Exhaustive Optimization Phase Order Space Exploration

Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson
Florida State University
Computer Science Department
Tallahassee, FL 32306-4530
{kulkarni,whalley,tyson}@cs.fsu.edu

Jack W. Davidson
University of Virginia
Department of Computer Science
Charlottesville, VA 22904-4740
jwd@virginia.edu

Abstract

The phase-ordering *problem is a long standing issue for compiler writers. Most optimizing compilers typically have numerous different code-improving phases, many of which can be applied in any order. These phases interact by enabling or disabling opportunities for other optimization phases to be applied. As a result, varying the order of applying optimization phases to a program can produce different code, with potentially significant performance variation amongst them. Complicating this problem further is the fact that there is no universal optimization phase order that will produce the best code, since the best phase order depends on the function being compiled, the compiler, and the target architecture characteristics. Moreover, finding the optimal optimization sequence for even a single function is hard as the space of attempted optimization phase sequences is huge and the interactions between different optimizations are poorly understood.*

Most previous studies performed to search for the most effective optimization phase sequence assume the optimization phase order search space to be extremely large, and hence consider exhaustive exploration of this space infeasible. In this paper we show that even though the attempted *search space is extremely large, with careful and aggressive pruning it is possible to limit the* actual *search space with no loss of information so that it can be completely evaluated in a matter of minutes or a few hours for most functions. We were able to exhaustively enumerate all the possible function instances that can be produced by different phase orderings performed by our compiler for more than 98% of the functions in our benchmark suite. In this paper we describe the algorithm we used to make exhaustive search of the optimization phase order space possible. We then analyze this space to automatically calculate relationships between different phases. Finally, we show that the results of this analysis can be used to reduce the compilation time for a conventional batch compiler.*

1. Introduction

Most current optimizing compilers contain tens of different optimization phases. Each optimization phase analyzes the program representation and performs one or more transformations that will preserve the semantics of the program, while attempting to improve program characteristics, such as execution time, code size, and energy consumption. Many of these optimization phases use and share resources, such as registers, and most require specific conditions in the code to be applicable. As a result these phases interact with each other by enabling and disabling opportunities for other optimizations.

Due to these interactions between optimization phases, the final code produced differs depending on the order in which optimizations are applied, leading to different program performance. Most of these phase interactions are difficult to predict since they are highly dependent on the program being compiled, the underlying architecture, and also on their specific implementation in the compiler. Most command-line compilers provide the user with some specific flags (such as -O, -O2, -O3 for GCC) that applies optimization phases to all programs in one fixed order. However, it is widely acknowledged that a single order of optimization phases does not produce optimal code for every application [1, 2, 3, 4, 5, 6]. Therefore, researchers have long been trying to understand the properties of the optimization phase order space so that near optimal optimization sequences can be applied when compiling applications.

A naive solution to the phase ordering problem is to exhaustively evaluate all orderings of optimization phases. This solution quickly becomes infeasible in the face of tens of different optimization phases typically present in most current compilers and few restrictions on the ordering of these phases. This problem is made more complex by the fact that many optimizations are successful multiple times, which makes it impossible to fix the sequence length for all functions. The compiler used in this paper uses 15 distinct optimization phases, which over a sequence length of

n would result in 15^n possible attempted sequences. In fact, the largest sequence length of phases changing the program representation that we found in our study was 32 and 15^{32} is a very large number of sequences. Due to the *attempted* search space being so large, we are unaware of any prior work to completely enumerate it.

While keeping the above statistics in mind, it is at the same time important to realize that not all optimization phase sequences that can be attempted will produce unique code. Earlier studies [7] have suggested that the *actual* search space might not be this large, which is partially caused by many optimization phases often not being successfully applied when attempted, i.e. they are not able to apply transformations that will affect the generated code. Likewise, many optimization phases are independent in that the order in which successful phases are applied will have no effect on the final code. We have found that it is possible in many cases to exhaustively enumerate the actual optimization phase order space for a variety of functions. Out of the 111 functions we analyzed within six different benchmarks, we were able to completely enumerate the search space of different phase orderings performed by our compiler for all but two of the functions. These enumerations were performed in a matter of a few minutes in most cases, and a few hours for most of the remainder. Thus, the major contributions of this work are: (1) a realization that the actual optimization phase order space is not as large as earlier believed, is considerably smaller than the attempted space, and hence can often be exhaustively enumerated; (2) the first analysis of this space to capture probabilities of various phase interactions, such as inter-phase enabling/disabling relationships and inter-phase independence; and (3) the first application of phase enabling/disabling probabilities to improve compilation time for a conventional compiler.

The remainder of this paper is organized as follows. In Section 2 we review a summary of other work related to this topic. In Section 3 we give an overview of our experimental framework. We describe our algorithm to enumerate the optimization phase order space in Section 4. In Section 5 we present the results of our analysis of the space. We used the analysis results in Section 6 to improve the compilation time of our conventional batch compiler. In the last two sections we list several directions for future work and our conclusions.

2. Related Work

In prior work researchers have studied optimization phase order and parameter characteristics by enumerating the search space in certain limited contexts. The small area of the transformation space formed by applying loop unrolling (with unroll factors from 1 to 20) and loop tiling (with tile sizes from 1 to 100) was analyzed for a set of three program kernels across three separate platforms [8]. This study found the search space to be highly non-linear, containing many local minima and some discontinuities. As the number of optimizations considered are increased, the attempted search space grows exponentially. Enumerations of search spaces formed by a larger set of distinct optimization phases have also been investigated [9]. One important deduction was that the search space generally contains enough local minima that biased sampling techniques, such as hill climbers and genetic algorithms, should find good solutions. But even in this study the exhaustive search was not allowed to span all the optimizations available in the compiler, and it took months to search even this limited space when evaluating each benchmark.

Several groups have also worked on the problem of attempting to find the best sequence of compiler optimization phases and/or optimization parameters in an attempt to reduce execution time, code size, and/or energy consumption. Specifications of code improving transformations have been automatically analyzed to determine if one type of transformation could enable or disable another [10, 2]. This work was limited by the fact that in cases where phases do interact, it was not possible to automatically determine the order-dependent phases without requiring detailed knowledge of the compiler. Rather than changing the order of optimization phases, other researchers have attempted to find the best set of optimizations by turning on or off optimization flags to a conventional compiler [11, 12].

Research has also be done to find an effective optimization phase sequence by aggressive pruning and/or evaluation of only a portion of the search space. A method, called Optimization-Space Exploration [6], uses static performance estimators to reduce the search time. In order to prune the search they limited the number of configurations of optimization-parameter value pairs to those that are likely to contribute to performance improvements. This area has also seen the application of other search techniques to *intelligently* search the optimization space. Hill climbers [9, 5] and grid-based search algorithms [13] have been employed during iterative algorithms to find optimization phase sequences better than the default one used in their compilers. Other researchers have used genetic algorithms [3, 4] with aggressive pruning of the search space [14] to make searches for effective optimization phase sequences faster and more efficient.

3. Experimental Framework

The research in this paper uses the Very Portable Optimizer (VPO) [15], which was a part of the DARPA and NSF co-sponsored National Compiler Infrastructure project to produce and distribute a compiler infrastructure to be used by researchers in universities, government, and indus-

Optimization Phase	Id	Description
branch chaining	b	Replaces a branch or jump target with the target of the last jump in the jump chain.
common subexpression elimination	c	Performs global analysis to eliminate fully redundant calculations, which also includes global constant and copy propagation.
remove unreachable code	d	Removes basic blocks that cannot be reached from the function entry block.
loop unrolling	g	Loop unrolling to potentially reduce the number of comparisons and branches at runtime and to aid scheduling at the cost of code size increase.
dead assignment elim.	h	Uses global analysis to remove assignments when the assigned value is never used.
block reordering	i	Removes a jump by reordering blocks when the target of the jump has only a single predecessor.
minimize loop jumps	j	Removes a jump associated with a loop by duplicating a portion of the loop.
register allocation	k	Uses graph coloring to 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 deter.	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.
reverse branches	r	Removes an unconditional jump by reversing a conditional branch branching over the jump.
instruction selection	s	Combines pairs or triples of instructions together where the instructions are linked by set/use dependencies. After combining the effects of the instructions, it also performs constant folding and checks if the resulting effect is a legal instruction before committing to the transformation.
remove useless jumps	u	Removes jumps and branches whose target is the following positional block.

Table 1. Candidate Optimization Phases with Their Designations

try. VPO is a compiler back end that performs all its optimizations on a single low-level intermediate representation called RTLs (Register Transfer Lists) [16]. Because VPO uses a single representation, it can apply most analyses and optimization phases repeatedly and in an arbitrary order. 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 StrongARM SA-100 processor using Linux as its operating system.

Table 1 describes each of the 15 candidate code-improving phases that we used during the exhaustive exploration of our optimization phase order search space. In addition, *register assignment*, which is a compulsory phase that assigns pseudo registers to hardware registers, must be performed. VPO implicitly performs register assignment before the first code-improving phase in a sequence that requires it. Two other phases, *merge basic blocks* and *eliminate empty blocks*, were removed from the optimization list used for the exhaustive search since these phases only change the internal control-flow representation as seen by the compiler and do not directly affect the final generated code. These phases are now implicitly performed after any transformation that has the potential of enabling them. 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 runtime stack. Finally, the compiler also performs predication and instruction scheduling before generating the final output code. These last two optimizations should only be performed late in the compilation process, and so are not included in our set of phases used for exhaustive optimization space exploration.

A few dependences between some optimization phases in VPO makes it illegal for them to be performed at certain points in the optimization sequence. The first restriction is that *evaluation order determination* can only be performed before *register assignment*. This optimization is meant to reduce the number of temporaries that *register assignment* later allocates to registers. VPO also restricts some optimizations that analyze values in registers, such as *loop unrolling*, *loop strength reduction*, *induction variable elimination* and *recurrence elimination*, to be performed after *register allocation*. These phases can be performed in any order after *register allocation* is applied. *Register allocation* itself can only be performed after *instruction selection* so that candidate load and store instructions can contain the addresses of arguments or local scalars.

In our study we are only investigating the phase ordering problem and do not vary parameters for how phases should be applied. For instance, we do not attempt different configurations of loop unrolling, but always attempt it with a loop unroll factor of two since we are generating code for an embedded processor where code size can be a significant issue. The fifteen candidate code-improving phases used in VPO represent commonly used code-improving phases in compiler back ends. Many other optimizations not performed by VPO, such as loop tiling/interchange, inlining, and some other interprocedural optimizations are typically performed in a compiler frontend, and so are not present in our compiler. We also do not perform ILP (frequent path) optimiza-

tions since the ARM is typically a single issue processor and ILP transformations would be less beneficial. In addition, frequent path optimizations require a profile-driven compilation process that would complicate the study.

Note that some phases in VPO represent multiple optimizations in many compilers. However, there exists compilers, such as GCC, that have a greater number of distinct optimization phases. Unlike VPO, most compilers are much more restrictive regarding the order in which optimizations phases are performed. In addition, the more obscure a phase is, the less likely that it will be successfully applied and affect the search space. While one can always claim that additional phases can be added to a compiler or that some phases can be applied with different parameters (e.g., different unroll factors for loop unrolling), completely enumerating the optimization phase order space for the number of phases in our compiler has never before been accomplished to the best of our knowledge.

We used a subset of the *MiBench* benchmarks, which are C applications targeting specific areas of the embedded market [17], in our study. We evaluated one benchmark from each of the six categories of applications. Table 2 contains descriptions of these programs.

Category	Program	Description
auto	bitcount	test processor bit manipulation abilities
network	dijkstra	Dijkstra's shortest path algorithm
telecomm	fft	fast fourier transform
consumer	jpeg	image compression / decompression
security	sha	secure hash algorithm
office	string-search	searches for given words in phrases

Table 2. MiBench Benchmarks Used

4. Approach for Exhaustive Enumeration of the Phase Order Space

In this section we explain the approach we used to exhaustively search the space for all possible optimization phase orders. As noted earlier, an important realization is that the actual phase order space is many orders of magnitude smaller than an exhaustive enumeration of all combinations of attempted optimization phases. It is also worthwhile to note that any such attempt to enumerate all combinations of optimizations is in principle restricted by our lack of knowledge of the optimal sequence length for that particular function. The sequence length changes since optimization phases may be successful multiple times in the same sequence due to the enabling relationships between optimization phases. Previous studies have identified a large variation in the successful sequence lengths for different functions [4]. Hence, choosing a conservative sequence length might allow us to generate all possible combinations, but may not lead to an optimal phase sequence. Likewise, a long optimization sequence length would prove to be overkill for most functions and make it impractical to even enumerate attempted search spaces for smaller functions.

Interestingly, another way of viewing 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. A function instance is a version of the code that can be generated from applying a sequence of phases. This approach to the same problem clearly makes the solution much more practical. For any given function there are certainly many fewer distinct function instances than there are optimization phase orderings since different orderings can lead to the same resulting code. Two or more sequences produce the same function instance when the generated instructions are identical. So the problem now is to delineate all possible function instances that can be produced by a compiler by changing the phase order.

Our solution to this problem divides the phase ordering space into multiple levels, as shown in Figure 1 for four distinct optimization phases. At the root (level 0) we have the unoptimized function instance. For level 1, we generate the function instances produced by an optimization sequence length of 1, by applying each optimization phase individually to the base unoptimized function instance. For all other higher levels, optimization phase sequences are generated by appending each optimization phase to all the sequences at the preceding level. Thus, for each level n, we in effect generate all combinations of optimizations of length n. As can be seen from Figure 1, this space grows exponentially and would very quickly become infeasible to traverse. This exponentially growing search space can often be tractable without losing any information due to using two pruning techniques which we describe in the next two sections.

4.1. Detecting Dormant Phases

The first pruning technique exploits the fact that not all optimization phases are successful at all levels and at all positions. We call applied phases *active* when they produce changes to the program representation. A phase when applied is said to be *dormant* when it could not find any opportunities to be successful. Detecting dormant phases eliminate entire branches of the tree in Figure 1. The search space taking this factor into account can be envisioned as shown in Figure 2. The optimization phases found to be inactive are shown by dotted lines. Note that an active phase at one level is not even attempted at the next level since no phase in our compiler can be applied successfully more than once consecutively.

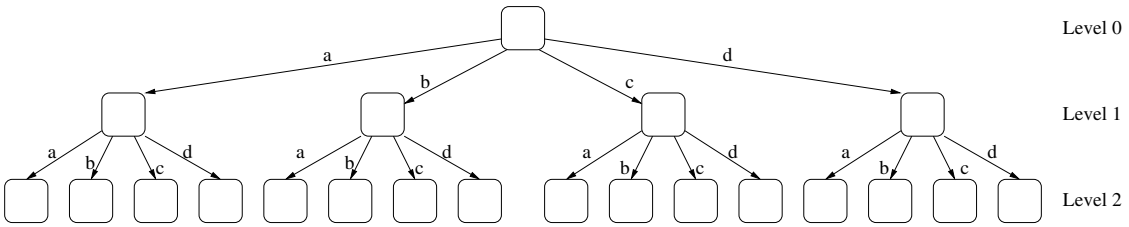


Figure 1. Naive Optimization Phase Order Space for Four Distinct Optimizations

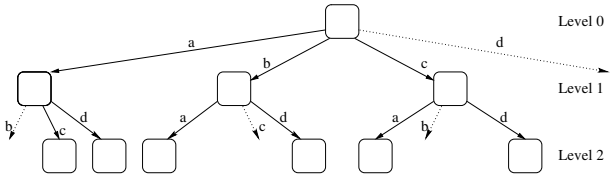


Figure 2. Effect of Detecting Dormant Phases on the Search Space in Figure 1

original code segment	**original code segment**
`r[2]=1;`	`r[2]=1;`
`r[3]=r[4]+r[2];`	`r[3]=r[4]+r[2];`
after instruction selection	**after constant propagation**
`r[3]=r[4]+1;`	`r[2]=1;`
	`r[3]=r[4]+1;`
	after dead assignment elimination
	`r[3]=r[4]+1;`

Figure 3. Diff. Opts. Having the Same Effect

4.2. Detecting Identical Function Instances

The second pruning technique relies on the assertion that many different optimizations at various levels produce function instances that are identical to those already encountered at previous levels or those generated by previous sequences at the same level. There are a couple of reasons why different optimization sequences would produce the same code. The first reason is that some optimization phases are inherently independent. For example, the order in which *branch chaining* and *register allocation* are done does not typically affect the final code. These optimizations do not share resources, are mutually complementary, and work on different aspects of the code. Secondly, different optimization phases may produce the same code. This can be seen from Figure 3. *Instruction selection* symbolically merges the instructions and checks to see if the resulting instruction is valid. In this case, the same effect can be produced by *constant propagation* (part of *common subexpression elimination* in VPO) followed by *dead assignment elimination*. The effect of different optimization sequences producing the same code is to transform the tree structure of the search space as seen in Figures 1 and 2 to a directed acyclic graph (DAG) structure as shown in Figure 4. By comparing Figures 1, 2 and 4, it is apparent how these two characteristics of the optimization search space help to make exhaustive search feasible. Note that the optimization phase order space for functions processed by our compiler is acyclic since no phase in VPO undoes the effect of another. However, a cyclic phase order space could also be exhaustively enumerated using our approach since identical function instances are detected.

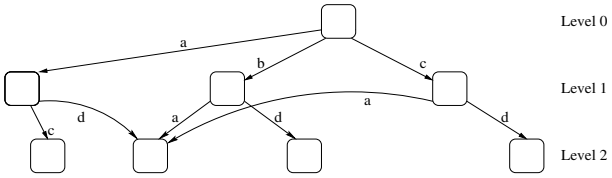


Figure 4. Detecting Identical Code Transforms the Tree in Figure 2 to a DAG

4.2.1 Efficient Detection of Identical Instances

In order to reduce the search overhead it is essential to be able to quickly compare different function instances. To check for a match we potentially need to compare each function instance with all previous function instances. A search may result in thousands of unique function instances, which may be too large to store in memory and very expensive to access on disk. Instead of comparing all function instances on a per-instruction basis every time, we calculate multiple hash values for each function instance and compare the hash values to make such comparisons efficient. For each function instance we store three numbers: a count of the number of instructions, byte-sum of all instructions, and the CRC (cyclic-redundancy code) checksum on the bytes of the RTLs in that function. This approach has been used in previous studies to detect redundant sequences when ap-

plying a genetic algorithm to search for effective phase sequences [14, 7]. CRCs are commonly used to check the validity of data transmitted over a network and have an advantage over conventional checksums in that the order of the bytes of data does affect the result [18]. CRCs are useful in our case since function instances can be identical except for different order of instructions. We have verified that when using all the three checks in combination it is extremely rare (we have never encountered an instance) that distinct function instances would be detected as identical.

From previous studies we have realized that it is possible for different function instances to be identical except for register numbers used in the instructions [14, 7]. This situation can occur since different optimization phases compete for registers. It is also possible that a difference in the order of optimizations may create and/or delete basic blocks in different orders causing them to have different labels. For example, consider the source code in Figure 5(a). Figures 5(b) and 5(c) show two possible translations given two different orderings of optimization phases that consume registers and modify the control flow.

```
sum = 0;
for (i = 0; i < 1000; i++)
   sum += a[i];
```
(a) Source Code

```
r[10]=0;
r[12]=HI[a];
r[12]=r[12]+LO[a];
r[1]=r[12];
r[9]=4000+r[12];
L3:
r[8]=M[r[1]];
r[10]=r[10]+r[8];
r[1]=r[1]+4;
IC=r[1]?r[9];
PC=IC<0,L3;
```
(b) Register Allocation before Code Motion

```
r[11]=0;
r[10]=HI[a];
r[10]=r[10]+LO[a];
r[1]=r[10];
r[9]=4000+r[10];
L5:
r[8]=M[r[1]];
r[11]=r[11]+r[8];
r[1]=r[1]+4;
IC=r[1]?r[9];
PC=IC<0,L5;
```
(c) Code Motion before Register Allocation

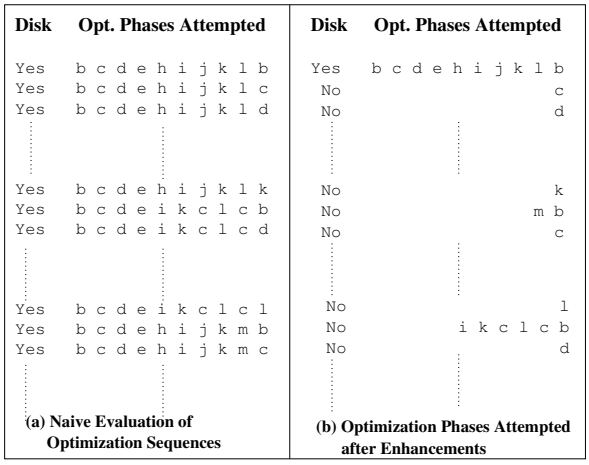
```
r[1]=0;
r[2]=HI[a];
r[2]=r[2]+LO[a];
r[3]=r[2];
r[4]=4000+r[2];
L01:
r[5]=M[r[3]];
r[1]=r[1]+r[5];
r[3]=r[3]+4;
IC=r[3]?r[4];
PC=IC<0,L01;
```
(d) After Mapping Registers

Figure 5. Different Functions with Equivalent Code

To detect this situation when calculating the CRC checksum, we map each register and block label-number to a different number depending on when it is encountered in the control flow. We start scanning the function from the top basic block. Each time a register is encountered we map it to a distinct number starting from 1. This register would keep the same mapping throughout the function. Note that this is different from register remapping of live ranges [14, 7], and is in fact much more naive. Although a complete live range register remapping might detect more instances as being equivalent, we recognize that a live range remapping at intermediate points in an optimization phase sequence would be unsafe as it changes the register pressure which might affect other optimizations applied later. During this function traversal we simultaneously remap block labels as well, which also involves mapping the labels used in the actual RTLs. Figure 5(d) shows that the same function instance is obtained after remapping of function instances 5(b) and 5(c). This approach of detecting equivalent function instances enables us to do more aggressive pruning of the search space.

4.3. Improvements for Faster Searches

During the search process we have to compile the same function with thousands of different optimization phase sequences. Evaluating every new optimization sequence involves discarding the previous compiler state (which was produced by the preceding sequence), reading the unoptimized function back from disk, applying all the optimizations in the current sequence, and then comparing the function instance produced with previous code instances. We realized that it is possible to optimize a few steps in our search algorithm to reduce the evaluation time for each optimization sequence. The first enhancement is to keep a copy of the unoptimized function instance in memory to avoid disk accesses for all optimization sequence evaluations, except the first. Our second enhancement exploits a property of our search algorithm. During the search algorithm, the sequences at any level are generated by appending each optimization phase to all active sequences at the preceding level. Thus, many optimization sequences at any level share common prefixes. By storing function instances after each active phase and by arranging the sequences to promote prefix phase sharing, it is possible to reduce the number of optimization phases applied for evaluating each sequence. This can be illustrated from Figure 6, which shows the phases that are applied before and after our enhancements. We found that these enhancements reduced the search time at least by a factor of 5 to 10.



(a) Naive Evaluation of Optimization Sequences

(b) Optimization Phases Attempted after Enhancements

Figure 6. Enhancements for Faster Searches

4.4. Search Space Enumeration Results

Table 3 displays results when enumerating the actual search space for each function by our compiler. The functions are sorted by the number of instructions (*Insts*) in the unoptimized version of the function. The results clearly illustrate that while our technique would seem to only apply to small functions, it also works well for many large functions. Note that our compiler optimizes each function individually and in isolation of all others in the same program. Since it was not possible to list information for all the 111 functions we evaluated, we show the averages for the functions we could not display. During this set of experiments we terminated the search any time the number of optimization sequences to apply at any particular level (see Figure 4) grew to more than a million. For such functions we mark the search space as too big to be exhaustively enumerated in a reasonable amount of time. We found that we could not completely enumerate the space for only two of the 111 functions, which are indicated in Table 3.

Studying the results in Table 3 allows us to make many interesting observations about the optimization space. The number of attempted phases is larger than the number of function instances in Table 3 since the compiler also needs to attempt phases to detect when they are dormant. The numbers in the column of maximum active phase sequence length indicate that the optimization phase order search space is 15^{12} on average, and can grow to 15^{32} in the worst case for the compiler and benchmarks used in this study. Thus, we can also see that although the attempted search space is extremely large, the number of distinct function instances is only a tiny fraction of this number. This is precisely the reason which makes our technique of exhaustive phase order search space enumeration possible. The leaf function instances are those for which no further phases in our compiler are successful in changing the program representation. The small number of leaf instances in each case suggests that after getting considerably wide, the optimization space dag (see Figure 4) typically converges later. The last three columns show the range of the static number of instructions for the leaf function instances. This number is significant as it illustrates the maximum difference in code size that is possible due to different phase orderings in our compiler. Thus, we can see that on average there is a gap of 37.8% in code size between the best and the worst phase ordering. The numbers in the table also suggest that although functions with more instructions have larger search spaces, it is the control-flow characteristics of the function, more than the number of instructions, that determine the width of the DAG. Many large functions with simple control flows have search spaces which are deep, instead of being wide, which allows faster enumeration of the space.

5. Optimization Phase Interaction Analysis

Exhaustive enumeration of the optimization phase order search space for a sizable number of functions has given us a large data set that we can analyze to gather information about how optimization phases interact. In this section we present some of our analysis results. To assemble these statistics we first represented the search space in the form of a DAG. The nodes in the DAG represent distinct function instances and the arcs are marked by the optimization phase that was applied from one node (function instance) to the next. This representation is illustrated in Figure 7. The nodes of the DAG are weighted by their position in the DAG and the number of children that it can have. The leaf nodes have a weight of 1. The weight of each interior node is the summation of the weights of all its child nodes. Thus, the weight of each interior node gives the number of distinct sequences that are active beyond that point. Active phases at each node (indicated in brackets for interior nodes) in Figure 7 are simply the active phases that are applied on outgoing edges of that node. We studied three different phase interactions: enabling, disabling and independence relationships between optimization phases. The following sections describe the results of this study.

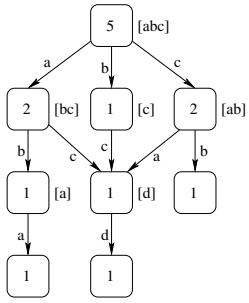


Figure 7. Weighted DAG Showing Enabling, Disabling, and Independence Relations

5.1. Enabling Interaction between Phases

A phase *x* is said to enable another phase *y*, if *y* was dormant (inactive) at the point just before *x* was applied, but then becomes active again after application of *x*. For example, *b* enables *a* along the path *a-b-a* in Figure 7. Note that it is possible that a phase could enable some other phase on some sequence but not on others. Thus, it could be seen that *a* is not enabled by *b* along the path *c-b*. Likewise, it is also possible for phases to be dormant at the start of the compilation process, and become active later (e.g., phase *d* along the path *b-c-d*). As a result we represent this information in the form of the probability of each phase enabling

Function	Insts	Blk	Brch	Loop	Fn_inst	Attempt Phases	Len	CF	Leaf	Codesize		
										Max.	Min.	% Diff
start_inp...(j)	1,371	88	69	2	74,950	1,153,279	20	153	587	463	426	8.7
parse_swi...(j)	1,228	198	144	1	200,397	2,990,221	18	53	2365	592	490	20.8
start_inp...(j)	1,009	72	55	1	39,152	597,147	16	18	324	370	285	29.8
start_inp...(j)	971	82	67	1	64,571	999,814	18	47	591	319	301	6.0
start_inp...(j)	795	63	50	1	7,018	106,793	15	37	52	281	259	8.5
fft_float(f)	680	45	32	4	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
main(f)	624	50	35	5	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
sha_trans...(h)	541	33	25	6	343,162	5,119,947	26	95	2964	280	138	102.9
read_scan...(j)	480	59	52	2	34,270	511,093	15	57	540	230	164	40.2
LZWRea...(j)	472	44	33	2	49,412	772,864	20	41	159	210	138	52.2
main(j)	465	40	28	1	33,620	515,749	17	12	153	156	151	3.3
dijkstra(d)	354	30	22	3	86,370	1,361,960	20	18	1168	165	91	81.3
usage(j)	344	3	1	0	34	511	7	1	3	81	79	2.5
GetCode(j)	339	14	11	1	56,166	850,977	18	20	75	94	85	10.6
bmhi_init(s)	309	30	22	4	10,235	156,378	20	11	145	166	90	84.4
preload_i...(j)	268	29	22	2	5,356	80,715	13	10	38	132	85	55.3
get_inter...(j)	249	20	17	1	16,880	258,690	19	10	78	90	58	55.2
bmha_init(s)	248	31	22	3	32,446	486,024	20	56	438	143	66	116.7
read_quan...(j)	239	25	21	2	8,016	119,749	14	28	304	109	79	38.0
load_inte...(j)	235	19	13	2	3,516	52,419	13	8	14	111	60	85.0
main(b)	220	22	15	2	92,834	1,367,101	21	91	171	92	77	19.5
get_word_...(j)	220	11	7	1	1,882	29,563	15	4	53	98	44	122.7
read_colo...(j)	218	20	16	2	1,166	18,035	14	8	24	162	91	78.0
set_sampl...(j)	209	38	33	2	22,735	337,928	21	53	279	112	79	41.8
bmha_search(s)	201	29	24	3	659,222	9,937,042	32	468	2779	98	51	92.2
bmh_init(s)	194	21	15	3	5,693	86,157	20	11	137	108	54	100.0
bmhi_search(s)	184	29	24	3	232,367	3,466,408	27	331	965	88	49	79.6
bmh_search(s)	181	29	24	3	400,805	5,977,825	27	601	1643	93	48	93.8
main(s)	175	19	12	3	30,975	477,277	19	12	175	81	70	15.7
main(d)	175	21	15	3	8,566	127,688	18	28	143	86	68	26.5
get_8bit_...(j)	171	8	5	1	2,345	35,412	13	4	33	66	34	94.1
get_scale...(j)	166	11	7	1	1,139	17,690	15	4	29	70	37	89.2
get_16bit...(j)	158	8	5	1	844	13,020	12	4	25	62	29	113.8
set_quant...(j)	156	30	25	2	10,245	151,068	15	49	59	86	60	43.3
preload_i...(j)	156	14	9	1	664	10,053	10	8	9	73	44	65.9
sha_final(h)	155	7	4	0	1,738	26,457	13	3	64	39	33	18.2
select_fi...(j)	149	25	21	0	400	6,118	10	10	12	72	68	5.9
byte_reve...(h)	146	8	5	1	1,661	24,898	13	11	24	68	26	161.5
get_24bit...(j)	145	8	5	1	2,002	30,194	12	4	39	54	29	86.2
get_word_...(j)	144	11	7	1	2,043	32,660	15	4	49	60	33	81.8
read_text...(j)	141	21	16	2	48,301	728,287	17	34	358	66	42	57.1
read_scan...(j)	139	27	22	1	42,712	648,832	19	47	207	54	48	12.5
ntbl_bitc...(b)	138	3	1	0	48	720	7	1	8	30	27	11.1
read_pbm_...(j)	134	27	21	2	3,174	48,461	13	19	42	66	50	32.0
remaining 67	60.4	7.6	4.5	0.4	1,870.2	28,178.6	8.8	7.7	38.8	24.0	17.6	24.0
average	166.7	16.9	12.0	0.9	25,362.6	381,857.7	12.0	27.5	182.9	69.0	49.9	37.8

Function - function name followed by benchmark indicator [(s)-stringsearch, (b)-bitcount, (h)-sha, (f)-fft, (d)-dijkstra, (j)-jpeg], Inst - number of instructions in the unoptimized function, Blk - number of basic blocks, Brch - number of conditional and unconditional transfers of control, Loop - number of loops, Fn_inst - number of distinct function instances, Attempt Phases - number of optimization phases attempted, Len - largest active optimization phase sequence length, CF - number of distinct control flows, Leaf - number of leaf function instances, Codesize - max., min., and % difference in number of instructions for leaf function instances, N/A - search space of function exceeded our search criteria limit

Table 3. Function-Level Search Space Statistics for MiBench Benchmarks Used in the Experiments

each other phase. We calculate the enabling probabilities by considering *dormant* → *active* and *dormant* → *dormant* transitions of phases between nodes, adjusted by the weight of the child node. The probability is the ratio of the number of *dormant* → *active* transitions to the sum of *dormant* → *active* and *dormant* → *dormant* transitions between optimization phases. We do not consider *active* → *active* and *active* → *dormant* transitions since phases already active cannot be enabled. We summarize the enabling information we collected for all the functions in Table 4, where each row represents the probability of that phase being enabled by other phases represented in columns.

A few points regarding the enabling information are worth noting. For our benchmarks *instruction selection*(s) and *common subexpression elimination*(c) are always active initially. In contrast, *register allocation*(k) requires *instruction selection*(s) to be enabled in VPO so that the loads and stores contain the addresses of local scalars. *Instruction selection*(s) is frequently enabled by *register allocation*(k) since loads and stores are replaced by register-to-register moves, which can typically be collapsed by *instruction selection*(s). In contrast, control flow optimizations (e.g., *branch chaining*(b)) are never enabled by *register allocation*(k), which does not affect the control flow. The numbers in the table also indicate that many optimizations have a very low probability of being enabled by any other optimization phase. Such optimizations will typically be active at most once in each optimization sequence. *Remove unreachable code*(d) is never active for the functions in our benchmark suite, which indicates the need for a larger set of functions. Note that, unreachable code occasionally left behind by *branch chaining* is removed during *branch chaining* itself, since we found such code hindering some analysis which caused later optimizations to miss some code improving opportunities.

5.2. Disabling Interaction between Phases

Another related measure is the probability of each phase disabling some other phase. This relationship can be seen in Figure 7 along path *b*-*c*-*d*, where *a* is active at the root node, but is disabled after *b*. The statistics regarding the disabling interaction between optimization phases is illustrated in Table 5. Each value in this table is the weighted ratio of *active* → *dormant* transitions to the sum of *active* → *dormant* and *active* → *active* transitions. We do not consider *dormant* → *dormant* and *dormant* → *active* transitions since a phase has to be active to be disabled.

From Table 5 it can be seen that phases are much more likely to be disabled by themselves than by other phases. We can also see that phases such as *register allocation*(k) and *common subexpression elimination*(c) always disable *evaluation order determination*(o) since they require *register assignment*, and *evaluation order determination* can only be performed before *register assignment*. In our test suite all jumps eliminated by *useless jump elimination*(u) are always also eliminated by *block reordering*(i). Thus, disabling information could be used to detect when one phase may always subsume another.

5.3. Optimization Phase Independence

The third interaction we measured was the probability of independence between any two optimization phases. Two phases can be considered to be independent if their order does not matter to the final code that is produced. This is illustrated in the Figure 7 along the paths *a*-*c* and *c*-*a*. Both orders of phases *a* and *c* in these sequences produce identical function instances, which would mean that they are independent in this situation. In contrast, sequences *b*-*c* and *c*-*b* do not produce the same code. Thus, they are considered dependent in this situation. If two optimizations are detected to be completely independent, then we would never have to evaluate them in different orders. This observation can lead to the potential of even greater pruning of the search space. Table 6 shows the probability of each phase being independent of some other phase. This is a weighted ratio of the times two consecutively active phases produced the same code to the number of times they were consecutively active.

Unlike the enabling and disabling relationships shown in Tables 4 and 5, independence is a symmetric relationship, as shown in Table 6. In addition, Table 6 is less sparse indicating that many phases are typically independent of each other. For instance, it can be seen that *register allocation*(u) is highly independent of most control flow optimizations. *Instruction selection*(s) and *common subexpression elimination*(c) frequently act on the same code sequences, and so we see a low level of independence between them. Since most of the phases are independent of each other most of the time it is frequently possible to reorder phases without any side-effect. Consequently, many different optimization sequences produce the same code resulting in greater convergence in the DAG and fewer leaf function instances for most functions, as seen from Table 3.

6. Probabilistic Batch Optimization

The analysis results and observations assembled during our experiments can be further used to improve upon various compiler features. As a case study, we use some of these results to support faster compilations in this section. The VPO compiler applies optimization phases to all functions in one default order. To allow aggressive optimizations, VPO applies many optimization phases in a loop until there are no further program changes produced by any

Phase	St	b	c	d	g	h	i	j	k	l	n	o	q	r	s	u
b	0.62		0.01		0.15	0.06										
c	1.00	0.02			0.23	0.14		0.12	0.99	0.72	0.38	0.33	1.00	0.05	0.32	
d																
g		0.01							0.18							0.01
h	0.06		0.70		0.02				0.01	0.03					0.46	
i	0.61	0.01	0.01					0.61								
j	0.03	0.01			0.13											
k			0.01			0.11									0.81	
l	0.59		0.06		0.02	0.01			0.03						0.06	
n	0.42		0.04		0.22	0.01	0.04			0.01		0.01		0.03	0.05	0.03
o	0.87														0.01	
q			0.16												0.08	
r	0.45	0.02			0.15					0.05	0.01					
s	1.00		0.29		0.16	0.23			0.97	0.53	0.20		1.00			
u		0.73			0.03											

Blank cells indicate an enabling probability of less than 0.005. St represents the probability of a phase being active at the start of compilation.

Table 4. Enabling Interaction between Optimization Phases

Phase	b	c	d	g	h	i	j	k	l	n	o	q	r	s	u
b	1.00			0.15		0.02				0.08			0.05		0.31
c		1.00			0.02									0.15	
d			1.00												
g	0.35			1.00		0.19	0.02							0.03	
h		0.01			1.00										
i	0.08			0.06		1.00	0.14						0.14		0.55
j						0.13	1.00						0.49		0.14
k		0.04		0.01				1.00							
l		0.71			0.07			0.30	1.00					0.73	
n	0.33	0.49		0.09	0.25		0.07	0.31	0.53	1.00	0.02			0.33	
o		1.00			1.00			1.00	1.00	1.00	1.00			0.21	
q												1.00		0.12	
r	0.05			0.01	0.03	0.53							1.00		
s		0.11												1.00	
u	0.08					1.00	0.20								1.00

Blank cells indicate a disabling probability of less than 0.005.

Table 5. Disabling Interaction between Optimization Phases

Phase	b	c	d	g	h	i	j	k	l	n	o	q	r	s	u
b				0.84		0.94		0.97	0.95	0.82			0.96		0.95
c				0.96	0.91			0.45	0.44	0.65	0.12	0.98	0.99	0.22	
d															
g	0.84	0.96			0.98	0.84			0.96	0.98				0.96	
h		0.91		0.98				0.79	0.95	0.88	0.59		0.98	0.96	
i	0.94			0.84			0.98	0.97	0.96				0.71		0.50
j						0.98							0.97		0.98
k	0.97	0.45			0.79	0.97			0.87	0.81	0.30		0.99	0.82	0.97
l	0.95	0.44		0.96	0.95	0.96		0.87		0.78	0.45			0.45	
n	0.82	0.65		0.98	0.88			0.81	0.78		0.58			0.61	
o		0.12			0.59			0.30	0.45	0.58				0.39	
q		0.98												0.89	
r	0.96	0.99			0.98	0.71	0.97	0.99						0.94	
s		0.22		0.96	0.96			0.82	0.45	0.61	0.39	0.89	0.94		
u	0.95					0.50	0.98	0.97							

Blank cells indicate an independence probability of greater than 0.995.

Table 6. Independence Relationship between Optimization Phases

optimization phase. Thus, although VPO can attempt a different number of phases for different functions, the order in which they are attempted still remains the same. Applying optimizations in a loop also means that many optimization phases when attempted are dormant.

We use information about the probability of phases enabling and disabling each other to dynamically select optimizations phases depending on which previous ones were active. The probability of each optimization phase being active by default is used at the start of the optimization process. Using these probabilities as initial values, we dynamically determine which phase should be applied next depending on which phase has the highest probability of being active. After each active optimization phase, we update the probabilities of all other phases depending on the probability that the last phase would enable or disable it. This algorithm is depicted in Figure 8. We denote the compiler using this new algorithm of dynamically selecting optimization phases as the *probabilistic batch compiler*.

```
# p[i] - current probability of phase i being active
# e[i][j] - probability of phase j enabling phase i
# d[i][j] - probability of phase j disabling phase i
foreach phase i do
    p[i] = e[i][st];    # start phase probabilities (see Table 4)
while any p[i] > 0 do
    Select j as the current phase with highest probability of
    being active;
    Apply phase j;
    if j was active then
        foreach phase i do
            if i != j then
                p[i] += ((1-p[i]) * e[i][j]) - (p[i] * d[i][j]);
    p[j] = 0;
```

Figure 8: **Probabilistic Compilation Algorithm**

From Table 7 it can be seen that the new probabilistic mode of compilation achieves performance comparable to the old batch mode of compilation and requires less than one-third of the compilation time on average. Although the probabilistic approach reduces the number of attempted phases from 230, on average, to just 48, the number of active phases is in fact greater in the new approach. Many phases attempted in the old compiler were found by the probabilistic compiler to be disabled and were therefore not attempted. Presently, the probabilistic compiler selects the next phase only on the basis of the probability of it being active. Our method does not consider the benefits each phase can potentially provide when applied. This is the main reason for the slight degradation in performance, on average, over the old method. Thus, the probabilistic compilation paradigm, even though promising, can be further improved by taking phase benefits into account.

7. Future Work

There is a variety of enhancements that we plan to make in the future. First, we would like to examine methods to further speed up the enumeration algorithm. The phase order space can be reduced by changing the implementation of some compiler analysis and optimizations, so that false dependences due to incomplete analysis are no longer present. Phase interaction information, such as independence relationships, could also be used to more aggressively prune the enumeration space. Second, we plan to improve nonexhaustive searches of the phase order space. The enabling/disabling relationships between phases could be used for faster genetic algorithm searches [14]. Presently the only feedback we get from each optimization phase was whether it was active or dormant. We do not keep track of the number and type of actual changes for which each phase is responsible. Keeping track of this information would be very useful to get more accurate phase interaction information. Third, we plan to study additional methods to enhance conventional compilers for both compilation time and the efficiency of generated code by using probabilistic phase interaction relationships in determining the next phase to be applied. Finally, the eventual goal is to find the function instance giving near-optimal execution performance in a reasonable amount of time. Achieving this goal requires gathering execution results, which would be very time consuming when there are hundreds of thousands of executions required. The small number of distinct control flows of functions (see column *CF* in Table 7) can be used to infer the dynamic instruction count of one execution from another. Dynamic instruction counts, unlike cycle counts, are a crude approximation of execution efficiency. However, these counts could be used to prune function instances from being simulated that are likely to produce inefficient code.

8. Conclusions

The phase ordering problem has been an old and as yet unresolved problem in compiler optimizations. Hitherto, it was assumed that the optimization phase order space is too large to be completely enumerated in a reasonable amount of time. In this paper we have shown, for most of the functions in our benchmark suite, that exhaustive enumeration of all optimization phase sequences is possible for the phases in our compiler. This enumeration was made possible by detecting which phases were active and whether or not the generated code was unique, making the actual optimization phase order space much smaller than the attempted space. Using an innovative enumeration algorithm and aggressive search space pruning techniques, we were able to find all possible function instances that can be produced by different phase orderings for 109 out of the 111 total func-

Function	Old Compilation			Prob. Compilation			Prob/Old		
	Attempted Phases	Active Phases	Time	Attempted Phases	Active Phases	Time	Time	Size	Speed
start_inp...(j)	233	16	3.10	55	14	1.45	0.469	1.014	N/A
parse_swi...(j)	233	14	7.52	53	12	2.79	0.371	1.016	0.972
start_inp...(j)	270	15	2.20	55	14	0.78	0.353	1.010	N/A
start_inp...(j)	233	14	1.83	49	13	0.77	0.420	1.003	N/A
start_inp...(j)	231	11	1.21	53	12	0.53	0.436	1.004	1.000
fft_float(f)	463	28	2.65	99	25	1.20	0.451	1.012	0.974
main(f)	284	20	1.83	73	18	1.01	0.550	1.007	1.000
sha_trans...(h)	284	17	0.68	67	16	0.41	0.605	0.965	0.953
read_scan...(j)	233	13	0.99	43	10	0.34	0.342	1.018	N/A
LZWReadByte(j)	268	12	0.64	45	11	0.21	0.325	1.014	N/A
main(j)	270	12	1.04	57	14	0.39	0.375	1.007	1.000
dijkstra(d)	231	9	0.37	43	9	0.15	0.409	1.010	1.000
usage(j)	188	3	0.17	47	6	0.07	0.428	1.025	N/A
GetCode(j)	270	15	0.36	61	13	0.18	0.508	1.000	N/A
bmhi_init(s)	231	10	0.34	53	11	0.15	0.440	1.011	N/A
preload_i...(j)	233	12	0.33	43	12	0.11	0.337	1.012	N/A
bmha_init(s)	268	12	0.40	85	17	0.28	0.704	1.024	N/A
get_inter...(j)	233	14	0.21	51	12	0.09	0.401	0.967	N/A
read_quan...(j)	233	9	0.32	43	10	0.10	0.317	1.013	N/A
load_inte...(j)	233	11	0.22	43	12	0.08	0.349	1.000	N/A
main(b)	233	14	0.29	59	14	0.14	0.489	1.026	1.000
get_word_...(j)	233	10	0.18	51	11	0.07	0.377	0.918	N/A
read_colo...(j)	233	12	0.21	57	13	0.12	0.593	0.986	N/A
set_sampl...(j)	231	13	0.24	61	12	0.11	0.474	1.012	N/A
bmha_search(s)	231	11	0.17	63	14	0.08	0.495	1.078	N/A
bmh_init(s)	231	8	0.17	53	11	0.07	0.440	1.000	N/A
bmhi_search(s)	231	10	0.16	49	12	0.07	0.398	1.082	N/A
bmh_search(s)	231	11	0.16	63	14	0.08	0.484	1.083	N/A
main(s)	268	14	1.99	67	14	1.54	0.774	1.013	1.000
main(d)	245	13	0.20	59	13	0.09	0.438	1.015	1.000
get_8bit_...(j)	233	11	0.15	51	11	0.05	0.366	0.944	N/A
get_scale...(j)	233	10	0.15	51	11	0.05	0.349	1.000	N/A
get_16bit...(j)	233	9	0.13	51	11	0.05	0.405	1.034	N/A
preload_i...(j)	231	10	0.15	45	11	0.04	0.297	1.000	N/A
sha_final(h)	233	8	0.13	41	9	0.04	0.276	1.000	1.000
set_quant...(j)	233	15	0.20	55	12	0.07	0.367	1.016	N/A
select_fi...(j)	231	9	0.25	43	10	0.07	0.289	1.029	1.071
byte_reve...(h)	270	11	0.15	75	15	0.11	0.722	1.030	1.000
get_24bit...(j)	233	10	0.13	51	11	0.04	0.344	1.000	N/A
get_word_...(j)	233	9	0.13	51	11	0.04	0.321	0.943	N/A
ntbl_bitc...(b)	188	4	0.11	39	7	0.04	0.363	1.037	1.036
read_text...(j)	268	14	0.18	67	14	0.07	0.395	1.000	N/A
read_scan...(j)	231	12	0.15	57	13	0.06	0.390	1.061	N/A
bitcount(b)	188	5	0.09	37	6	0.02	0.263	1.333	1.321
remaining 67	217.9	6.9	0.09	42.1	7.6	0.02	0.206	0.998	0.996
average	230.3	8.9	0.34	47.7	9.6	0.14	0.297	1.015	1.005

Old Compilation - original batch compilation, Prob. Compilation - new probabilistic mode of compilation, Attempted Phases - number of attempted phases, Active Phases - number of active phases, Time - compilation time in seconds, Prob/Old - ratio of probabilistic to old compilation for compilation time, code size, and dynamic instruction counts, respectively.

Table 7. Comparison between the Old Batch and the New Probabilistic Approaches of Compilation

tions we evaluated. It is now possible to find the optimal phase ordering for some characteristics. For instance, we are able to find the minimal code size for most of the functions in our benchmark suite. This enumeration study also provided us with a large data set, which we showed can be analyzed to study various optimization phase interactions. Automatically calculating probabilities is much more reliable than relying on a compiler writer's intuition since it is very difficult for even the most experienced compiler writer to envision all of the possible interactions between phases [14]. We used the enabling/disabling probabilities to reduce the compilation time for a conventional compiler to about one-third of the time originally required, while maintaining comparable performance. We have also described several other potential applications of using phase interaction information. In summary, we believe that our approach for efficient and exhaustive enumeration of optimization phase order space has opened a new area of compiler research.

9. Acknowledgements

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by NSF grants EIA-0072043, CCR-0208892, CCR-0312493, CCF-0444207, and CNS-0305144.

References

[1] Steven 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.

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

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

[4] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, Compiler, and Tool for Embedded Systems*, pages 12–23. ACM Press, 2003.

[5] T. Kisuki, P. Knijnenburg, and M.F.P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proc. PACT*, pages 237–246, 2000.

[6] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *Proceedings of the international symposium on Code Generation and Optimization*, pages 204–215. IEEE Computer Society, 2003.

[7] Prasad A. Kulkarni, Stephen R. Hines, David B. Whalley, Jason D. Hiser, Jack W. Davidson, and Douglas L. Jones. Fast and efficient searches for effective optimization-phase sequences. *ACM Trans. Archit. Code Optim.*, 2(2):165–198, 2005.

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

[9] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd 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, New York, NY, USA, 2004. ACM Press.

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

[11] Elana D. Granston and Anne Holler. Automatic recommendation of compiler options. 4th Workshop of Feedback-Directed and Dynamic Optimization, December 2001.

[12] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizatons. Proc. 2nd Workshop on Feedback Directed Optimization, 1999.

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

[14] 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, June 2004.

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

[16] M. E. Benitez and J. W. Davidson. The advantages of machine-dependent global optimization. In *Proceedings of the 1994 International Conference on Programming Languages and Architectures*, pages 105–124, March 1994.

[17] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[18] W. Peterson and D. Brown. Cyclic codes for error detection. In *Proceedings of the IRE*, volume 49, pages 228–235, January 1961.