access to enough ARM machines to make such a native evaluation feasible for all our experiments. In contrast, when using simulation, multiple individual function-level searches can be performed in parallel on our cluster of high-performance x86 machines. ARM processors are also considerably slower

(no back-edges), and so we call them DAGs in this paper. The acyclic nature of the graphs is not a requirement for the exhaustive search algorithm.

DOI: 10.1002/spe

than state-of-the-art x86 machines, which increases the time spent during compilation even as it reduces the program execution time (compared to simulation). Therefore, performing hundreds of long-running experiments was impractical for us to arrange with the available (ARM) hardware resources. We use the SimpleScalar set of functional and cycle-accurate simulators [24] for the ARM to *estimate* dynamic performance measures. To our knowledge, SimpleScalar provides the most advanced and popular open-source simulators for the ARM processors. 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 [19].

Even invoking the cycle-accurate simulator for estimating the performance of every distinct phase sequence produced by the search algorithm is prohibitively expensive. Additionally, the validated SimpleScalar cycle-accurate simulator only provides whole-program performance results, that are not at the granularity of individual benchmark functions. Therefore, we have adopted another technique that can provide quick (and micro-architecture independent) *dynamic instruction counts* for all function instances with only a few program simulations (with the faster functional SimpleScalar simulator) per phase order search [25, 13]. Researchers have previously shown that dynamic instruction counts bear a strong correlation with simulator cycles for simple embedded processors like the ARM SA-11xx [13]. In this scheme, program simulation is only needed on generating a function instance during the exhaustive search with a yet unseen *control-flow*. Such function instances are then instrumented using the EASE instrumentation framework [26]. On program execution / simulation, the added instrumentations output the number of times each basic block in that control-flow is reached during execution. Then, dynamic performance is calculated as the sum of the products of each block's execution count times the number of static instructions in that block. The VPO compiler has also been updated to track changes made by optimizations to the control-flow graph, like reversing branch conditions, to accurately detect all distinct control-flows. Later, we use the whole-program SimpleScalar cycle-accurate simulator to validate our techniques by computing and comparing the run-time processor cycles of only the *best* generated codes delivered by the dynamic instruction count estimates for several experiments in this paper.

The SimpleScalar simulator we employ models the very old ARM-SA11xx (micro) architecture core. Although accurate for this older architecture, this simulator does not guarantee precise performance estimates for newer architectures. Therefore, we also employ the latest ARM "pandaboard" to validate many of our benchmark-wide *best* performance results. Our OMAP4460 based pandaboard contains a 1.2Ghz dual-core ARM chip implementing the Cortex A9 architecture. We installed the latest Ubuntu Linux operating system (version 10.10) on this board. The Cortex A9 implements an 8-stage pipeline, which is, unfortunately, very different from the 5-stage pipeline used by the SA-11xx ARM cores. Similarly, the Cortex A9 also uses 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 are primarily concerned with reducing the size of the phase order search spaces without negatively affecting the performance of the best generated function codes. Therefore, we believe that the ARM A9 results in this paper are still valuable for providing such validation of our techniques on the latest ARM micro-architecture.

Additionally, 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 (and hence used by VPO). 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*. We also observed that inserting timer instrumentations in the source code to collect function-level run-time measures introduces significant noise for most program functions in our benchmark set. Therefore, in this work we only use timer instrumentations at the start and end of the `main` function, and only collect whole-program run-times on the ARM hardware. In summary, we use the simple and fast dynamic

DOI: 10.1002/spe

instruction counts to estimate the best function-level phase orderings for our various experimental configurations, and then validate the performance of programs generated using only these best per-function optimization sequences with the SimpleScalar cycle-accurate simulator and on the latest ARM Cortex A9 processor hardware.

2.4. Baseline Exhaustive Phase Order Search Space Results

Table III in Appendix A presents various statistics for our MiBench benchmark functions and the exhaustive phase order search space experiments that are used as a baseline for this work. For each program in our benchmark suite, Table III provides its constituent functions in the first column sorted in ascending order of its code-size listed in the next column. We measure the function code-size in terms of the number of RTLs in the unoptimized function instance. For each function, column 3 presents the dynamic instruction count for code generated using the default VPO *batch* compiler. The batch VPO compiler applies a fixed order of optimization phases in a loop until there are no additional changes made to the program by any phase, and thus provides a very aggressive baseline. An empty cell in this column indicates that the corresponding function is not reached during execution with the standard input provided with MiBench. Column 4 presents per-function phase order search space size (number of distinct function instances in the search space DAG). An empty cell denotes that the function phase order search space is too big to exhaustively explore using our current algorithm stopping criteria (search time exceeding two weeks). The search time is directly proportional to the search space size. Note that, the goal of this work is to understand and eliminate false phase interactions to reduce the phase order search space size and make it possible to enumerate even the larger function search spaces in reasonable time. Finally, column 5 in Table III lists the percentage reduction in dynamic instruction count over the batch compiler performance for functions compiled using the best phase ordering provided by the per-function exhaustive phase order search space evaluation experiments. Thus, estimations provided by dynamic instruction counts reveal that customizing optimization phase orderings improves performance by as much as 33%, with a geometric mean of over 4% over our 81 executed benchmark functions. Note that, in the final row of Table III, and throughout the rest of this article, we report the arithmetic mean to summarize raw values and the geometric mean to summarize normalized values [27].

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) and the remaining functions with the batch VPO optimization sequence. We then employ the cycle-accurate SimpleScalar simulator and native execution on the ARM A9 processor to measure the whole program performance gain of function customization over an executable compiled with the VPO batch compiler. However, our 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. We choose to measure the *best* run-time since the goal of the exhaustive phase order searches is to find the ideal phase ordering for each function/benchmark. Our experimental setup allows the scenario where quicker (but less precise) simulation results are used to search the entire phase order space, followed by a limited set of runs with only the "potentially best" sequences on the real embedded (ARM) hardware to find the actual ideal performance. Additionally, while simulator cycle counts are deterministic, actual program run-times are not due to unpredictable hardware, timer, and operating system (OS) effects. Therefore, for all native ARM experiments we run each program 61 times (including one startup run), and report the average run-time over the final 60 runs.

Figure 2 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. Thus, we can see that using customized optimization sequences, whole program

DOI: 10.1002/spe

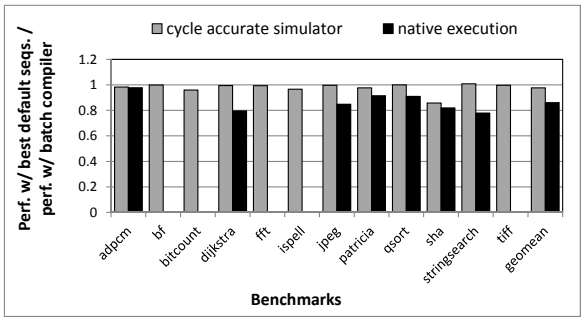


Figure 2. Whole program performance benefit of customizing phase sequences over VPO batch compiler with SimpleScalar cycle-accurate simulator counts and native execution on ARM A9 processor. The following benchmarks contain floating-point instructions and do not execute on the ARM A9 with VPO generated codes: *blowfish*, *bitcount*, *fft*, *ispell*, and *tiff*.

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. However, 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. We again note that SimpleScalar models a very different (StrongARM) micro-architectural as compared the ARM A9 processor, which may explain the difference in performance improvement numbers. However, for our set of benchmarks, we find that customizing phase sequences results in performance gains for many functions and programs.

Please note that the purpose of our validation experiments is to show that the best per-function sequence(s) found during the exhaustive search using *dynamic instruction counts* also produces a corresponding benefit in *simulated processor cycle counts* and *wall-time* on a modern ARM processor. Since the goal of this work is to analyze and exploit optimization phase interactions to prune the phase order search space (while maintaining the same best phase ordering performance), each benchmark/function is executed with only a single input in all our experiments in this paper. For these same reasons, validating/exploring the benefit of a single (*best*) phase ordering sequence with multiple benchmark inputs is outside the scope of this work. However, the effect of different input data sets on iteratively tuning compilers and optimization orderings has been studied by earlier works [28, 29]. These works have observed that a compiler phase sequence trained using one or a few input data sets still performs very well on other data sets.

3. OPTIMIZATION PHASE INTERACTIONS

The goal of this work is to explore, understand, and possibly resolve phase interactions that cause the phase ordering problem, without affecting the best (phase ordering) code produced by the compiler. We employed the exhaustive phase order search algorithm to generate the search spaces for a few of our benchmark functions, and designed several scripts to assist our *manual* study of these search spaces to detect and analyze the most common phase interactions. Due to the importance of registers during many optimization phases, we focused on the effect of register availability and assignment on phase interactions, and the impact of such interactions on the size of the phase order search space.

Architectural registers are a key resource whose availability, or the lack thereof, can affect (enable or disable) several compiler optimization phases. Our explorations confirmed earlier observations that the limited number of available registers in current machines and the requirement for particular program values (like arguments) to be held in specific registers hampers compiler optimizations and is a primary cause for the phase ordering problem [3]. As an example, consider the interaction

DOI: 10.1002/spe

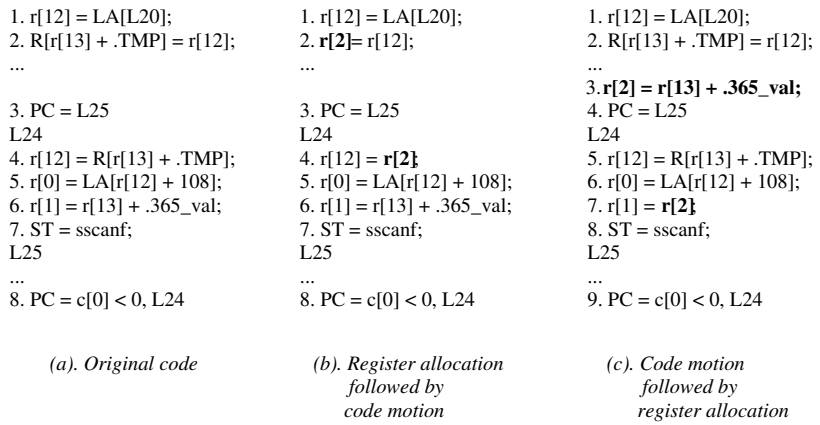


Figure 3. Optimization Phase Interaction between Register Allocation and Code Motion. The RTL notations used in the figure are as follows: *LA* – Load Address; *R[..]* – Memory load/store; *PC*= – Program Counter, denotes branch/jump; *ST*= – Function Call; *r[13]* – Stack Pointer; *c[0]* – Condition Flag

between the phases of *register allocation* and *code motion*, each of which requires registers to do their work. Figure 3(a) shows an example code segment before applying either register allocation or code motion. Consider that, at this point, there is only one register that is available across the entire code segment. If register allocation is applied before code motion, then it uses the available register to hold a value that would otherwise have to be retrieved from memory on every iteration of the loop, generating code as shown in Figure 3(b). Conversely, if code motion is applied before register allocation, it employs the available register to store a sum that would otherwise have to be recomputed on every iteration of the loop, generating code as shown in Figure 3(c). Thus, different ordering of applying optimization phases produces distinct function instances. This example shows a case of a *true* phase interaction that exists due to the contention between optimization phases for the limited number of available machine registers.

However, we were surprised to observe that many individual phase interactions occur, not due to conflicts caused by the limited number of available registers, but by the particular register *numbers* that are used in surrounding instructions. The limited supply of registers on most conventional architectures force optimization phases to minimize their use, and recycle register numbers as often as possible. Many compilers also use a fixed order in which free registers are assigned, when needed. Different phase orderings can assign different registers to the same program live ranges. Different register assignments sometimes result in false register dependences that disable optimization opportunities for some phase orderings while not for others, and cause optimizations applied in different orders to produce distinct codes. Such false register dependence may result in additional *copy* (register to register move) instructions in certain cases, or may cause optimizations to miss opportunities at code improvement due to unfavorable reuse of certain registers at particular program locations. We term phase interactions that are caused by false register dependences as *false interactions*. Such false interactions are often quite arbitrary and not only impact the search space size, but may also make it more difficult for manual and intelligent heuristic search strategies to *predict* good phase orderings.

Figures 4 and 5 illustrate examples of phase interactions due to false register dependence between *instruction selection* and *common subexpression elimination (CSE)*. In the first example, Figure 4(a) shows the code before applying either of these two phases. Figures 4(b) and 4(c) show code instances that are produced by applying CSE and instruction selection in different orders. Without going into the specific details of what this code does, we note that the code in Figure 4(c) is inferior due to a redundant copy instruction left in the code due to an unfavorable register assignment. Even later and repeated application of optimization phases are often not able to correct the effects of such register assignments. Similarly, in the second example shown in Figure 5, applying CSE before

DOI: 10.1002/spe

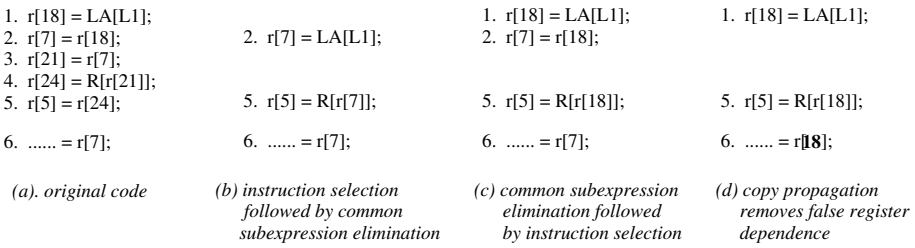


Figure 4. Using *copy propagation* to eliminate false register dependence

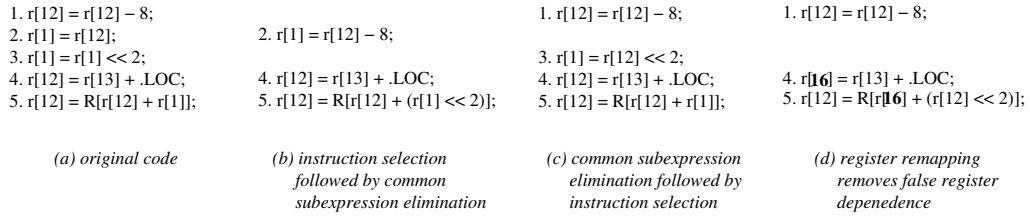


Figure 5. Using *register remapping* to eliminate false register dependence

instruction selection is inferior due to the reuse of register `r[12]`, which prevents instruction selection from combining instructions numbered 3 and 5, and thus leaving an additional instruction in the generated code (Figure 5(c)). Applying instruction selection before CSE avoids this false register dependence issue, producing better code in Figure 5(b). Thus, phase interactions due to false register dependences can produce distinct function instances. Successive optimization phases working on such unique function instances produce even more distinct points in the search space in a cascading effect that often causes an explosion in the size of the phase order search space. At the same time, we believe that due to their arbitrary nature, even existing machine-learning and classifier-based heuristic techniques will find it hard to systematically account for such false interactions. In the next section, we show that the problem of false phase interactions persists even when the number of available registers is virtually unlimited.

4. EFFECT OF REGISTER PRESSURE ON PHASE ORDER SEARCH SPACE AND PERFORMANCE

We introduced the notion of true and false register dependences (and phase interactions) in the last section. Unlike the true dependences, false register dependence will likely not be resolved even if more registers become available later. Such resolution, if it happens, may show itself by shrinking the size of the exhaustive phase order search space since the different phase orderings in such cases will converge to identical code. However, a greater number of registers may also *enable* additional optimization phases and expand the phase order search space. The effect of these additional transformations in such cases may then become visible by some increase in performance of the best phase ordering generated during the search. In this section we present the first study of the effect of different number of available registers on the size of the phase order search space and the performance (dynamic instruction counts) of the best code that is generated.

The ARM architecture provides 16 general-purpose registers, of which three are reserved by VPO (stack pointer, program counter, and link register). We modified the VPO compiler to produce code with several other register configurations ranging from 24 to 512 registers. We successfully gathered the entire phase order search space in all register configurations for 230 of the 234 original benchmark functions (with the exception of: *ispell-askmode*, *ispell-correct*, *tiff-main*, and *fft-main*).

DOI: 10.1002/spe

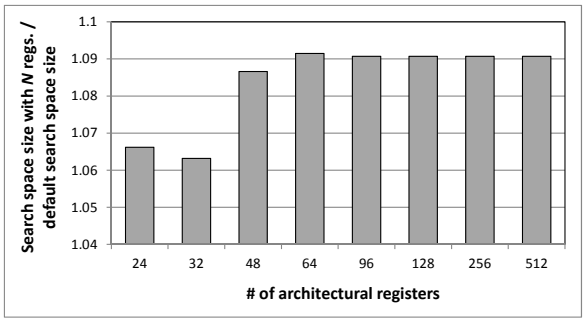


Figure 6. Search space size compared to default for different register configurations

These remaining four functions generated search spaces that were too large to completely enumerate in at least one of the register configurations.

Since our existing ARM-SimpleScalar simulator cannot simulate codes generated (by VPO) with the other *illegal* register configurations, we use a novel strategy to evaluate dynamic instruction counts in such cases. As described in Section 2.3, measuring dynamic performance during our search space exploration only requires program simulations for instances with unseen basic block control-flows. Thus, until the generation of a new control-flow, there is no need for further simulations. Our performance evaluation strategy stores all the control-flow information generated for each function during its exhaustive search space search with 16 registers, and reuses that information to collect dynamic instruction count results during the other illegal VPO register configurations. We find that eight of the 81 executed benchmark functions (*dijkstra-main*, *ispell-strtoichar*, *adpcm-adpcm_coder*, *stringsearch-main*, *ispell-askmode*, *jpeg-parse_switches*, *tiff-main*, and *fft-main*) either generate additional control flows or generate search spaces that are too large to exhaustively enumerate for these other VPO configurations. Thus, our scheme allows us to measure and compare the dynamic instruction counts for 73 executed functions in all register configurations.

Figure 6 illustrates the impact of various register configurations on the size of the phase order search space, averaged (geometric mean) over all 230 benchmark functions, as compared to the default search space size with 16 registers. Thus, we can see that the search space, on average, increases mildly with increasing number of available registers, and reaches a steady state when the additional registers are no longer able to create any further optimization opportunities for any benchmark functions. Figure 7 shows the number of functions that notice a difference in the size of the search space with changing number of available registers. We observe that the search spaces for about 30% of the functions are affected by varying the number of registers. Furthermore, search space sizes are more likely to increase rather than decrease when the number of architectural registers is increased. Performance for most of the 73 executed functions either improves or remains the same, resulting in an average improvement of 1.85% to 1.89% in all register configurations over the default.

The overall increase in the search space size indicates that the expansion caused by additional optimization opportunities generally exceeds the decrease (if any) caused by reduced phase interactions. In fact, we have verified that the current implementation of phases in VPO assumes limited registers and naturally reuses them whenever possible, regardless of prevailing register pressure. Therefore, false register dependences may be an important contributor to the phase order search space. More informed optimization phase implementations may be able to minimize false register dependences and reduce the phase order search space. We explore this possibility further in the next two sections.

DOI: 10.1002/spe

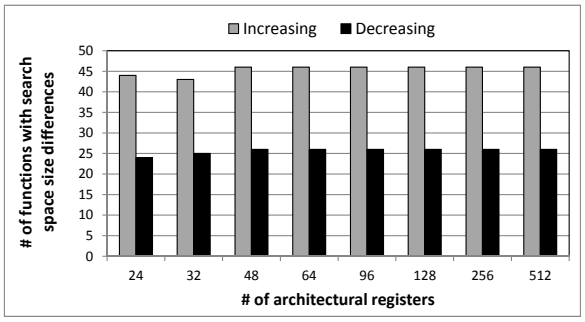


Figure 7. Number of functions that changed search space size for different register configurations

5. MEASURING THE EFFECT OF FALSE REGISTER DEPENDENCE ON THE PHASE ORDER SEARCH SPACE

Our observations presented in the previous section suggest that current implementation of optimization phases typically do not account for the effect of unfavorable register assignments producing false phase interactions. Rather than altering the implementation of all VPO optimization phases, we propose and implement two new stand-alone optimization phases in VPO, *copy propagation* and *register remapping*, that are implicitly applied after every reorderable phase during our iterative search space algorithm to reduce false register dependences between phases. The default VPO compiler applies copy and constant propagation during common-subexpression elimination. *Implicit* phase application implies that the optimization phase is automatically applied, is invisible and is not *explicitly* selected for application by the user. In this section, we evaluate these techniques in a compiler configuration with sufficient (512) number of registers to avoid register pressure issues. In the default VPO configuration extended with 512 available registers, the search spaces for two of the original 234 benchmark functions (*ispell-askmode* and *fft-main*) become too large to exhaustively enumerate. Therefore, in this section, all of our experiments were performed over the remaining 232 benchmark functions. In Section 6 we employ observations from this Section to adapt our techniques to reduce the search space size and improve performance in the default ARM-VPO configuration with 16 registers.

5.1. Copy Propagation to Remove False Register Dependences

Based on our manual analysis of false phase interactions in VPO, we implemented a stand-alone version of *copy propagation* ([30], pp. 356 – 362) to be *implicitly* applied after each of the other phases to potentially minimize the effects of unfavorable register assignments. Copy propagation is often used in compilers as a *clean-up* phase to remove copy instructions by replacing the occurrences of targets of direct assignments with their values. Copy propagation is also often performed during register allocation by *coalescing* the pairs of nodes in the copy instructions in the *interference graph* [31, 32]. Although copy propagation is a well-known optimization, no previous study has explored the effect of (implicit) application of this transformation on phase order search space size.

Figure 4(d) shows the result of applying copy propagation (implicitly after every phase) to the code in Figure 4(c). We can see that applying copy propagation transmits and replaces `r[7]` by `r[18]` on line 6 of Figure 4(d) and eliminates the dead copy instruction on line 2. Thus, the resulting code in Figure 4(d) is now *equivalent* to that in Figure 4(b). Hence, copy propagation can remove some false interactions between optimization phases. However, copy propagation can also extend register live ranges and thus tends to increase register pressure, which can affect the operation of successively applied phases. Therefore, we performed our initial experiments to study the *potential* impact of implicitly applying copy propagation to reduce false phase interactions on the size of the phase order search space in a compiler configuration with sufficient (512) number of registers to avoid register pressure issues.

DOI: 10.1002/spe

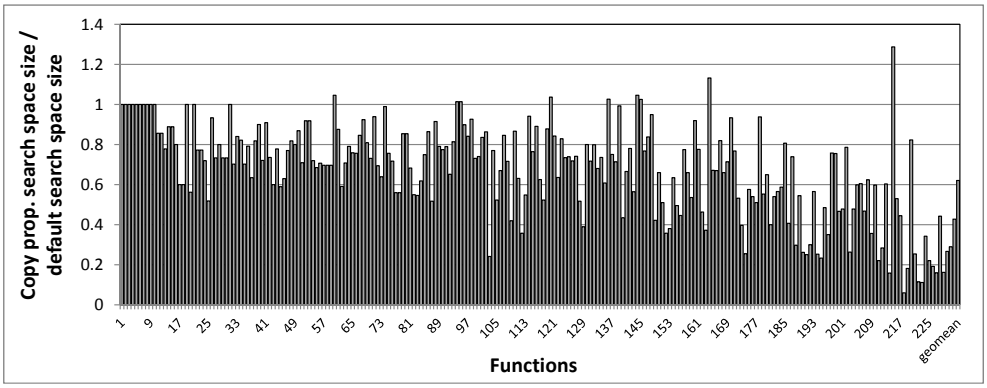


Figure 8. Search space size (# of nodes) with copy propagation implicitly applied during the exhaustive search (512 registers)

Figure 8 shows the change in the phase order search space size compared to default (with 512 registers) if every original VPO phase when successful is followed by the clean-up phase of copy propagation during the exhaustive phase order search space exploration for each function. In this figure, and in each of the subsequent figures comparing the search space size presented in this paper, functions are displayed in the order from smallest to largest default search space size in the graphs. The rightmost bar in each figure presents the average (geometric mean). Thus, on average, the application of copy propagation is able to reduce the size of the search space by 37.9% per function. Interestingly, this technique has a much more significant impact on functions with larger default search space sizes. Indeed, the sum of the search space sizes across all functions with this configuration compared to the sum of search space sizes with the default VPO configuration (with 512 registers) shows a total search space reduction of more than 67.2%.

Our intention for this work is to employ copy propagation as a cleanup phase to reduce the phase order search space while achieving the *same* best phase ordering code as with the default configuration. However, copy propagation can also directly improve performance by eliminating copy instructions. Thus, while our new configuration that *implicitly* applies copy propagation after every phase achieves at least the same performance (dynamic instruction counts) as the default configuration in all cases, it occasionally improves the best generated phase ordering performance (0.41% better than default, on average). [‡]

5.2. *Register Remapping to Remove False Register Dependences*

Experiments described in this section evaluate the potential of renaming register names on the size of the phase order search space and the dynamic instruction counts of the best phase ordering. Register *remapping* or *renaming* reassigns registers to live ranges in a function, and is a transformation that is commonly employed to reduce false register dependences, especially during phases such as *instruction scheduling* [30]. Similar to copy propagation in the earlier section, register remapping is also a popular optimization phase. However, this is the first work that proposes and studies the use of this phase to reduce false phase interactions to prune the size of the phase order search space. Figure 5(d) illustrates the effect of applying register remapping (after every phase) to the code in Figure 5(c) to eliminate the false dependence caused by `r[12]` in instruction #4 by remapping that register to `r[16]`.

[‡]We also report that including our new stand-alone copy propagation *explicitly* as a distinct (16^{th}) reorderable phase during the search space exploration (and not applying it implicitly after every phase) expands the phase order search space for most functions. 13 of the original 232 search spaces become too large to completely enumerate in our search time limit of two weeks. Among the enumerated search spaces, the average search space size increases by 69.7%. Performance improves by 0.6%, on average, over the default configuration.

DOI: 10.1002/spe

Algorithm 1 Register Remapping Algorithm

Input: Function code (in RTLs) organized as a control flow graph (*cfg*)
Output: Function code with registers remapped

```
 1: procedure REMAPREGISTERS(cfg)
 2:     rlg ← calculateRegisterLiveRanges(cfg)
 3:     rlg.markReservedRegs()
 4:     ireg ← initRemapRegNum()                    ▷ 16, to not conflict with hardware registers
 5:     cfg ← remapRTLs(cfg, rlg, ireg)
 6:     return cfg
 7: end procedure

 8: procedure REMAPRTLS(cfg, rlg, ireg)
 9:     for block in cfg.blocks() do
10:         for rtl in block.RTLs() do
11:             for reg in rtl.regs() do
12:                 rlgnode ← rlg.findNode(block, rtl, reg)
13:                 if not rlgnode.isRemapped() then
14:                     if rlgnode.isReserved() then
15:                         rlgnode.remapRegNum ← getRegNum(reg)
16:                     else
17:                         rlgnode.remapRegNum ← ireg
18:                         for sibrlg in rlgnode.siblings() do
19:                             sibrlg.remapRegNum ← ireg
20:                             sibrlg.setRemapped()
21:                         end for
22:                         ireg ← nextRemapRegNum()          ▷ typically, ireg+1
23:                     end if
24:                     rlg.setRemapped()
25:                 end if
26:                 copyreg(reg, makeReg(rlgnode.remapRegNum))
27:             end for
28:         end for
29:     end for
30:     return cfg
31: end procedure
```

In this study we use our extended 512-register ARM configuration to remap as many of the conflicting live ranges as possible to unique register numbers and measure its impact on the size of the phase order search space. Algorithm 1 presents our algorithm for remapping program registers. Our algorithm employs register live range information, which is stored in a graph structure during remapping (*rlg*). Each register occurrence (set or use) in the input code has a corresponding node (*rlgnode*) in the register live range graph and sibling nodes (*rlgnode.siblings*) in the graph correspond to register live ranges. A linear scan over the RTLs assigns free register numbers to live ranges as they are encountered and overwrites each register occurrence in the input code with a new register corresponding to the remapped register number. Reserved registers (stack pointer, program counter, and link register) and registers constrained by machine calling conventions are not remapped. As done with copy propagation in the previous section, we implicitly apply this transformation after each regular optimization during the exhaustive phase order search space exploration for every function.

Figure 9 compares the size of the resulting search space with the default for each function. The results show significant variations with the search spaces for several functions, either substantially increasing or shrinking as compared to the default. Although register remapping can reduce the size

DOI: 10.1002/spe

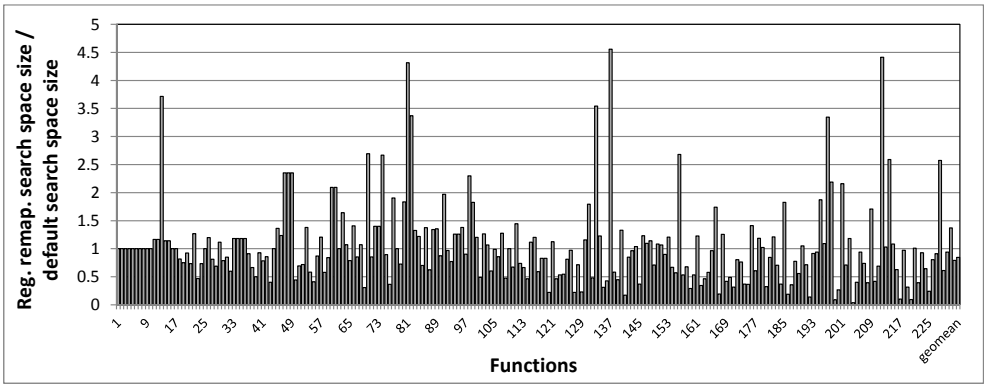


Figure 9. Search space size with register remapping implicitly applied during the exhaustive search (512 registers)

of the phase order search space by removing false register dependences, it is also an *enabling* phase that can provide more opportunities to optimizations following it. These new opportunities increase the size of the search space for several functions. On average, the size of the space reduces by 15.6% per function, while the sum of search space sizes over all functions reduces by only 4.1% compared to the default.

Interestingly, while the objective of our experiment was only to explore the impact of implicit application of register remapping on search space size, we observed that eliminating false register dependences enable additional optimization opportunities for the remaining phases. Consequently, implicit application of register remapping has an inadvertent positive impact on the code quality delivered by individual compiler phase sequences. Figure 10 shows the performance of the best code generated for each function in the exhaustive search configuration with register remapping implicitly applied as compared to the best code generated by the default configuration (with 512 registers). 65 of the original 81 executed functions (with the exception of: *jpeg-pbm_getc*, *bitcount-bit_count*, *bitcount-bit_shifter*, *fft-reverse_bits*, *ispell-trydict*, *jpeg-start_input_ppm*, *dijkstra-main*, *ispell-strtoichar*, *adpcm-adpcm_coder*, *ispell-ichartostr*, *ispell-skipoverword*, *ispell-treelookup*, *ispell-askmode*, *ispell-pfx_list_chk*, *tiff-main*, and *fft-main*) did not generate a new control flow in the configuration with register remapping implicitly applied and thus we measure their performance as described in Section 2.3. Although the best phase ordering code improves in most cases, we surprisingly found that this configuration may also degrade the best performance in a few cases. We explore this phenomenon in more detail in Section 5.4. On average, implicitly applying register remapping during the exhaustive search enabled an improvement in the best performance reached over the default configuration by 0.84%. [§]

5.3. *Combining Register Remapping and Copy Propagation*

Combining our two techniques has a greater impact on pruning the phase order search spaces. Thus, as shown in Figure 11, implicitly applying both register remapping and copy propagation after every phase prunes the phase order search spaces by over 68.3%, on average. This technique also has a more significant effect on functions with larger default search spaces. Thus, this configuration reduces the total number of distinct function instances generated across all functions by a substantial 86.1%. The performance characteristics of the best code generated in this configuration are similar to what we saw in the configuration with only register remapping implicitly applied, yielding a

[§]Again, we report that *explicit* application of register remapping as the 16^{th} reorderable phase in VPO during the exhaustive phase order searches causes an unmanageable increase in the size of the search space for most functions, preventing the searches for 46 of the original 232 functions from finishing. Among the search spaces we were able to gather completely, the average search space size increases by almost 14 times the original size. Performance improves by 4.12%.

DOI: 10.1002/spe

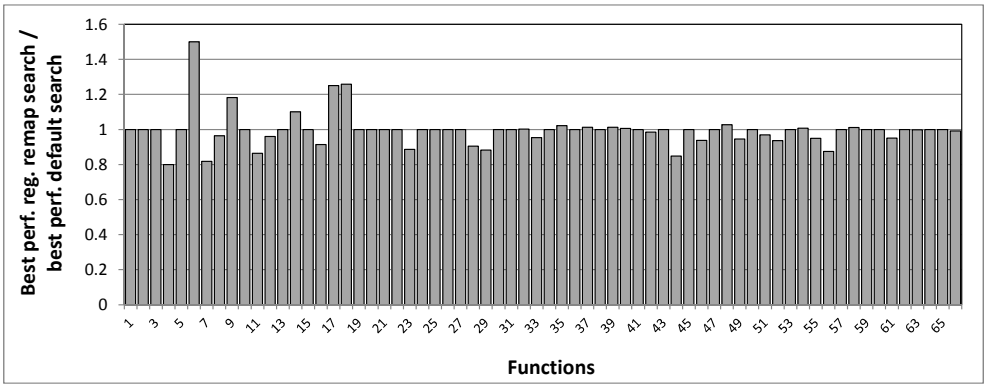


Figure 10. Best code performance with register remapping implicitly applied during the exhaustive search (512 registers)

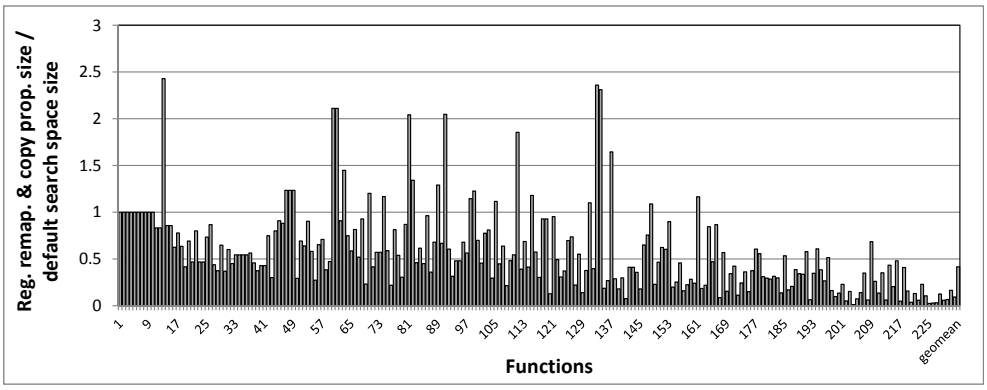


Figure 11. Search space size with register remapping and copy propagation implicitly applied during the exhaustive search (512 registers)

slightly smaller average improvement over the default configuration (0.7%). Thus, reducing false phase interactions caused by differing register assignments can significantly prune the phase order search space size for most functions, given a large number of available registers.

5.4. Machine-dependent Optimization Phase Implementations to Improve Performance

We analyzed several of the functions where our configuration with register remapping implicitly applied is not able to find a phase ordering with performance better than or equal to that found by the default exhaustive phase order search (see Figure 10). Interestingly, in every case we studied, the performance loss was due to issues stemming from the non-orthogonal use of registers. The calling convention for our target ARM machine requires that function arguments be held in specific registers. We found that implicitly applying register remapping may preclude register assignments that interact favorably with this requirement.

An example of the non-orthogonal use of registers affecting the performance of register remapped code is shown in Figure 12. Figures 12(a) and 12(b) show the example codes before and after applying *register allocation* (RA), with no implicit application of register remapping. Figures 12(c) and 12(d) show the corresponding codes in a configuration with register remapping applied after every phase. RA allocates locals (`WORD` in this example) to a register. VPO uses a fixed order to assign available registers, where `r[12]` (if available) is assigned before `r[0]`. For the code in Figure 12(a), the live range of register `r[12]` (instructions #1 and #2) conflicts with the live range

DOI: 10.1002/spe

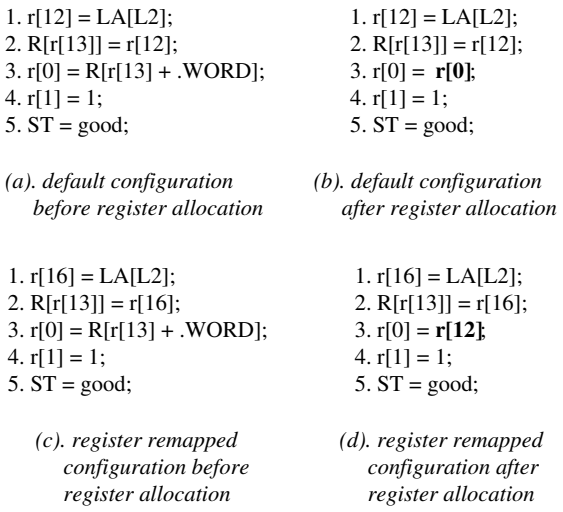


Figure 12. Non-orthogonal use of registers affects the performance of register remapped code

of the local `WORD`, which causes RA to assign `WORD` to register `r[0]`. Note that `WORD` is passed as the first argument to the function `good`, and thus according to the ARM-Linux calling conventions, needs to be present in register `r[0]` before the call to function `good`. RA has conveniently assigned `WORD` to `r[0]` allowing a later application of dead assignment elimination to trivially eliminate instruction #3 in Figure 12(b). On the other hand, in our configuration with register remapping implicitly applied, the live range of register `r[12]` is remapped to register `r[16]`. Since register `r[12]` is now available, RA is able to assign the live range of the local `WORD` to `r[12]`. Unfortunately, this assignment ensures that the copy instruction required to pass `WORD` as the first argument to function `good` in register `r[0]` (instruction #3 in Figure 12(d)) persists in the final code. Thus, non-orthogonal use of registers can cause the compiler with register remapping implicitly applied to produce poorer code than the default, and can result in substantial performance losses if the extraneous instruction(s) is executed repeatedly. This effect is particularly burdensome with our implementation of register remapping which assigns register live ranges to virtual registers (numbered 16–512) and *always* requires an additional copy instruction to prepare arguments for function calls.

As done in VPO, compiler optimization phases are often designed to be machine-independent to enhance compiler portability for easier retargeting to different architectures. The interaction between register remapping and register allocation in VPO is an example of the potential performance tradeoff of this design decision. To minimize this interaction for our current experiments, we updated the RA phase in VPO to encode the knowledge regarding function calling conventions for our target ARM architecture. Thus, for locals that are passed as arguments in function calls, RA attempts to allocate them to the same registers as they would be expected to occupy before function calls.

We again ran the earlier experiment (from Section 5.2) comparing our configuration with implicit application of register remapping to the default exhaustive search setup (targeted to a machine with 512 registers) with the machine-dependent version of RA applied in each configuration. Figure 13 compares the performance of the best code generated for each function. Thus, while some functions still yield some performance degradation, machine-dependent RA mitigates or completely eliminates most of the performance degradation introduced by the implicit application of register remapping. On further analysis, we found that the remaining performance degradations in this experiment were still due to non-orthogonal use of registers. Although possible, resolving these issues would require more extensive modifications to other VPO optimization phase implementations, and so we leave those to future work. The average performance of the best code

DOI: 10.1002/spe

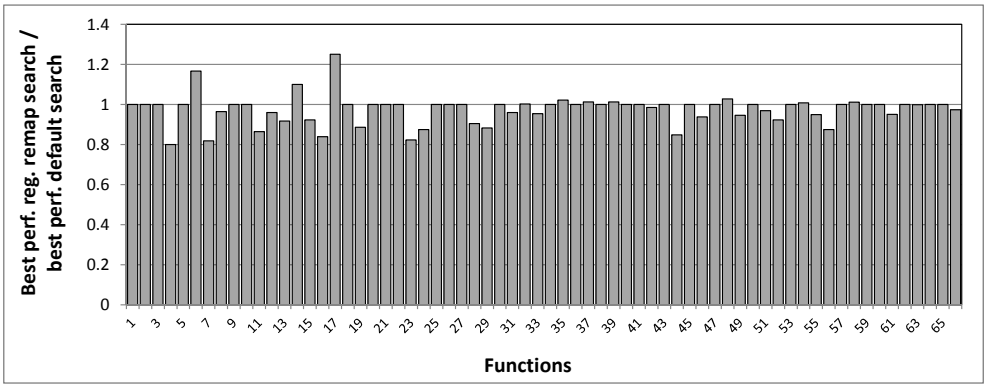


Figure 13. Best code performance with register remapping implicitly applied during the exhaustive search (512 registers) in a compiler with machine-dependent implementation of register allocation

generated for all of the executed functions now improves by 2.63%, on average (as compared to only 0.84% improvement in the original configuration). The change in search space size is similar to what we saw in the configurations with the original RA algorithm. Thus, eliminating false register dependences by the implicit application of register remapping during the exhaustive phase order search can improve the best performing code without prohibitively increasing the search space size.

Note also that, such machine-dependent phase implementation changes coming at the cost of compiler retargetability may be desirable in compilers built for achieving the best possible output code for specialized application domains like embedded systems. We found that applying this small change during the exhaustive search with the default VPO compiler (with no register remapping and targeted to a real machine with 16 registers) improves the best performance found for 4 of our 81 executed functions. While this results in only a small (0.33%) average improvement, this technique does not incur any performance degradations, and the performance of some functions improves significantly (the maximum improvement for one function is 11.4%).

5.5. Observations Applicable to Real Architectures

Our results in this section on a hypothetical architecture with 512 registers demonstrate the potential of copy propagation and register remapping in reducing false register dependence. Copy propagation (by itself and in combination with register remapping) results in a significant pruning of the phase order search space, while finding at least the same best phase ordering performance in all cases. Additionally, these results also show that false register dependence is a major contributor to false phase interactions (in VPO). We also observe that implicit application of register remapping is not as effective by itself in reducing the default phase order search space size. At the same time, we find that eliminating the false register dependences by register remapping shows the potential of enabling additional transformations in other optimization phases, resulting in improving the best phase ordering performance.

6. ELIMINATING FALSE REGISTER DEPENDENCE ON REAL EMBEDDED ARCHITECTURES

In the previous section, we show that applying copy propagation and register remapping can effectively reduce the phase order search space and/or improve performance in a machine with virtually unlimited registers. Unfortunately, both these transformations show a tendency to increase register pressure, which can affect the operation of successively applied phases. In this section we show how we can employ our observations from the last section to adapt the behavior and

DOI: 10.1002/spe

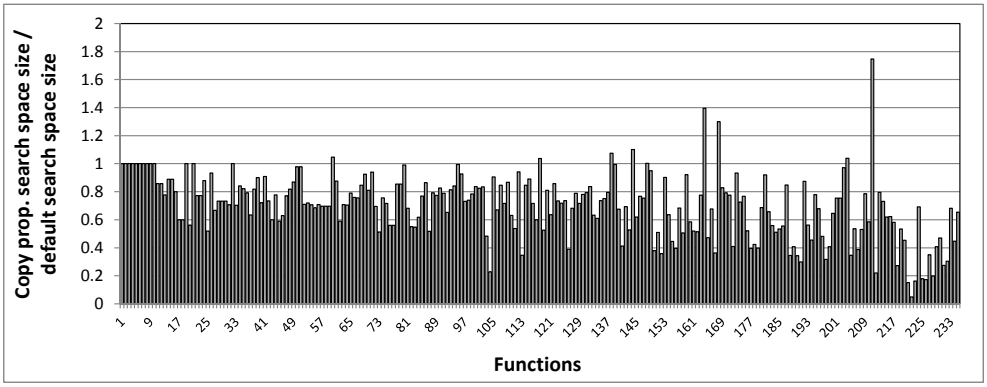


Figure 14. Search space size with copy propagation implicitly applied during the exhaustive search (16 registers)

application of these transformations for use on real machine architectures to reduce search space size and improve generated code quality.

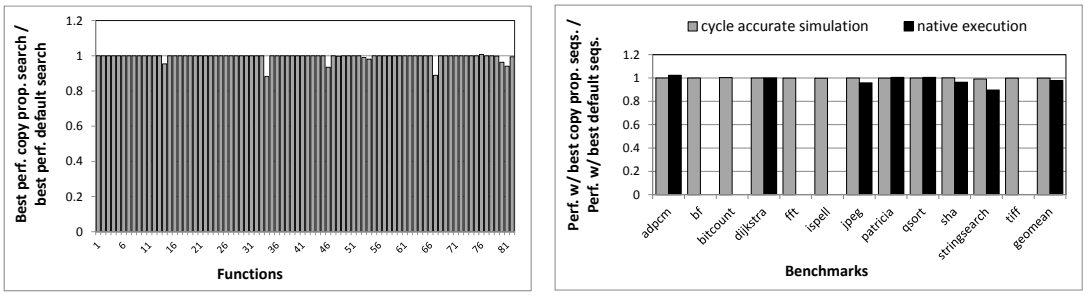
6.1. *Reducing the Search Space with Copy Propagation*

Aggressive application of copy propagation can increase register pressure and introduce register spill instructions. Increased register pressure can further affect other optimizations, that may ultimately result in changing the shape of the original phase order search space and eliminate the best code instance that is detected during the default search. For this reason, we developed a conservative implementation of copy propagation for application on real machines (ARM with 16 registers) that is only successful in cases where the copy instruction becomes redundant and can be removed later. Thus, our transformation only succeeds in instances where we can avoid increasing the register pressure. Our aim here is still to reduce the phase order search space size, while achieving the same best phase ordering performance as detected by the default exhaustive phase order search.

We now apply our version of conservative copy propagation implicitly after each reorderable optimization phase during exhaustive phase order search space exploration (similar to its application in the last section). Figure 14 plots the size of the search space with this configuration as compared to the search space size with the default compiler for each of our benchmark functions. Thus, we can see that, similar to our results in the last section, our technique here reduces the size of the search space, on average, by 34.6% per function, and the total number of distinct function instances by 57.5%. Figure 15 compares the performance of the best code generated *with* and *without* implicit application of copy propagation during the exhaustive search algorithm in terms of per-function dynamic instruction counts (Figure 15(a)) and whole-program simulator cycles and native ARM run-time (Figure 15(b)). Figure 15(a) shows that implicit application of copy propagation only improves the best generated code for a few functions, leaving the average dynamic instruction count almost unchanged (reduced by 0.59%).[¶] The whole-program simulator cycles and native run-time are collected using the same methodology described earlier in Section 2.4 (Figure 2). In particular, we first enumerate the phase orderings that provide the best dynamic instruction counts for each executed function. For each program, VPO generates code by randomly selecting one of its best phase orderings to optimize each executed function and the default batch sequence for all other compiled functions, and repeats this process to generate 100 executables for each benchmark. The best whole-program cycle-count / run-time from these 100 executables is used for our plots. Additionally, the average time of the (last) 60 program runs is used for each native ARM experiment.

[¶]Applying conservative copy propagation explicitly as a distinct reorderable increases the size of the search space by over 77%, on average, and reduces dynamic instruction counts by 0.76%, on average, over the default VPO configuration.

DOI: 10.1002/spe



(a) (b)

Figure 15. Dynamic instruction count and performance benefit of implicitly applying copy propagation during the exhaustive search (16 registers). 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*.

Thus, we can see from Figure 15(b) that the whole-program SimpleScalar simulator cycles show almost no significant change for any benchmark (the largest difference is a 0.8% improvement for *stringsearch*), while run-time (on ARM Cortex A9) improves by 2.2%, on average, over the best respective performance measures obtained by the default exhaustive phase order search space experiments. Thus, prudent application of copy propagation to remove false register dependences can be very effective at reducing the size of the phase order search space even on real ARM machines.

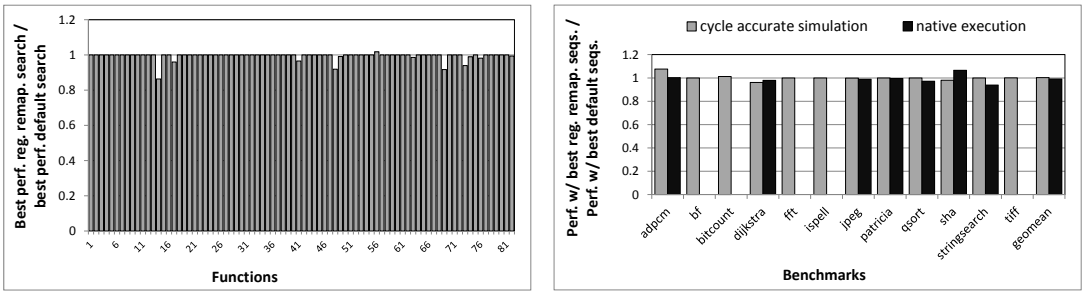
6.2. *Localized Register Remapping to Improve Performance on Real Embedded Architectures*

We have found it more difficult to develop an effective conservative version of register remapping for implicit application during phase order searches. Instead, building on our observations regarding the phase enabling characteristics of register remapping, we develop techniques to explore if removing false register dependences *during* traditional optimization phases can be used to increase optimization opportunities and improve the quality of the generated code.

We select instruction selection to demonstrate our application of *localized* register remapping, but the same technique can also be applied to other phases. As illustrated in Figure 5(c), instruction selection (or some other optimization phase) might miss optimization opportunities due to some false register dependences. We modify instruction selection to only remap those live ranges that are blocking its application due to a false register dependence, if the transformation would be successful otherwise. Thus, when instruction selection fails to combine instructions due to one or more register conflicts, we identify the conflicting live ranges in these instructions, attempt to remap them so that they no longer conflict, and then attempt to combine the instructions again. Such localized application of register remapping can minimize any increase in register pressure as well as potentially provide further optimization opportunities and generate better code.

We found that, on average, this technique is able to marginally improve the performance of the best codes generated during the exhaustive search with only limited increases to search space size. Figure 16 shows the difference in the per-function dynamic instruction counts and whole-program performance (simulator cycles and native ARM run-time) of the best code found by the exhaustive phase order search space algorithm with the modified instruction selection over the default exhaustive search configuration. We again employ a similar experimental setup as used in Section 6.1 to compare the best whole-program processor cycle-counts and run-times. Thus, Figure 16(a) shows that this technique reduces the dynamic instruction counts of the best generated code for 10 of our 81 functions (with one function improving by 13.6%), and yields an average improvement of 0.6%. Figure 16(b) shows that, on average, our instruction remapping technique yields best whole-program simulator cycles that are similar to those obtained by the default exhaustive search, although one benchmark (*adpcm*) degrades by 7.6%. Similarly, average hardware run-time improves slightly (by 0.9%), but one benchmark (*sha*) degrades by 6.5% when compared to

DOI: 10.1002/spe



(a) (b)

Figure 16. Dynamic instruction count and performance benefit for best codes generated with instruction selection remapping transformation compared to default exhaustive search (16 registers). 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 performance of codes compiled with the best sequences from the default exhaustive search. We note that our localized register remapping can also be applied to eliminate false register dependences for the remaining VPO optimization phases and may enable higher performance benefits. This register remapping technique can increase optimization opportunities and results in a small increase in the size of the search space by 7.96% on average. The total number of distinct function instances increases by 19.1%.

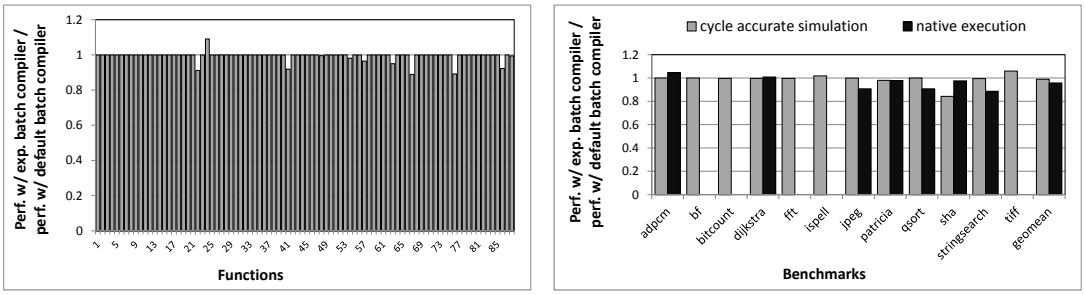
7. APPLYING EXHAUSTIVE SEARCH TECHNIQUES TO CONVENTIONAL COMPILATION

Even with large search space reductions, exhaustive searches may still be too time consuming for many problem domains. In this section, we employ our analysis of the phase order search space to improve conventional compilation. We also present results on finding a single best phase ordering sequence over all functions.

7.1. Copy Propagation and Localized Register Remapping to Improve Performance of Conventional Compilation

Our observations from the previous sections indicate that eliminating false register dependences achieved by conservative copy propagation and register remapping can improve code quality by enabling additional opportunities for other optimizations. In this section we test the usefulness of implicitly applying our conservative version of copy propagation as well as employing our modified instruction selection (with localized register remapping) during conventional (batch) compilation. Figure 17(a) shows the dynamic execution counts of the code generated by this configuration compared to the code generated by the default batch compiler for each of our 87 executed functions. Thus, we can see that, on average, our techniques improve the quality of the code generated by our batch compiler by 0.59%. The modified batch compiler affects the function dynamic instruction counts for ten of the executed functions. Of these, nine yield performance improvements (with a maximum improvement of 11.1%), while the performance of one function degrades (by 9.1%). We analyzed this degrading function and found that, although our techniques do not affect register pressure, they may still affect the operation of successive phases. In this case, our techniques enable the combination of some instructions that contain common subexpressions. This affects the applicability of common subexpression elimination and other successive phases. Thus, in the case of this degrading function, removal of false phase interactions reduces the performance of the generated code by exposing a new *true* phase interaction that was previously not encountered.

DOI: 10.1002/spe



(a) (b)

Figure 17. Performance of batch compilation with copy propagation applied after every phase and localized register remapping applied during instruction selection.

We also validate our results in this section, by measuring the whole-program cycle count performance (with the SimpleScalar cycle-accurate simulator) and native run-time (on the ARM A9) of the code generated by our modified batch compiler. As we can see in Figure 17(b), this technique yields improvements in whole-program processor cycle counts over the default batch compiler (1.1%, on average), although individual benchmark performance may show some degradations (*tiff* shows the largest degradation of 5.9%). Similarly, native run-time execution also improves on average (by 4.3%), with one benchmark (*adpcm*) yielding a 4.7% performance loss. Thus, eliminating false register dependences during conventional compilation shows potential to improve code quality even during conventional compilation.

7.2. *Searching for Optimization Phase Sequences that Achieve the Best Average Performance for a Set of Functions*

Conventional compiler developers are typically tasked with the challenge of finding a single optimization phase sequence that performs well for all input programs. In contrast, evaluating the entire phase order search space makes it possible to find an optimization phase sequence that generates the best code for an *individual function*, but is very expensive even when possible. Recently, researchers are also developing compiler design time supervised learning techniques to find a mapping between program features and effective optimization phase sequences. Such mappings can then be used to customize optimization decisions for unseen programs [12, 33]. However, there has been little work in exploring algorithms to find the best performance that can be achieved by a single phase sequence over a large benchmark set. Unfortunately, such algorithms are likely to be NP-complete. In this section, we develop some *approximation* algorithms to derive a single optimization phase sequence to produce the best phase ordering code, on average, for a set of functions, and discuss the challenges and the progress we make in resolving this important problem.

Any *path* from the root to another node in the search space DAG denotes a phase sequence, which generates the function instance corresponding to that *node*. There can be orders of magnitude more paths than nodes in the DAG. Therefore, although researchers have developed algorithms that can evaluate the performance of all nodes, there has been no attempt to evaluate all paths in the search space DAG. Given a set of functions (with corresponding search space DAGs), our algorithms in this section attempt to find a single phase sequence that generates function instances (nodes) to achieve best average performance over all functions. Our algorithm *simultaneously* searches the entire set of search space DAGs over all functions. This approach is shown in Algorithm 2. Starting at the root nodes of each DAG (and with an empty phase sequence), we compute the average performance of the current set of nodes. Next, we "apply" each of our optimization phases to the current set of nodes (line 19). In this context, applying an optimization phase returns the node reached on application of that phase (i.e., we simply follow the edge from the given node corresponding to that phase). If a node has no edge corresponding to an optimization phase, application of that phase returns the original node. We then perform a recursive call with this new set of nodes (line 26) to continue the

DOI: 10.1002/spe

Algorithm 2 Simultaneous Search Algorithm

Input: Root node of every search space DAG and the set of optimization phases
Output: The best average performance and the phase sequence that generates it
Note: *pos*, *bestperf*, and *bestseq* are global variables.

```
 1: procedure MAIN(roots, opts)
 2:     pos ← 1
 3:     bestperf ← avgperf(roots)
 4:     bestseq ← curseq ← Null
 5:     for i ← 1 to length(roots) do
 6:         markBestSubtreePerfs(roots[i], opts)
 7:     end for
 8:     simSearch(roots, pos, opts)
 9:     return (bestperf, bestseq)
10: end procedure

11: procedure SIMSEARCH(nodes, pos, opts)
12:     if avgperf(nodes) < bestperf then
13:         bestperf ← avgperf(nodes)
14:         copyseqn(bestseq, curseq, pos)
15:     end if
16:     for o ∈ opts do
17:         newchild ← False
18:         for i ← 1 to length(nodes) do
19:             newnodes[i] ← apply(o, nodes[i])
20:             if newnodes[i] ≠ nodes[i] then
21:                 newchild ← True
22:             end if
23:         end for
24:         if newchild and bestSubtreePerfAvg(newnodes) < bestperf then
25:             curseq[pos] ← o
26:             simSearch(newnodes, pos + 1)
27:         end if
28:     end for
29: end procedure
```

search. We do not have any way of keeping track of which sets of nodes have already been traversed together as a set. Instead, we simply compare the new node set to the current node set and, if none of the nodes have changed, do not continue the search from this point. Throughout the search, we maintain global data to record the best average performance and the sequence that generates this performance, and we return this information when the search terminates.

The complexity of the *simSearch* algorithm is equal to the sum of the set of distinct paths (phase sequences) over all of our input DAGs. The number of distinct paths in each search space DAG is 15^N, where '15' is the number of distinct optimization phases in VPO and 'N' is the depth of the DAG (in our experiments, maximum depth of a search space DAG is 37). Techniques to detect and eliminate duplicate function nodes in the DAG enable us to reduce the number of distinct phase sequences scanned by this algorithm for a given set of search space DAGs. Unfortunately, we found even this number is too large (over $1.8 * 10^{16}$) to complete an exhaustive simultaneous search in a practical amount of time. As a result, we devised additional measures to prune away portions of the search space that cannot lead to the best average performance. Before beginning the search, we invoke another algorithm that employs a depth-first search to determine the best performance in the subtree rooted at each node in the DAG (this is shown in Algorithm 2 by calling *markBestSubtreePerfs* on line 6). We use this information during our *simSearch* algorithm to only

DOI: 10.1002/spe

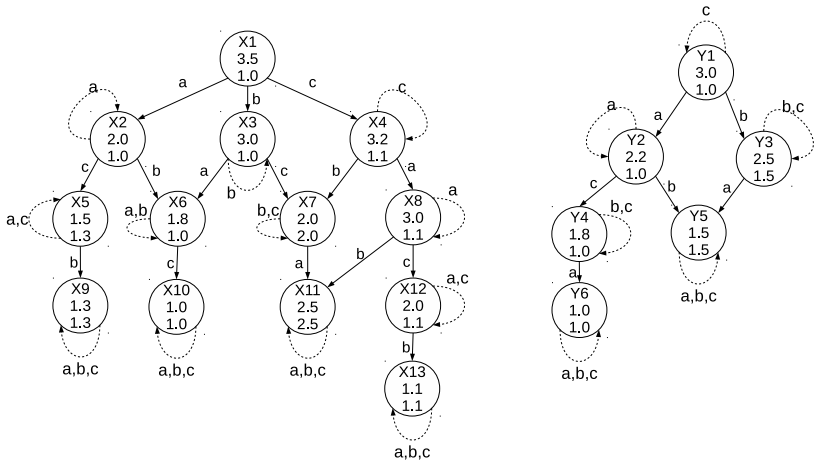


Figure 18. Example DAGs for Simultaneous Search Algorithm

continue searching whenever the average (geometric mean) of the best reachable performances from the subtrees of the new set of nodes is better than the best average performance we have computed up to this point.

We illustrate our algorithm using Figure 18 that shows example DAGs for two hypothetical search spaces with optimization phases `a`, `b`, and `c`. In this example, each node is labeled with (from top to bottom): a unique label to identify the node, the actual performance of the node, ‖ and the best performance reachable from the node's subtree. Edges are labeled with the optimization phase whose application corresponds to the transition from one function instance to another.** The simultaneous search algorithm begins with the root nodes $X1$ and $Y1$ and the initial best average performance is computed as the average performance of these two nodes (3.25). Inside the *simSearch* function, we "apply" our first optimization phase `a` to the current nodes ($X1$ and $Y1$) in each DAG to create a new node set: $X2$ and $Y2$. The best average reachable performance for these new nodes (1.0) is less than the best average performance we have computed up to this point (3.25), and thus, we perform a recursive call with this new node set. Inside the recursive call, we find that the average performance of our current node set (2.1) is less than the current best average performance. Hence, we update our global state information to record our new best average performance and best phase sequence (which, at this point, is simply (`a`)). From here, we again attempt to apply each optimization phase to the current node set. Applying phase `a` does not change the current node set and so we next apply phase `b`, which yields the nodes $X6$ and $Y5$, and the search continues. In this example, the search returns the sequence (`c,a,c,a,b`) that achieves the best average normalized performance of 1.05. Note that we are also able to prune many phase orderings from our search by employing the best subtree performance information stored at each node. For instance, in this example, after exploring the phase orderings prefixed with phase `a`, we do not need to explore any phase orderings prefixed with phase `b` because at this point, ($X3$ and $Y3$) in the search algorithm, we have already discovered a phase sequence (`a,c,a,b`) with better average performance (1.15) than the best possible performance of phase sequences prefixed with phase `b` (1.25).

We implemented and executed the simultaneous search algorithm with the search space DAGs for our 81 executed benchmark functions as input. Unfortunately, despite the pruning techniques described above, we found that there are simply too many phase orderings to explore. Our

‖In order to weigh functions equally, we normalize the performance of each function instance to the optimal function instance in their respective search spaces, with the performance of the optimal phase ordering node(s) as 1.0.

**Edges corresponding to applications of optimization phases which do not lead to different function instances are not explicitly represented in our actual data structures, but these are helpful in understanding our algorithm and thus shown here as dashed edges.

DOI: 10.1002/spe

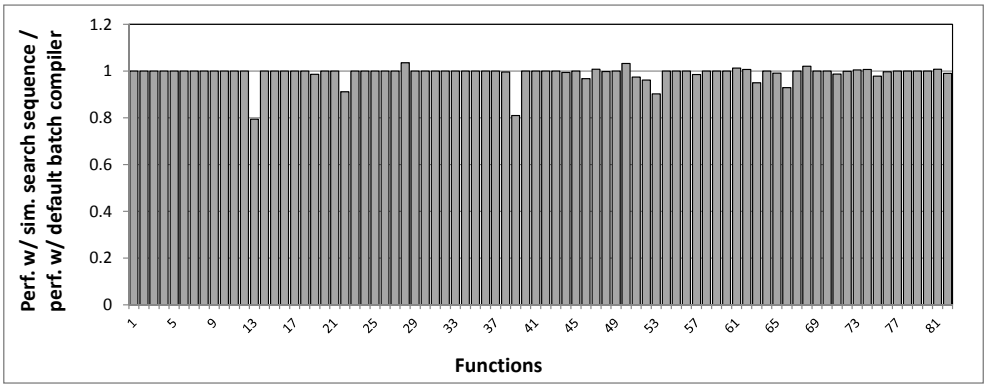


Figure 19. Performance of sequence generated by simultaneous search (with edge marking) compared to default batch compiler. Functions along the x axis are unordered and the average is shown as the rightmost bar.

experiment did not complete even after several weeks. Thus, more advanced pruning techniques will be necessary to realize our goal of finding one phase sequence that achieves the best average performance across all our benchmark functions.

Even though our *exact simSearch* algorithm did not finish, we found that slight modifications to this simultaneous search algorithm can quickly generate *very good* phase sequences. We modified our *simSearch* algorithm to mark the edges in each search space DAG as it traverses them and to only perform recursive calls when application of an optimization phase traverses at least one unmarked edge. This version of the simultaneous search considers each phase interaction at least once, but limits the number of phase orderings considered to the number of edges in our search space DAGs (rather than the enormous number of paths). Running the modified simultaneous search with our 81 executed functions as input completes in only a few seconds and generates the following phase sequence:

```
(b, o, s, o, j, k, h, k, l, s, h, k, s, c, b, s, h, k, c, q, l, g,
     b, s, h, c, s, h, l, i, r, s, j, c, l, s, n, d, r, i, g)
```

Figure 19 compares the performance of the code generated by this sequence to the code generated by the default batch compiler. Thus, this sequence generates slightly better code than the manually tuned and aggressive phase sequence in our default batch compiler that applies phases in a loop until no phase can produce any further changes to the code. One function achieves a dynamic instruction count benefit of more than 20%, while, on average, this sequence yields an average improvement of 1% over the batch compiler sequence. We also find that the performance of the codes produced by this single phase sequence are, on average, only 3.16% worse than the performance of the optimal function instances produced by the individual exhaustive phase order searches.

While this is an encouraging result, this technique may be ignoring single phase orderings that could potentially achieve even better average performance. Furthermore, there is no way of knowing how close the performance of this sequence is to the best average performance that can be achieved by any single sequence for our set of functions. Therefore, we performed further experiments to place bounds on the best achievable single sequence average performance. These experiments restrict our simultaneous search algorithm to only search those portions of the search space that achieve a certain target average performance. If the restricted search completes without finding a sequence that achieves the target performance, there must not exist any such single phase sequence for the input set of functions. We modified our original simultaneous search algorithm to only recurse on sets of nodes when the average of their best subtree performances is less than some target average performance. We then conducted a series of experiments where we gradually increase the target performance (in increments of 0.05%) until our final run did not complete even after several

DOI: 10.1002/spe

weeks. We find that there is no single sequence that can achieve an average performance within 2.2% of the average individual optimal function instances in our benchmark set. Thus, our best *detected* single sequence achieves an average performance that is at least within 0.96% of any other potential best single sequence average performance for our benchmark set.

8. RELATED WORK

In this section we describe previous work in the areas of understanding and addressing the issues of optimization phase ordering and selection. Researchers have developed techniques to enumerate the phase order search space generated by sub-groups of optimization phases in their compilers to understand aspects of the phase order search space. One such work exhaustively enumerated a 10-of-5 subspace (optimization sequences of length 10 from 5 distinct optimizations) for some small programs [14]. Another study evaluated different orderings of performing loop unrolling and tiling with different unroll and tiling factors [21, 6]. Kulkarni et al. developed a novel search strategy to achieve exhaustive evaluation of their compiler's entire phase order search space to find the *best* phase ordering for functions in their embedded systems benchmarks [22, 13]. Such exhaustive searches typically required several hours to a few weeks in most cases [14, 22]. These research efforts have found the search space to be highly non-linear, but with many local minima that are close to the global minimum [34, 14, 20]. Such analysis has also helped researchers devise better heuristic search algorithms. We employ Kulkarni et al.'s algorithm for exhaustive search space evaluation (described in Section 2.2) for our current experiments and show that our techniques to reduce false phase interactions often prunes the phase order search space to enable much faster exhaustive searches. Thus, our work to understand and reduce the phase order search space will most likely further benefit such exhaustive enumeration schemes.

Most recent research efforts to address the phase ordering problem employ iterative compilation to partially evaluate a part of the search space that is most likely to provide good solutions. Many such techniques use machine learning algorithms, such as genetic algorithms, hill-climbing, simulated annealing and predictive modeling to find effective, but potentially suboptimal, optimization phase sequences [10, 34, 11, 14, 12, 20, 15]. Other approaches employ statistical techniques such as fractional factorial design and the Mann-Whitney test to find the set of optimization flags that produce more efficient output code [35, 36, 7]. Researchers have also observed that, when expending similar effort, most heuristic algorithms produce comparable quality code [14, 20]. Our results presented in this paper can enable iterative searches to operate in smaller search spaces, allowing faster and more effective phase sequence solutions.

Our motivation for understanding phase interactions in this work is to reduce the phase order search space to enable faster searches. Past work has also developed algorithms to manage the search time during iterative searches. Static estimation techniques have been employed to avoid expensive program simulations for performance evaluation [25, 13, 9]. Similar techniques are used in our work to speed-up program performance estimations. Agakov et al. characterized programs using static *features* and developed adaptive mechanisms using statistical correlation models to reduce the number of sequences evaluated during the search [12]. Using program features they first characterized an optimization space of 14^5 phase sequences, and then employed statistical correlation models to speed up the search on even the larger optimization spaces. Kulkarni et al. employed several pruning techniques to detect *redundant* phase orderings that are guaranteed to produce code that was already seen earlier during the search to avoid over 84% of program executions during their genetic algorithm search [37]. Fursin et al. exploited program *phases* in larger programs to evaluate multiple optimization configurations within different equal intervals of the same program run to enable faster iterative searches [38]. We employ several of these complementary techniques even as we explore new mechanisms that exploit false phase interactions in this work to realize our goal of understanding and pruning the exhaustive phase order search space.

The traditional approach of iterative compilation that repeatedly executes individual benchmarks with different optimization combinations may be too expensive in certain application domains.

DOI: 10.1002/spe

Consequently, researchers have also developed techniques that use iterative compilation along with machine learning at compiler *design time* to build static models that guide optimization decisions at runtime. For example, Stephenson et al. invented a methodology called *meta optimization* that uses iterative compilation with genetic programming to automatically tune complex compiler heuristics [39]. Supervised learning algorithms have also been used to build classifier-based techniques that automatically learn if, how, and when to apply optimization phases. MILEPOST GCC employs empirical training runs to learn a model that can correlate program *features* with optimization settings [40]. The model can then be used to predict good phase orderings and sequences for unseen programs based on their feature sets. Similarly, Stephenson and Amarasinghe employed program feature-vector based supervised learning to construct nearest neighbor and support vector machine classifier systems to predict loop unroll factors [41]. Supervised learning techniques have also been developed to predict customized method-specific optimization sequences during dynamic just-in-time compilations, based on their corresponding method features [42, 33]. Such compiler design-time techniques provide a more practical mechanism to get the benefits of program customization in a real-world scenario. We expect our insights from this work to make exhaustive phase order searches faster, which can then be used to evaluate the effectiveness of heuristic schemes in finding the best solution.

Research has also been conducted to understand and apply observations regarding optimization phase interactions. Some such studies use static and dynamic techniques to determine the enabling and disabling interactions between optimization phases. Such observations allowed researchers to construct a single *compromise* phase ordering offline [17] and generate a *batch* compiler that can automatically adapt its phase ordering at runtime for each application [22]. The goal of these earlier works was to improve conventional compilation, and their techniques were able to generate phase orderings that often performed better that the original optimization sequence used in their compilers. Instead, our current work attempts to understand the causes behind (the enabling/disabling) phase interactions and exploit that understanding to prune the phase order search space. We also explore the issue of finding a single *best* phase sequence over our set of benchmark functions.

Most related to our current research are studies that analyzed and corrected the dependences between specific pairs of optimization phases. Leverett noted the interdependence between the phases of *constant folding* and *flow analysis*, and *register allocation* and *code generation* in the PQCC (Production-Quality Compiler-Compiler) project [43]. Vegdahl studied the interaction between *code generation* and *compaction* for a horizontal VLIW-like instruction format machine [5], and suggested various approaches to combine the two phases together for improved performance in certain situations. The interaction between *register allocation* and *code scheduling* has been studied by several researchers. Suggested approaches include using postpass scheduling (after *register allocation*) to avoid overusing registers and causing additional spills [16, 30], construction of a register dependence graph (used by instruction scheduling) during register allocation to reduce false scheduling dependences [44, 45], and other methods to combine the two phases into a single pass [4]. Earlier research has also studied the interaction between *register allocation* and *instruction selection* [3], and suggested using a common representation language for all the phases of a compiler, allowing them to be re-invoked repeatedly to take care of several such phase re-ordering issues. Other research has also observed that combining analyses and transformations instead of applying them in some (or more) orders can yield better results [46]. As a proof of concept, the researchers combined the optimization phases of *conditional constant propagation* and *global value numbering* to get an optimization that is more than the sum of its parts. Much of this earlier work was focused on resolving the issue of phase ordering between specific pairs of phases for efficient code generation in a traditional compiler. Our work, in the context of exhaustive phase order search space evaluation, discovers and addresses causes of *false* phase interactions between all compiler phases to reduce the phase order search space.

DOI: 10.1002/spe

9. FUTURE WORK

There are several avenues for future work. For our current research we focus on phase interactions produced by false register dependences and different register assignments. In the future we plan to study other causes of false phase interactions and investigate possible solutions. We believe that eliminating such false interactions will not only reduce the size of the phase order search space, but will also make the remaining interactions more predictable. We would like to explore if this predictability can allow heuristic search algorithms to detect better phase ordering sequences faster. Here, we integrated localized register remapping with instruction selection to produce higher-quality code. In the future, we will attempt to similarly modify other compiler optimizations and study their effect on performance. We also plan to explore if it is possible to implicitly apply other optimization phases outside the phase order search to reduce the search space size without affecting the best achievable performance. Finally, we plan to continue our exploration of new algorithms to find exact empirical bounds on the best average performance that can be achieved by any single optimization phase sequence, and to find the single best phase ordering for a given set of functions.

10. CONCLUSIONS

The problem of optimization phase ordering is a pervasive and long-standing issue for optimizing compilers. The appropriate resolution of this problem is crucial to generate the most effective code for applications in cost and performance-critical domains, such as embedded systems. Although several research directions are attempting to address this problem, very few recent efforts focus on understanding and alleviating optimization phase interactions, which are the root cause of this problem. In this work we highlight the issue of *false phase interactions*, and in particular false register dependences, in our compiler VPO. We devise experiments to show that just lowering the register pressure may not be sufficient to eliminate these false register dependences. Based on our manual observations regarding phase interactions in VPO, we employ two common transformations, *copy propagation* and *register remapping*, to reduce these false dependences. We also show how the reduction in false phase interactions achieved by these transformations can be used to prune the phase order search space size while generating the same best phase ordering solution. In this work we also develop and study algorithms to find a single phase sequence (of any length) to achieve the best performance for all our benchmark functions, and discuss the challenges in resolving this problem. Our approximation algorithms can be used by compiler developers to *automatically* find an effective single phase ordering to be employed in traditional compilers. Our results can not only enable fast exhaustive phase order searches to generate the best phase ordering code, but may also allow conventional compilation to produce better codes.

We also note that different compilers can employ different intermediate representations, and implement different and/or more optimization phases. For example, GCC uses three different intermediate representations (VPO-like RTL, tree-based GENERIC, and static-single-assignment-based GIMPLE), and additional esoteric optimization phases as compared to VPO [47]. As a result, different compilers may show distinct true and false phase interactions. Consequently, techniques that we used in this work to reduce false phase interactions for VPO may be applicable to varying degrees in other compiler frameworks. Regardless, the goal of this work is primarily to highlight the issue of false phase interactions and show the possibility, potential, and benefits of understanding and resolving this problem in compilers. We believe that such understanding is essential to provide guidelines to compiler developers to appropriately address this problem in all current and future compilers.

DOI: 10.1002/spe

A. FUNCTION-LEVEL RESULTS OF OUR EXHAUSTIVE PHASE ORDER SEARCH SPACE EXPLORATION

Function	Code size (# of RTLs)	Dynamic exe. count (batch compiler)	Default search space size (# of nodes)	Dyn. count with best seq. from search / dyn. count with batch compiler
adpcm				
main	139	16465	1676	0.9999
adpcm_decoder	306		32381	
adpcm_coder	384	59612617	24437	0.9828
bf				
main	460	4989969	1896446	0.9906
bitcount				
bfclose	16		9	
alloc_bit_array	24		30	
bit_count	35	5450392	169	0.9862
flipbit	38		61	
getbit	41		55	
ntbl_bitcnt	43	7650000	220	0.7647
bit_shifter	47	18600000	224	1.0000
btbl_bitcnt	47		252	
bfread	58		206	
bfwrite	59		230	
bfopen	62		862	
setbit	64		736	
BW_btbl_bitcount	68	1875000	72	1.0000
bstr_i	70		4497	
AR_btbl_bitcount	83	1650000	87	1.0000
bitcount	133	2100000	44	1.0000
ntbl_bitcount	138	2100000	48	1.0000
main	220	3412776	60282	0.9231
dijkstra				
qcount	12	59980	7	1.0000
print_path	63	3556	188	0.9488
dequeue	76	374375	102	1.0000
enqueue	122	7705076	468	0.9981
main	175	121393	9462	0.9984
dijkstra	353	37852465	88586	0.9600
fft				
IsPowerOfTwo	30	7	378	1.0000
CheckPointer	35	18	93	0.6667
ReverseBits	44	278528	304	1.0000
NumberOfBitsNeeded	59	58	3235	0.7759
Index_to_frequency	86		234	
main	624	1011840	2135723	0.9595
fft_float	679	2712533		
ispell				
move	5		3	
inverse	5		3	
erase	5		3	
backup	5		3	
normal	5		3	
stop	9		3	
putch	11		15	
mymalloc	11		15	
posscmp	18		16	
tryveryhard	23	4669	22	0.8571

DOI: 10.1002/spe

Function	Code size (# of RTLs)	Dynamic exe. count (batch compiler)	Default search space size (# of nodes)	Dyn. count with best seq. from search / dyn. count with batch compiler
ispell				
tbldump	25		92	
pdictcmp	26		27	
combineaffixes	33		106	
done	40		100	
chupcase	41		25	
lowcase	41		327	
upcase	41	20838	327	1.0000
forcelc	43		1261	
flagout	46		30	
wrongcapital	47	7337	36	0.9091
line_size	50		2941	
issubset	51		1178	
treeload	51		296	
myfree	52		440	
ins_cap	55		6367	
onstop	61		10	
TeX_skip_parens	62		330	
TeX_open_paren	62		330	
printichar	65		561	
copyout	66	19234	834	0.9359
ichartosstr	66		70	
TeX_strncmp	67		2190	
strtosichar	70		78	
entryhasaffixes	77		318	
forcevheader	78		1538	
toutword	85		712	
trydict	88	168	783	0.9048
extraletter	91	54878	3543	0.9459
ins_root_cap	94		8147	
expand_pre	97		977	
expand_suf	110		779	
transposedletter	117	53705	5350	0.9572
hash	119	13753679	55126	0.9497
insert	127		4294	
strtoichar	140	74842	10812	1.0000
TeX_skip_args	146		871	
usage	153		16	
setdump	155		6084	
subsetdump	156		2288	
inserttoken	162		20445	
treelookup	166	1638486	67507	1.0000
getline	173		26340	
TeX_math_check	174		613	
show_line	178		12583	
TeX_LR_check	179		3732	
ichartostr	184	13852632	40956	1.0000
wrongletter	192	2235563	22137	0.8699
whatcap	192		21443	
lookharder	194		18042	
lookup	194	12553001	24936	0.9286
tinsert	196		7458	
entdump	205		2896	
dumpmode	207		1160	
skipoverword	211	69090	63772	0.9291
TeX_LR_begin	215		264	
addvheader	217		1631	
compoundgood	220	6670	118136	0.9000
TeX_math_end	221		568	

DOI: 10.1002/spe

Function	Code size (# of RTLs)	Dynamic exe. count (batch compiler)	Default search space size (# of nodes)	Dyn. count with best seq. from search / dyn. count with batch compiler
ispell				
save_cap	226		123891	
acoversb	233		10450	
chk_suf	240	10026766	62709	0.9916
findfiletype	241		15014	
missingletter	252	2344344	8922	0.8858
pr_suf_expansion	257		18660	
missingspace	261	56858	14974	0.9614
combine_two_entries	261		1492	
xgets	273	61520	37960	1.0000
makepossibilities	278	177422	52555	1.0000
stringcharlen	283		65712	
toutent	283		6564	
combinecaps	284		23789	
update_file	294		5460	
chk_aff	298	13638854	149513	0.9994
good	311	17268636	82476	1.0000
show_char	331		57285	
TeX_math_begin	336		50113	
casecmp	339		366006	
givehelp	346		584	
checkfile	414		118966	
shellescape	417		244201	
dofile	431		3432	
pr_pre_expansion	466			
terminit	475		3072	
expandmode	491		23530	
treeinsert	509		1416522	
cap_ok	520			
initckch	534	1922	652348	1.0000
makedent	552		1014659	
flagpr	581			
pfx_list_chk	638	18069232	428526	0.9589
treeinit	660	117	8940	1.0000
TeX_skip_check	679		5112	
treeoutput	767	13		
suf_list_chk	820	169775823		
skiptoword	866	82646	257483	0.9422
askmode	937	7368	144305	0.9647
save_root_cap	1132			
correct	1289		961334	
checkline	1362	94365		
linit	1831	578309		
main	3323	289		
jpeg				
finish_input_tga	5		3	
finish_input_ppm	5	1	3	1.0000
finish_input_gif	5		3	
finish_input_bmp	5		3	
write_stdout	16	4	9	1.0000
read_stdin	16		9	
SkipDataBlocks	18		66	
pbm_getc	40	375	92	1.0000
text_getc	40		92	
jinit_read_ppm	45	11	22	1.0000
read_non_rle_pixel	48		969	
ReInitLZW	51		52	
jinit_read_gif	52		30	
read_byte	52		112	

DOI: 10.1002/spe

Function	Code size (# of RTLs)	Dynamic exe. count (batch compiler)	Default search space size (# of nodes)	Dyn. count with best seq. from search / dyn. count with batch compiler
jpeg				
ReadByte	52		112	
read_byte_targa	52		112	
jinit_read_bmp	52		30	
jinit_read_targa	52		30	
get_raw_row	59	3328	105	1.0000
InitLZWCode	64		60	
DoExtension	66		20	
GetDataBlock	69		88	
get_memory_row	69		129	
get_8bit_gray_row	70		504	
ReadColorMap	79		272	
get_text_gray_row	79		912	
keymatch	92		26939	
get_24bit_row_targa	104		868	
read_rle_pixel	116		12750	
get_pixel_rows	117		1292	
get_text_rgb_row	123		944	
read_colormap_targa	123		1072	
get_scaled_gray_row	125		1396	
read_pbm_integer	129	185	3690	0.9514
get_8bit_row_targa	131		1352	
read_scan_integer	138		44402	
read_text_integer	138		41427	
get_word_gray_row	143		3352	
get_24bit_row	145		2168	
select_file_type	147	28	400	1.0000
set_quant_slots	154		11365	
preload_image_targa	156		760	
get_16bit_row_targa	158		1040	
get_scaled_rgb_row	165		1512	
get_8bit_row	171		2800	
set_sample_factors	204		20511	
read_colormap	216		1036	
get_word_rgb_row	219		2688	
load_interlaced_image	235		4784	
read_quant_tables	238		9171	
get_interlaced_row	247		16900	
preload_image	266		3510	
GetCode	339		63641	
usage	344		34	
main	464	4244	24379	0.9472
LZWReadByte	470		38535	
read_scan_script	478		30000	
start_input_ppm	789	117	7018	0.9829
start_input_tga	961		51316	
start_input_gif	997		38605	
parse_switches	1216	176	162066	0.9375
start_input_bmp	1361		38390	
patricia				
bit	15	525440	37	1.0000
pat_count	68		430	
pat_search	108	1982309	5608	0.9945
insertR	160	49783	2462	0.9953
pat_insert	466	217990	950285	1.0000
main	476	864505	12818	0.9275
pat_remove	549		924825	

DOI: 10.1002/spe

Function	Code size (# of RTLs)	Dynamic exe. count (batch compiler)	Default search space size (# of nodes)	Dyn. count with best seq. from search / dyn. count with batch compiler
qsort				
compare	37	1171434	81	0.7944
main	174	140037	31559	1.0000
sha				
sha_stream	55	489	210	0.9162
sha_print	60	12	45	0.9167
sha_init	87	15	68	1.0000
main	101	30	11262	0.9333
sha_update	118	57042	5504	0.9993
byte_reverse	146	2685023	2755	0.9964
sha_final	155	23	1853	1.0000
sha_transform	541	11397947	459442	0.9128
stringsearch				
bhmi_cleanup	14		7	
init_search	103	1437417	1430	0.9963
strsearch	127	73770	22835	0.9857
main	175	39050	30941	1.0000
bmh_search	179		345619	
bmhi_search	182		179178	
bmh_init	194		5900	
bmha_search	199		372445	
bmha_init	248		35531	
bmhi_init	308		11644	
tiff				
checkcmap	67		2106	
cpTags	68	152	965	1.0000
usage	78		1324	
compresssep	82		345	
compresscontig	94	25031072	240	1.0000
compresspalette	106		17496	
processCompressOptions	289		1734	
cpTag	301	515	522	0.9883
main	1268	39104	1969143	0.9998
totals				
min	5	1	3	0.6667
max	3323	169775823	2135723	1.0000
average	234.27	5515263.82	74176.34	0.9598

Table III. Function-level statistics for our basic exhaustive phase order search space evaluation results

DOI: 10.1002/spe

REFERENCES

1. Jantz MR, Kulkarni PA. Eliminating false phase interactions to reduce optimization phase order search space. *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '10, ACM, 2010; 187–196, doi:http://doi.acm.org/10.1145/1878921.1878950.
2. Davidson JW. A retargetable instruction reorganizer. *SIGPLAN Not.* 1986; **21**(7):234–241, doi:http://doi.acm.org/10.1145/13310.13334.
3. Benitez ME, Davidson JW. A portable global optimizer and linker. *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, 1988; 329–338.
4. Goodman JR, Hsu WC. Code scheduling and register allocation in large basic blocks. *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, 1988; 442–452.
5. Vegdahl SR. Phase coupling and constant generation in an optimizing microcode compiler. *Proceedings of the 15th Annual Workshop on Microprogramming*, IEEE Press, 1982; 125–133.
6. Kisuki T, Knijnenburg P, O'Boyle M, Bodin F, , Wijshoff H. A feasibility study in iterative compilation. *Proceedings of ISHPC'99, volume 1615 of Lecture Notes in Computer Science*, 1999; 121–132.
7. Pan Z, Eigenmann R. Fast and effective orchestration of compiler optimizations for automatic performance tuning. *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, IEEE Computer Society: Washington, DC, USA, 2006; 319–332, doi:http://dx.doi.org/10.1109/CGO.2006.38.
8. Haneda M, Knijnenburg PMW, Wijshoff HAG. Generating new general compiler optimization settings. *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*, 2005; 161–168.
9. Triantafyllis S, Vachharajani M, Vachharajani N, August DI. Compiler optimization-space exploration. *Proceedings of the International Symposium on Code Generation and Optimization*, 2003; 204–215.
10. Cooper KD, Schielke PJ, Subramanian D. Optimizing for reduced code space using genetic algorithms. *Workshop on Languages, Compilers, and Tools for Embedded Systems*, 1999; 1–9.
11. Kulkarni P, Zhao W, Moon H, Cho K, Whalley D, Davidson J, Bailey M, Paek Y, Gallivan K. Finding effective optimization phase sequences. *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, 2003; 12–23.
12. Agakov F, Bonilla E, Cavazos J, Franke B, Fursin G, O'Boyle MFP, Thomson J, Toussaint M, Williams CKI. Using machine learning to focus iterative optimization. *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA, 2006; 295–305.
13. Kulkarni PA, Whalley DB, Tyson GS, Davidson JW. Practical exhaustive optimization phase order exploration and evaluation. *ACM Transactions on Architecture and Code Optimization* 2009; **6**(1):1–36.
14. Almagor L, Cooper KD, Grosul A, Harvey TJ, Reeves SW, Subramanian D, Torczon L, Waterman T. Finding effective compilation sequences. *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2004; 231–239.
15. Hoste K, Eeckhout L. Cole: compiler optimization level exploration. *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, 2008; 165–174.
16. Hennessy JL, Gross T. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems* 1983; **5**(3):422–448.
17. Whitfield D, Soffa ML. An approach to ordering optimizing transformations. *Proceedings of the second ACM SIGPLAN symposium on Principles & Practice of Parallel Programming*, 1990; 137–146.
18. Kulkarni PA, Jantz MR, Whalley DB. Improving both the performance benefits and speed of optimization phase sequence searches. *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, LCTES '10, 2010; 95–104.
19. Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization* December 2001; .
20. Kulkarni PA, Whalley DB, Tyson GS. Evaluating heuristic optimization phase order search algorithms. *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, 2007; 157–169.
21. Bodin F, Kisuki T, Knijnenburg P, O'Boyle M, , Rohou E. Iterative compilation in a non-linear optimisation space. Proc. Workshop on Profile and Feedback Directed Compilation.Organized in conjuction with PACT'98 1998.
22. Kulkarni P, Whalley D, Tyson G, Davidson J. Exhaustive optimization phase order space exploration. *Proceedings of the Fourth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2006; 306–308.
23. Haneda M, Knijnenburg PMW, Wijshoff HAG. Optimizing general purpose compiler optimization. *CF '05: Proceedings of the 2nd conference on Computing frontiers*, ACM Press: New York, NY, USA, 2005; 180–188, doi:http://doi.acm.org/10.1145/1062261.1062293.
24. Burger D, Austin T. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News* 1997; **25**(3):13–25.
25. Cooper KD, Grosul A, Harvey TJ, Reeves S, Subramanian D, Torczon L, Waterman T. Acme: adaptive compilation made efficient. *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, 2005; 69–77.
26. Davidson JW, Whalley DB. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems* November 1991; **15**(9):459–472.
27. Fleming PJ, Wallace JJ. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM* Mar 1986; **29**(3):218–221, doi:10.1145/5666.5673.
28. Chen Y, Huang Y, Eeckhout L, Fursin G, Peng L, Temam O, Wu C. Evaluating iterative optimization across 1000 datasets. *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, ACM: New York, NY, USA, 2010; 448–459, doi:10.1145/1806596.1806647.
29. Haneda M, Knijnenburg PMW, Wijshoff HAG. On the impact of data input sets on statistical compiler tuning. *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, IEEE Computer Society: Washington, DC, USA, 2006; 385–385.

30. Gibbons PB, Muchnick SS. Efficient instruction scheduling for a pipelined architecture. *Proceedings of the SIGPLAN '86 Conference on Programming Language Design and Implementation* June 1986; :11–16.
31. Briggs P, Cooper KD, Torczon L. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems* May 1994; **16**:428–455, doi:http://doi.acm.org/10.1145/177492.177575.
32. George L, Appel AW. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems* May 1996; **18**:300–324, doi:http://doi.acm.org/10.1145/229542.229546.
33. Sanchez R, Amaral J, Szafron D, Pirvu M, Stoodley M. Using machines to learn method-specific compilation strategies. *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, 2011; 257 –266, doi:10.1109/CGO.2011.5764693.
34. Kisuki T, Knijnenburg P, , O'Boyle M. Combined selection of tile sizes and unroll factors using iterative compilation. *Internation Conference on Parallel Architectures and Compilation Techniques*, 2000; 237–246.
35. Box GEP, Hunter WG, Hunter JS. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*. 1 edn., John Wiley & Sons, 1978.
36. Haneda M, Knijnenburg PMW, Wijshoff HAG. Automatic selection of compiler options using non-parametric inferential statistics. *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, IEEE Computer Society: Washington, DC, USA, 2005; 123–132, doi:http://dx.doi.org/10.1109/PACT.2005.9.
37. Kulkarni P, Hines S, Hiser J, Whalley D, Davidson J, Jones D. Fast searches for effective optimization phase sequences. *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, Washington DC, USA, 2004; 171–182.
38. Fursin G, Cohen A, O'Boyle M, Temam O. Quick and practical run-time evaluation of multiple program optimizations. *Transactions on High-Performance Embedded Architectures and Compilers I* 2007; :34–53.
39. Stephenson M, Amarasinghe S, Martin M, O'Reilly UM. Meta optimization: improving compiler heuristics with machine learning. *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, ACM: New York, NY, USA, 2003; 77–90, doi:http://doi.acm.org/10.1145/781131.781141.
40. Fursin G, Miranda C, Temam O, Namolaru M, Yom-Tov E, Zaks A, Mendelson B, Bonilla E, Thomson J, Leather H, *et al.*. Milepost gcc: machine learning based research compiler. GCC Summit 2008. Http://www.milepost.eu/.
41. Stephenson M, Amarasinghe S. Predicting unroll factors using supervised classification. *Proceedings of the international symposium on Code generation and optimization*, CGO '05, IEEE Computer Society: Washington, DC, USA, 2005; 123–134, doi:http://dx.doi.org/10.1109/CGO.2005.29.
42. Cavazos J, O'Boyle MFP. Method-specific dynamic compilation using logistic regression. *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, ACM: New York, NY, USA, 2006; 229–240, doi:http://doi.acm.org/10.1145/1167473.1167492.
43. Leverett BW, Cattell RGG, Hobbs SO, Newcomer JM, Reiner AH, Schatz BR, Wulf WA. An overview of the production-quality compiler-compiler project. *Computer* 1980; **13**(8):38–49.
44. Pinter SS. Register allocation with instruction scheduling. *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, ACM: New York, NY, USA, 1993; 248–257, doi:http://doi.acm.org/10.1145/155090.155114.
45. Ambrosch W, Ertl MA, Beer F, Krall A, Anton M, Felix E, Krall BA. Dependence-conscious global register allocation. *In proceedings of PLSA*, Springer LNCS, 1994; 125–136.
46. Click C, Cooper KD. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems* March 1995; **17**:181–196, doi:http://doi.acm.org/10.1145/201059.201061.
47. GNU. The internals of the gnu compilers. http://gcc.gnu.org/onlinedocs/gccint/ 2011.