Eliminating False Phase Interactions to Reduce Optimization Phase Order Search Space

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

ABSTRACT

Compiler optimization phase ordering is a longstanding problem, and is of particular relevance to the performance-oriented and cost-constrained domain of embedded systems applications. Optimization phases are known to *interact* with each other, enabling and disabling opportunities for successive phases. Therefore, varying the order of applying these phases often generates distinct output codes, with different speed, code-size and power consumption characteristics. Most current approaches to address this issue focus on developing innovative methods to selectively evaluate the vast phase order search space to produce a good (but, potentially suboptimal) representation for each program.

In contrast, the goal of this work is to study and identify common causes of optimization phase interactions across all phases, and then devise techniques to eliminate them, if and when possible. We observe that several phase interactions are caused by *false* register dependence during many optimization phases. We further find that depending on the implementation of optimization phases, even an increased availability of registers may not be able to significantly reduce such false register dependences. We explore the potential of cleanup phases, such as *register remapping* and *copy propagation*, at reducing false dependences. We show that innovative implementation and application of these phases to reduce false register dependences not only reduces the size of the phase order search space substantially, but can also improve the quality of code generated by optimizing compilers.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors- compilers, optimization

General Terms

Performance, Measurements, Algorithms.

Keywords

Phase Ordering, False Register Dependence.

1. INTRODUCTION

Compiler optimization phase ordering and selection have been longstanding and persistent problems for compiler writers and users alike [27, 23, 10]. Each optimization phase applies a sequence of transformations to improve the quality of the generated code for some measure of performance, such as speed, code-size or power consumption. Optimization phases require specific code patterns and/or availability of architectural registers to do their work. Consequently, phases interact with each other by creating or destroying the conditions necessary for the successful application of successive phases. Unfortunately, no single ordering of optimization phases is able to produce the best code for all programs in any investigated compiler [28, 8, 19, 26, 16]. Instead, the ideal phase sequence depends on the characteristics of the code being compiled, the compiler implementation, and the target architecture.

The most common solution to the phase ordering problem employs iterative search algorithms to evaluate the performance of the codes produced by many different phase sequences and select the best one. Although this process may take longer than traditional compilation, longer compilation times are often acceptable when generating code for embedded systems. Many embedded system developers attempt to build systems with just *enough* compute power and memory as is necessary for the particular task. Most embedded systems are also constrained for power. Thus, reducing the speed, code size, and/or power requirements is extremely crucial for embedded applications, as reducing the processor or memory cost can result in huge savings for products with millions of units shipped.

However, the large number of optimization phases typically available in current compilers results in extremely large phase order search spaces that are either infeasible or impractical to exhaustively explore [22]. Therefore, reducing the compilation time of iterative phase order space search is critical to harnessing the most benefits from modern optimizing compilers. Shorter iterative compilation times can be accomplished by two complementary approaches:

1. develop techniques to reduce the phase order search space itself, or
2. devise new search strategies that can perform a more *intelligent* but partial exploration of the search space.

Most recent and existing research effort is *solely* focused on the second approach, and attempts to employ statistical or machine learning methods, along with enhanced correlation techniques to restrict the number of phase sequences that are reached and require evaluation during the iterative search. By contrast, the goal of this work is to analyze and address the most common optimization phase interactions and develop solutions that can substantially reduce the phase order search space. We believe that such reduction will make exhaustive phase order searches more practical. At the same time, understanding and resolving optimization phase interactions can also enable better predictability and efficiency from intelligent heuristic based searches.

Registers are an important resource during the application of many optimization phases, especially in a compiler backend. It is well recognized that the limited number of registers causes several phase interactions and ordering issues by enabling phases when registers become available and disabling transformations when all registers are in use [4, 13]. Interestingly, our analysis of the phase order search space found that many phase interactions are not caused by register contention, but exist due to the dependences between and during phase transformations that reuse the same register numbers. We term such dependences as *false* phase interactions.

In this work, we develop approaches to explore the extent and impact of false phase interactions due to false register dependences on the phase order search space size and generated code performance. We find that techniques to reduce false register dependences *between* phases has a huge limiting effect of the size of the search space. We further find that reducing false dependences *during* phases can also provide additional optimization opportunities and result in improving the quality of the code produced by such phases. Thus, the work presented in this paper shows promise to not only improve the state of iterative compilation for optimization phase ordering, but also provide *guidelines* for compiler implementations to generate higher-quality code.

Thus, the major contributions of our research are:

1. This is the first research to analyze the optimization phase interactions to reduce the phase order search space and improve code quality.

2. We show that the problem of false register dependence is different from register pressure issues, and significantly impacts the size of the phase order search space.

3. We develop techniques to reduce false register dependence that substantially shrink the search space size and improve the quality of code generated by compiler optimizations.

The rest of the paper is organized as follows. We describe related work in the next section. In Section 3 we explain our observations regarding false register dependence between phases. We present our experimental framework in Section 4. In Section 5 we show that the effects of false register dependence are often independent of register pressure issues. We study the effect of eliminating false register dependences on the size of the search space in Section 6. In Section 7 we develop mechanisms to reduce false register dependence to limit the size of the search space and improve code quality. We list avenues for future research in Section 8, and present our conclusions in Section 9.

2. 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 observed that *exhaustive* evaluation of the phase order search space to find the optimal function/program instance, even when feasible, is generally too time-consuming to be practical. Therefore, most research in addressing phase ordering employs 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 [8, 15, 19, 2, 1, 21, 14]. 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 [5, 12, 24]. Researchers have also observed that when expending similar effort most heuristic algorithms produce comparable quality code [2, 21]. Our results presented in this paper can enable iterative searches to operate in smaller search spaces, allowing faster and more effective phase sequence solutions.

Investigators have 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 [7, 22, 26]. 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 [1]. 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 [17, 20]. However, in contrast to our approach, none of these methods make any attempt to understand phase interactions and alter the actual search space itself.

Research has also been conducted to completely enumerate and explore components of the phase order search space. Most of 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 [15, 2, 21]. Such analysis has helped researchers devise better heuristic search algorithms. Kulkarni et al. developed a novel search strategy to achieve exhaustive evaluation of the entire phase order search space and find the *best* phase ordering for most functions in their embedded systems benchmarks, but the searches required several hours to a few weeks in many cases [22]. We employ their algorithm for exhaustive search space evaluation (described in Section 4.2) in our current experiments and show that our techniques to reduce false phase interactions result in much faster exhaustive searches. Thus, our work to understand and reduce the phase order search space will most likely further benefit all such exhaustive enumeration schemes.

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 allow researchers to construct a single *compromise* phase ordering offline [28] or generate a *batch* compiler that can automatically adapt its phase ordering at runtime for each application [18]. Although such custom phase orderings generally perform better that the default sequence used in their compilers, none of these earlier works made any attempt to understand the causes behind those phase interactions.

Most related to our current research are studies that attempt to analyze and correct 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 [23]. Vegdahl studied the interaction between *code generation* and *compaction* for a horizontal VLIW-like instruction format machine [27], and suggested various approaches to combine the two phases together for improved performance in certain situations. The interaction between *register allocation* and

```
1. r[12] = r[12] − 8;
2. r[1] = r[12];
3. r[1] = r[1]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];
```
(a). original code

```
2. r[1] = r[12] − 8;

4. r[12] = r[13] + .LOC;
5. r[12] = Load [r[12] + (r[1]{2)];
```
(b) instruction selection followed by common subexpression elimination

```
1. r[12] = r[12] − 8;

3. r[1] = r[12]{2;
4. r[12] = r[13] + .LOC;
5. r[12] = Load[r[12] + r[1]];
```
(c) common subexpression elimination followed by instruction selection

```
1. r[12] = r[12] − 8;

4. r[0] = r[13] + .LOC;
5. r[12] = Load[r[0] + (r[12]{2)];
```
(d) register remapping removes false register dependence

Figure 1: Using *register remapping* to eliminate false register dependence

```
1. r[18] = Load [L1];
2. r[7] = r[18];
3. r[21] = r[7];
4. r[24] = Load[r[21]];
5. r[5] = r[24];

6. ...... = r[7];
```
(a). original code

```
2. r[7] = Load[L1];

5. r[5] = Load[r[7]];

6. ...... = r[7];
```
(b) instruction selection followed by common subexpression elimination

```
1. r[18] = Load [L1];
2. r[7] = r[18];

5. r[5] = Load [r[18]];

6. ...... = r[7];
```
(c) common subexpression elimination followed by instruction selection

```
1. r[18] = Load [L1];

5. r[5] = Load [r[18]];

6. ...... = r[18];
```
(d) copy propagation removes false register dependence

Figure 2: Using *copy propagation* to eliminate false register dependence

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 [13, 9], construction of a register dependence graph (used by instruction scheduling) during register allocation to reduce false scheduling dependences [25, 3], and other methods to combine the two phases into a single pass [10]. Earlier research has also studied the interaction between *register allocation* and *instruction selection* [4], 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. Unlike our current research, most of these earlier works only studied pair-wise and *true* phase interactions between optimization phases and did not study the effect of removing these interactions on the size of the phase order search space. Rather than focus on specific phases, our research attempts to discover and address causes of *false* phase interactions between all compiler phases to reduce the phase ordering search space.

3. FALSE PHASE INTERACTIONS

Architectural registers are a key resource whose availability, or the lack thereof, can affect (enable or disable) several compiler optimization phases. It is well-known 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 [4]. Our goal for this work is to study 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.

Towards this goal, we employed existing strategies [22] to generate the exhaustive phase order search spaces for a few of our benchmark functions. We also designed several scripts to assist our manual study of these search spaces to detect and analyze the most common phase interactions. Surprisingly, we observed that many individual phase interactions occur, not due to conflicts caused by 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. These 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 call 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 also make it more difficult for manual and intelligent heuristic search strategies to *predict* good phase orderings.

Figures 1 and 2 illustrate examples of phase interactions due to false register dependence between *instruction selection* and *common subexpression elimination (CSE)*. [1] In the first example, Figure 1(a) shows the code before applying either of these two phases. Figures 1(b) and 1(c) show code instances that are produced by applying CSE and instruction selection in different orders. Without

[1]The description of these phases can be found in Table 1.

Optimization Phase	Description
branch chaining	Replaces a branch or jump target with the target of the last jump in the jump chain.
common subexpression elimination	Performs global analysis to remove fully redundant calculations. Also includes global constant and copy propagation.
remove unreach. code	Removes basic blocks that cannot be reached from the function entry block.
loop unrolling	Potentially reduce the number of comparisons/branches at runtime and assist scheduling at the cost of code size increase.
dead assign. elim.	Uses global analysis to remove assignments when the assigned value is never used.
block reordering	Removes a jump by reordering blocks when the target of the jump has only a single predecessor.
minimize loop jumps	Removes a jump associated with a loop by duplicating a portion of the loop.
register allocation	Uses graph coloring to replace references to a variable within a live range with a register.
loop transformations	Performs loop-invariant code motion, recurrence elimination, loop strength reduction, and induction variable elimination on each loop ordered by loop nesting level.
code abstraction	Performs cross-jumping and code-hoisting to move identical instructions from basic blocks to their common predecessor or successor.
eval. order determ.	Reorders instructions within a single basic block in an attempt to use fewer registers.
strength reduction	Replaces an expensive instruction with one or more cheaper ones. For this version of the compiler, this means changing a multiply by a constant into a series of shift, adds, and subtracts.
reverse branches	Removes an unconditional jump by reversing a cond. branch when it branches over the jump.
instruction selection	Combines pairs or triples of instructions together where the instructions are linked by set/use dependencies. Also performs constant folding and checks if the resulting effect is a legal instruction before committing to the transformation.
remove useless jumps	Removes jumps and branches whose target is the following positional block.

Table 1: VPO Optimization Phases

going into the specific details of what this code does [2] we wanted to note that the code in Figure 1(c) is inferior due to the reuse of register `r[12]`, which prevents instruction selection (applied after CSE) from combining instructions numbered 3 and 5, and thus leaving an additional instruction in the generated code. Applying instruction selection before CSE avoids this false register dependence issue, producing better code in Figure 1(b). Similarly, in the second example shown in Figure 2, applying CSE before instruction selection leaves a redundant copy instruction in the code (Figure 2(c)) 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. 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. Before describing our proposed solution and experimental results, we present our experimental framework in the next section.

4. EXPERIMENTAL SETUP

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

4.1 Compiler Framework

The research in this paper uses the Very Portable Optimizer (VPO) [4], which is a compiler back-end that performs all its optimizations on a single low-level intermediate representation called RTLs (Register Transfer Lists). VPO applies several low-level optimization phases that involve registers, providing us with an ideal framework to investigate register dependence effects during phase orderings.

The 15 *reorderable* optimization phases currently implemented in VPO are listed in Table 1. 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 and with a loop unroll factor of 2 for our current experiments. 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 implicitly performed by VPO before the first code-improving phase in a sequence that requires it. After applying the last code-improving phase in a sequence, VPO performs another compulsory phase that inserts instructions at the entry and exit of the function to manage the activation record on the run-time stack. Finally, the compiler also performs *instruction scheduling* before generating the final assembly code.

[2]The '{' operator in the instructions in Figure 1 performs a left shift.

Category	Program	Files/	Funcs.	Description
auto	bitcount	10	18	test proc. bit manipulation abilities
network	dijkstra	1	6	Dijkstra's shortest path algorithm
telecomm	adpcm	2	3	compress 16-bit PCM samples
consumer	jpeg	7	62	image compression and decomp.
security	sha	2	8	secure hash algorithm
office	search	4	10	searches for given words in phrases

Table 2: MiBench Benchmarks Used

For our experiments in this paper, VPO has been targeted to generate code for the StrongARM SA-100 processor using Linux as its operating system. The ARM is a simple 32-bit RISC instruction set. The relative simplicity of the ARM ISA combined with the low-power consumption of ARM-based processors have made this ISA dominant in the embedded systems domain. We use the SimpleScalar set of functional simulators [6] for the ARM to get dynamic performance measures.

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 [11]. We selected one benchmark from each of the six categories of applications present in MiBench. Table 2 contains descriptions of these programs. VPO compiles and optimizes individual functions at a time. The 6 selected benchmarks contain a total of 107 functions, out of which 37 are executed (at least once) with the standard input data provided with each benchmark.

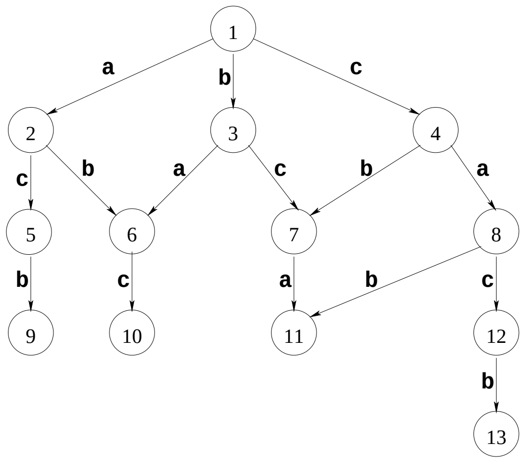

Figure 3: DAG for Hypothetical Function with Optimization Phases `a`**,** `b`**, and** `c`

4.2 Algorithm for Exhaustive Search Space Enumeration

Our goal in this research is to understand the effect of false register dependences on the size of the phase order search space. To generate these per-function exhaustive phase order search spaces we implemented the framework presented by Kulkarni et al. [22]. Our exhaustive phase order searches use all of VPO's 15 reorderable optimization phases. In this section we describe this algorithm to generate exhaustive phase order search spaces.

A naive 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 naive approach is clearly intractable because 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 because it is observed that different phase sequences display a lot of redundancy and frequently result in generating the same code as some earlier sequence. Thus, this interpretation of the phase ordering problem allows the phase order search space to be viewed as a directed acyclic graph (DAG) of *distinct* function instances.

Each DAG is function or program specific, and may be represented as in Figure 3 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 3 that no phase can be successful multiple times consecutively without any intervening phase(s) in between. This algorithm uses various redundancy detection schemes to find phase orderings that generate the same *function instance* as the one produced by some earlier phase sequence during the search. Such detection enables this algorithm to prune away significant portions of the phase order search space, and allows exhaustive search space enumeration for all 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* function instance. Furthermore, any phase sequence from the phase order search space can be mapped to a node in the DAG of Figure 3. This space of all possible *distinct function instances* for each function/program 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. 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.

4.3 Dynamic Performance Measurements

Invoking the *cycle-accurate* simulator for evaluating the performance of every distinct phase sequence produced by the search algorithm is prohibitively expensive. Therefore, we have adopted another technique that can provide quick *dynamic instruction counts* for all function instances with only a few program simulations per phase order search [7, 22]. 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 and simulated to determine 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. Interestingly, researchers have also shown that dynamic instruction counts bear a strong correlation with simulator cycles for simple embedded processors [22]. Note also, that the primary goal of this work is to uncover further redundancy in the phase order search space and reduce the time for phase order searches, while still producing the original best phase ordering code. Thus, although we do not use performance numbers obtained from a cycle-accurate simulator, our dynamic instruction counts are sufficient to validate such performance comparisons.

4.4 Parallel Searches

While the techniques described earlier have made exhaustive searches much more efficient, enumerating the search space for several of our benchmark functions still requires several days (or longer) of computation. The results in this paper required several hundred exhaustive phase order searches across all benchmark functions and experimental configurations. The experiments described in this paper were only made possible due to our access to a high performance computing cluster to run our compute-intensive and independent experiments. The Bioinformatics Cluster at the Information and Telecommunication Technology Center (ITTC) at the University of Kansas contains 176 nodes (with 4GB to 16GB of main memory on each node) and 768 total processors (with frequencies ranging from 2.8GHz to 3.2GHz). With this computing power, we were able to parallelize the phase order searches by running many different searches on individual nodes of the cluster. As an indication of the compute-intensiveness of this study, we note that the usage logs from the cluster show that experiments related to this project would have required several years of single CPU time. Despite the available computing power, we were unable to completely enumerate the exhaustive phase order space for all of our benchmark functions with some of the experimental configurations of VPO due to time / space constraints. Individual exhaustive searches that took longer than two weeks or generated raw data files larger than the maximum allowed on our 32 bit system (2.1GB) were simply stopped and discarded.

5. EFFECT OF REGISTER PRESSURE ON PHASE ORDER SPACE AND PERFORMANCE

We have seen that several optimization phase interactions are caused by different register assignments produced by different phase orderings. Such effects can cause a false register dependence to disable optimization opportunities for some phase orderings while not for others. False register dependence is probably an artifact of the limited number of registers available on most machines. Such register scarcity forces optimization phases to be implemented in a fashion that reassigns the same registers often and as soon as they become available. If phases are implemented correctly, then a decrease in register pressure should also reduce false register dependences. If so, then we should expect the phase order search space to shrink with increasing register availability. However, a greater number of registers may also *enable* additional phase transformations, expanding the phase order search space and made visible by some increase in performance of the best code generated during the search as compared to the default. 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 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 available registers. We perform experiments to measure the phase order search space size for our 107 benchmark functions in all register configurations.

Since the code generated by VPO with the other *illegal* register configurations cannot be simulated, we used a novel strategy to evaluate code performance in such cases. As described earlier, 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 performance results during the other illegal VPO register configurations. We found that no additional control flows were generated for 32 of the 37 executed benchmark functions for these other VPO configurations. Thus, our scheme allows us to measure and compare the dynamic performance for 32 executed functions in all register configurations.

Figure 5 illustrates the impact of various register configurations on the size of the phase order search space, averaged over all 107 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 5 shows the number of functions that notice a difference in the size of the search space with changing number of available registers. Thus, we observe that there is an almost equal number of functions that see a search space increase as the number that show a decrease, for all register configurations. Performance for all, except one, of the 32 executed functions either improves or remains the same, resulting in an average improvement of 1.1% 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, limited number of registers is not the sole cause for false register dependences. Consequently, 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.

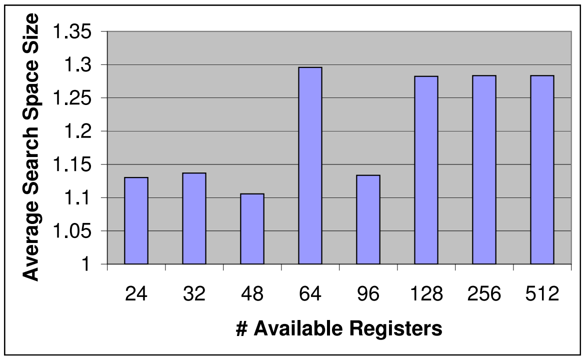

Figure 4: Search space size compared to default for different register configurations

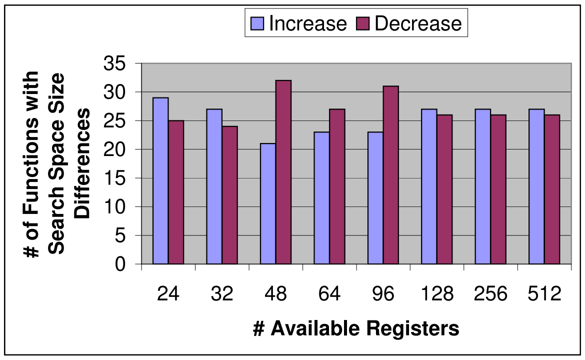

Figure 5: Number of functions with different search space size compared to default for different register configurations

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

Our results in the previous section suggests 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 transformations in VPO, *register remapping* and *copy propagation*, that are implicitly applied after every reorderable phase during our iterative search space algorithm to reduce false register dependences between phases. In this section, we show that removing such false phase interactions can indeed result in a dramatic reduction in the size of the phase order search space in a compiler configuration with sufficient (512) number of registers to avoid register pressure issues. In the next section we adapt and employ our techniques to reduce search space size and improve performance in the default ARM-VPO configuration with 16 registers.

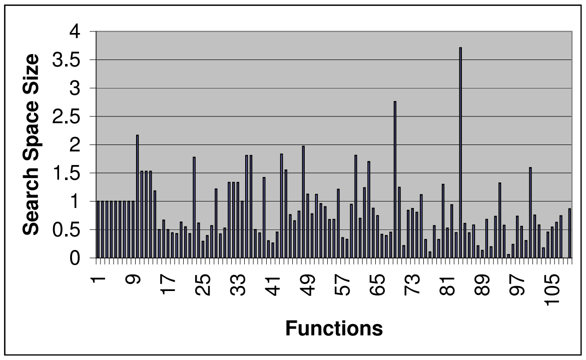

Figure 6: Search space size with register remapping compared to default (512 registers)

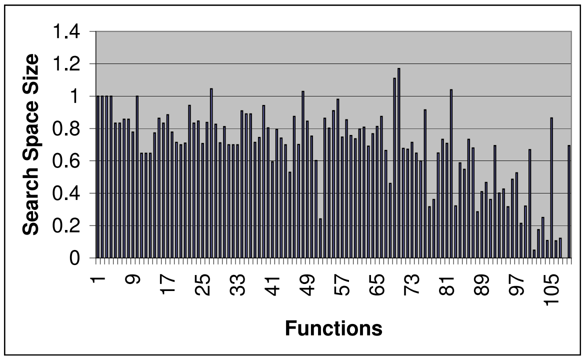

Figure 7: Search space size with copy propagation compared to default (512 registers)

6.1 Register Remapping to Remove False Register Dependences

Register *remapping* or *renaming* reassigns registers to live ranges in a function, and is a transformation commonly employed before *instruction scheduling* to reduce false register dependences and increase instruction level parallelism [9]. Figure 1(d) illustrates the effect of applying register remapping (after every phase) to the code in Figure 1(c) to remove the false interaction between instruction selection and CSE in Figure 1. In this study we use 512 available registers to remap as many of the conflicting live ranges as possible to unique register numbers. We apply this transformation after each regular optimization during the exhaustive phase order search space exploration for each function.

Figure 6 shows the effect of implicitly applying register remapping after every reorderable phase during the exhaustive search space exploration on the size of the search space for all 107 benchmark functions. In this figure, and in each of the subsequent figures presented in this paper, functions are sorted from smallest to largest default search space size and are displayed in this order in the graphs. The rightmost bar in each figure presents the average. Thus, on average, our compiler configuration with implicit register remapping is able to reduce the search space size by over 13% per function. Interestingly, this technique has a more significant impact on functions with larger default search spaces. Thus, summing up the search space sizes over all 107 functions, we find that the number of total distinct function instances reduces by 37.4% compared to the default.

Although register remapping cannot directly impact dynamic performance, it is 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. Indeed, *explicitly* including register remapping as the 16th reorderable phase in VPO during the exhaustive phase order searches causes an unmanageable increase in the size of the search space for all functions, preventing the searches for many functions from finishing even after several weeks. Therefore, it seems even more noteworthy that our configuration that *implicitly* applies this transformation after every phase can reduce the search space size so substantially even as it enables more phases. We also found that the additional optimization opportunities due to implicit application of register remapping only marginally affect the best code performance found during the exhaustive phase order search for a few functions, and results in an average performance improvement of 0.4%.

6.2 Copy Propagation to Remove False Register Dependences

Next, based on our manual analysis of false phase interactions in VPO, we implemented *copy propagation* as another transformation to potentially further 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. Figure 2(d) shows the result of applying copy propagation (after every phase) to the code in Figure 2(c). Thus, we can see that applying copy propagation transmits and replaces `r[7]` by `r[18]` on line 6 of Figure 2(d) and eliminates the dead copy instruction on line 2. Thus, the resulting code in Figure 2(d) is now equivalent to that in Figure 2(b). We performed experiments to study the impact of implicitly applying copy propagation to reduce false phase interactions on the size of the phase order search space.

Figure 7 shows the change in the phase order search space size compared to default if every original VPO phase when successful is followed by the clean-up phase of copy propagation during exhaustive phase order space search for each function. Thus, on average, the application of copy propagation is able to reduce the size of the search space by over 30.5% per function. Similar to our earlier approach with register remapping, this technique has a much more significant impact on functions with larger default search space sizes. Indeed, the sum the 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 68%. Unlike the enabling effect produced by register remapping, copy propagation can directly improve performance by eliminating copy instructions. We found that *implicitly* applying copy propagation after every phase allows the exhaustive phase order searches to generate best function instances that achieve 0.55% better performance that default, on average. At the same time, we also observed that including copy propagation *explicitly* as a distinct (16th) reorderable phase during the search space exploration (and not applying it implicitly after every phase) has no additional benefit in producing better code instances. Moreover, we observed that such a VPO configuration that explicitly applies copy propagation, instead of reducing the search space, doubles the size of the phase order search space per function, on average.

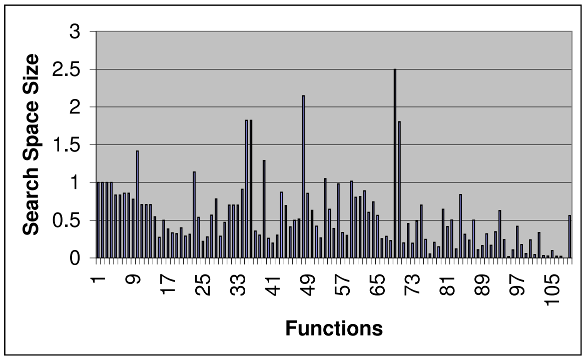

Figure 8: Search space size with register remapping, copy propagation compared to default (512 registers)

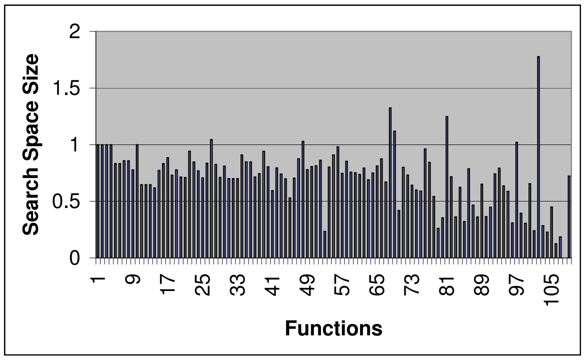

Figure 9: Search space size with copy propagation compared to default (16 registers)

6.3 Combining Register Remapping and Copy Propagation

Interestingly, combining our two techniques is able to further reduce false register dependences and the size of the phase order search spaces. Thus, as shown in Figure 8, implicitly applying both register remapping and copy propagation after every phase reduces the size of the phase order search spaces by over 43.8%, on average, while marginally improving the best average performance by 0.57%. This technique also has a much 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 an impressive 89.7%. Since both our implicit phases reduce false register dependences, our results in this section demonstrate that false phase interactions caused by differing register assignments are responsible for significantly contributing to the size of the phase order search space.

7. ELIMINATING FALSE REGISTER DEPENDENCE ON REAL EMBEDDED ARCHITECTURES

In the previous section, we showed that applying register remapping and copy propagation effectively reduces the phase order search space in a machine with virtually unlimited registers. Unfortunately, both these transformations show a tendency to increase register pressure, which can affect the operation of successive applied phases. In this section we show how we can employ our observations from the last section to adapt the behavior and application of these transformations for use on real embedded hardware to reduce search space size and improve generated code quality.

7.1 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 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 to reduce the phase order search space size, but still achieve that same best performance as detected by the default 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 9 plots the size of the search space 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 27.7% per function, and the total number of distinct function instances by 55.4%. Similarly, implicit application of copy propagation during the exhaustive search algorithm improves the best generated code for a few functions, improving average performance by 0.50%. Thus, prudent application of techniques to remove false register dependences can be very effective at reducing the size of the phase order search space on real machines.

7.2 Improving Performance with Localized Register Remapping

We have found that developing a similar conservative version of register remapping for implicit application during phase order searches is more difficult. Instead, we employ register remapping to show how 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 1(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 these 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.

An instruction selection transformation that is unable to combine instructions due to false register dependence, may still fail after our localized register remapping due to some other issues. Our first implementation of this transformation left such futile remappings in place introducing new (locally remapped) function instances in the

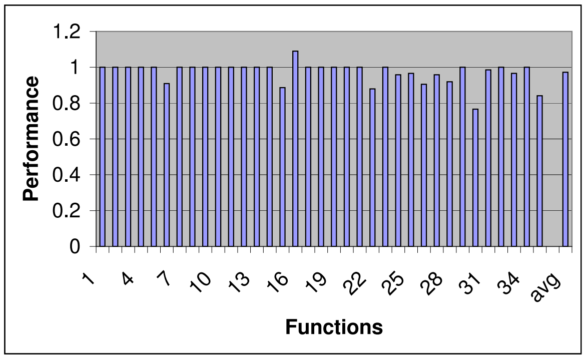

Figure 10: Performance with instruction selection remapping transformation compared to default (16 registers)

search space. This issue causes an explosion in the size of the phase order search space in several functions and results in an average per function search space size increase of 315%. Furthermore, due to the increased search space sizes, we were not able to complete the exhaustive search for 5 of our 107 benchmark functions (2 of which were executed).

Despite these issues, we found that the performance of the code generated with this technique improved considerably. Figure 10 shows the improvement in the performance of the best code found by the exhaustive phase order search space algorithm with the modified instruction selection for each of the 35 executed benchmark functions over the default. We found that the best code performance improved by 2.77%, on average. Further, we tested the usefulness of this approach during the conventional (batch) compilation. 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. We found that our modified instruction instruction enabled the batch compiler to improve the performance of the generated code by a healthy 1.46%, on average.

Please note that in addition to instruction selection, localized register remapping may also benefit several other low-level optimization phases. Therefore, we believe that the concept of addressing false dependences during optimization phases to improve the quality of generated code shows immense promise during iterative as well as conventional compilation.

8. FUTURE WORK

There are several avenues for future work. For our current research we focused 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. In this work we integrated localized register remapping with instruction selection to produce higher-quality code. In the future, we will explore the possibility of a more conservative version of register remapping to limit the bloat in the size of the phase order search space, while still retaining the performance benefits. At the same time, we will attempt to similarly modify other compiler optimizations and study their effect on performance. Finally, we 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.

9. CONCLUSIONS

Effectively addressing the optimization phase ordering problem is important for applications in the embedded systems domain. We found that the problem of huge phase order search spaces is partly a result of the interactions between optimization phases that are caused by false register dependences. We also discover that due to the current implementation of optimization phases, even reducing the register pressure by increasing the number of available registers is not sufficient to eliminate false register dependences. Our new transformations, register remapping and copy propagation, to reduce false register dependences are able to substantially reduce the size of the phase order search spaces, but at the cost of increased register pressure that is not sustainable on real machines. We then showed that conservative implementation of these transformations during and between phases can still achieve impressive reductions in the search space size, while also achieving better code quality.

10. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by NSF grant CNS-0953268.

11. REFERENCES

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

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

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

[4] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 329–338, 1988.

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

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

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

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

[9] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *Proceedings of the SIGPLAN '86 Conference on Programming Language Design and Implementation*, pages 11–16, June 1986.

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

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

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

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

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

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

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

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

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

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

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

[21] P. A. Kulkarni, D. B. Whalley, and G. S. Tyson. Evaluating heuristic optimization phase order search algorithms. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 157–169, 2007.

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

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

[24] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 319–332, 2006.

[25] S. S. Pinter. Register allocation with instruction scheduling. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 248–257, New York, NY, USA, 1993. ACM.

[26] S. Triantafyllis, M. Vachharajani, N. Vachharajani and D. I. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 204–215, 2003.

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

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