Practical Exhaustive Optimization Phase Order Exploration and Evaluation

PRASAD A. KULKARNI
University of Kansas, Lawrence, Kansas
DAVID B. WHALLEY, GARY S. TYSON
Florida State University, Tallahassee, Florida
JACK W. DAVIDSON
University of Virginia, Charlottesville, Virginia

Choosing the most appropriate optimization phase ordering has been a long standing problem in compiler optimizations. Exhaustive evaluation of all possible orderings of optimization phases for each function is generally dismissed as infeasible for production-quality compilers targeting accepted benchmarks. In this paper we show that it is possible to exhaustively evaluate the optimization phase order space for each function in a reasonable amount of time for most of the functions in our benchmark suite. To achieve this goal we used various techniques to significantly prune the optimization phase order search space so that it can be inexpensively enumerated in most cases, and to reduce the number of program simulations required to evaluate program performance for each distinct phase ordering. The techniques described are applicable to other compilers in which it is desirable to find the best phase ordering for most functions in a reasonable amount of time. We also describe some interesting properties of the optimization phase order space, which will prove useful for further studies of related problems in compilers.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*; D.4.7 [**Operating Systems**]: Organization and Design—*realtime systems and embedded systems, interactive*

Extension of Conference Paper: Preliminary versions of this research appeared in the *2006 International Symposium on Code Generation and Optimization (CGO)* under the title "Exhaustive Optimization Phase Order Exploration", and in the *2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)* under the title "In Search of Near-Optimal Optimization Phase Orderings".
The current work differs from both of these earlier papers in several aspects:

(1) We introduce a completely new search algorithm (Section 5) to more efficiently find the successive phase order sequences to evaluate. We also note the trade-offs in choosing any one algorithm over the other.
(2) We have doubled our benchmark set from 6 to 12 benchmarks, and more than doubled the number of studied functions, from 111 to 244. Many of the newer functions added are significantly larger, making our switch to the new search algorithm more critical.
(3) A new section (Section 8) presents interesting results from analyzing the exhaustive phase order space over the entire set of functions. These results, which are shown in Figures 11 through 19, required a significant amount of time to collect.
(4) All the remaining sections in this paper also go into more depth about the issues involved in the development of the search algorithm, pruning techniques, and other implementation issues.

This research was supported in part by NSF grants EIA-0072043, CCR-0208892, CCR-0312493, CCF-0444207, and CNS-0305144.

systems

General Terms: Performance, Measurement, Algorithms

Additional Key Words and Phrases: Phase Ordering, Exhaustive Search, Iterative Compilation

1. INTRODUCTION

Current optimizing compilers typically contain several different optimization *phases*. Each phase attempts to apply a series of *transformations*, each of which consists of a sequence of changes that preserves the semantic behavior of the program while typically improving its efficiency. Many of these optimization phases use and share resources (such as machine registers), and also need specific conditions in the code to be applicable. As a result, optimization phases interact with each other by enabling and disabling opportunities for other phases to be applied. Such interactions between optimization phases have been widely studied in the context of different compilers (with different sets of optimization phases) and different architectures [Whitfield and Soffa 1997; Almagor et al. 2004; Kisuki et al. 1999; Kulkarni et al. 2006a]. Based on such studies it is now definitively known that phase interaction often causes different orders of applying optimizations phases to produce different output code, with potentially significant performance variation. Therefore, finding the best order of applying optimization phases is important for application areas where developers are willing to wait for longer compilations, such as in high performance and embedded domains, so that more efficient code can be generated for each application. This challenge is commonly known as the *phase ordering problem* in compilers. Over four decades of research on the phase ordering problem has shown that the problem is difficult since a single order of optimization phases will not produce optimal code for every application [Vegdahl 1982; Whitfield and Soffa 1990; Cooper et al. 1999; Kulkarni et al. 2003; Triantafyllis et al. 2003; Kisuki et al. 1999]. The best order depends on the program being optimized, the manner in which the optimizations are implemented in the compiler, and the characteristics of the target machine.

A naive solution to the phase ordering problem is to exhaustively evaluate the performance of all possible orderings of optimization phases. This approach requires the resolution of two sub-problems, both of which have always been considered infeasible for production-quality compilers. The first sub-problem is to exhaustively *enumerate* all possible orderings of optimization phases. This enumeration is difficult since the phase ordering space has a tendency to quickly become impractical to completely explore in the face of several different optimization phases that are typically present in current compilers, with few restrictions on the ordering of these phases. The second sub-problem is to *evaluate* the performance of all the enumerated orderings to find the best performance. To achieve the desired accuracy, performance evaluation generally requires execution of the application, which is typically much more expensive than simply compiling the code. Many low-end embedded systems are unable to support a full-fledged compilation environment, which implies that the software development activity occurs external to the embedded device [Barr and Massa 2006]. Development activity for such systems often proceeds via simulation instead of native execution [Engblom 2007; Redhat 2004; Virtutech 2008], which is typically orders of magnitude more expensive. Thus, it is hardly surprising that exhaustive phase order space evaluation over all the optimization phases in

a mature compiler has never been successfully accomplished.

In this paper we show that by using various pruning techniques it is possible to exhaustively evaluate all possible orderings of optimization phases for our compiler targeting the ARM processor, and determine the best or *optimal* performing phase orderings with a very high degree of probability for most of the functions in our benchmark suite. Note that a different compiler, with a different or greater set of optimization phases can possibly generate better code than the *optimal* instance produced by our compiler. Thus, *optimal* in the context of this work refers to **the best code that can be produced by any optimization phase ordering in our compiler (VPO) that applies optimizations on a per-function basis, using identical parameters for all applications of each phase, on the ARM processor platform, and on the benchmark and input data set considered in our study**, and is not meant to imply a universally optimal solution. We describe the techniques we used to prune the phase order search space to generate all possible *distinct function instances* that can be produced by changing the optimization phase ordering in our compiler, for the vast majority of the functions that we studied. This paper also explains our approach of using simple and fast estimation techniques to reduce the number of simulations and yet determine the optimal performing function instance with a high degree of accuracy. We have used various correlation techniques to illustrate that our method of performance estimation is highly accurate for our purposes. Finally, exhaustive evaluation of the phase order space over a large number of functions has given us a large data set, which we have analyzed to determine various properties of the optimization space. Some of these results are presented in this paper.

The remainder of this paper is organized as follows. In the next section we review the previous work related to this topic. In Section 3 we give an overview of our compilation framework. In Section 4 we explain our techniques to exploit redundancy in the phase order space, and to reduce the number of simulations required to determine the performance of all distinct phase orderings. Our implementation of these techniques in the context of our experimental framework is described in Section 5. Our experimental results are presented in Section 6. In Section 7 we use two methods to demonstrate the strong correlation between our estimates of performance and actual simulation cycles. In Section 8 we conduct an analysis of the optimization phase order space and present some interesting results. The final two sections present directions for future work and our conclusions respectively.

2. RELATED WORK

As mentioned earlier, optimization phase ordering is a long standing problem in compilers and as such there is a large body of existing research on this topic. An interesting study investigating the decidability of the phase ordering problem in optimizing compilation proved that finding the optimal phase ordering is *undecidable* in the general schemes of iterative compilation and library generation/optimization [Touati and Barthou 2006]. However, their hypothesis assumes that the set of all possible programs generated by distinct phase orderings is infinite. This hypothesis is rational since optimizations such as loop unrolling, and strip mining [Hewlett-Packard 2000] can be applied an arbitrary number of times, and can generate as many distinct programs. In practice, however, compilers typically impose a restriction on the number of times such phases can be repeated in a normal optimization sequence. Additionally, most other optimizations are targeted to remove some program inefficiency, and/or exploit some architectural feature, which in turn limits

the number of times such phases can be active for any application. We explain this point in greater detail in Section 3, where we describe the experimental settings used for the results in this paper. Thus, finding the best phase ordering is decidable in most current compilers, albeit very hard.

Strategies to address the phase ordering problem generally pursue one of two paths: a model-driven approach or an empirical approach. A model-driven or analytical approach attempts to determine the properties of optimization phases, and then use some of these properties at compile time to decide what phases to apply and how to apply each phase. Such approaches have minimal overhead since additional profile runs of the application are generally not required. Whitfield and Soffa developed a framework based on axiomatic specifications of optimizations [Whitfield and Soffa 1990; 1997]. This framework was employed to theoretically list the *enabling* and *disabling* interactions between optimizations, which were then used to derive an application order for the optimizations. The main drawback was that in cases where the interactions were ambiguous, it was not possible to automatically determine a good ordering without detailed information about the compiler. Follow-up work on the same topic has seen the use of additional analytical models, including code context and resource (such as cache) models, to determine and predict other properties of optimization phases such as the *impact* of optimizations [Zhao et al. 2003], and the *profitability* of optimizations [Zhao et al. 2005]. Even after substantial progress, the fact remains that properties of optimization phases, as well as the relations between them are, as yet, poorly understood, and model-driven approaches find it hard to predict the best phase ordering in most cases.

With the growth in computation power, researchers have commonly adopted *empirical* approaches that use multiple program runs to search for the best phase ordering. Exhaustive evaluation of the entire optimization phase order space has generally been considered infeasible, and has never been successfully attempted prior to our work. Enumerations of search spaces over a small subset of available optimizations have, however, been attempted [Almagor et al. 2004]. This work exhaustively enumerated a 10-of-5 subspace (optimization sequences of length 10 from 5 distinct optimizations) for some small programs. Each of these enumerations typically required several processor months even for small programs. The researchers found the search spaces to be neither smooth nor convex, making it difficult to predict the best optimization sequence in most cases.

Researchers have also investigated the problem of finding an effective optimization phase sequence by aggressive pruning and/or evaluation of only a portion of the search space. This area has seen the application of commonly employed *artificial intelligence* search techniques to search the optimization space. Hill climbers [Almagor et al. 2004; Kisuki et al. 2000], grid-based search algorithms [Bodin et al. 1998], as well as genetic algorithms [Cooper et al. 1999; Kulkarni et al. 2003] have been used during iterative searches to find optimization phase sequences better than the default one used in their compilers. Most of the results report good performance improvements over their fixed compiler sequence.

In order to tackle the huge optimization phase order search spaces it is important to find ways to drastically prune these search spaces. A method called Optimization-Space Exploration [Triantafyllis et al. 2003], uses static performance estimators to reduce the search time. In order to prune the search space they limit the number of configurations of optimization-parameter value pairs to those that are likely to contribute to performance

improvements. In other attempts to reduce the cost of iterative optimizations, researchers have used predictive modeling and code context information to focus search on the most fruitful areas of the phase order space for the program being compiled for static compilers [Agakov et al. 2006] as well as for dynamic compilers [Cavazos and O'Boyle 2006]. In our past research, we used genetic algorithms with aggressive pruning of the search space [Kulkarni et al. 2004; Kulkarni et al. 2005] to make searches for effective optimization phase sequences faster and more efficient. During this work we realized that a significant portion of typical phase order search spaces is redundant because many different orderings of optimization phases produce the same code. This observation was the major motivation for this research.

Studies of using static performance estimations to avoid program executions have also been done previously [Knijnenburg et al. 2000; Wagner et al. 1994; Cooper et al. 2005]. Wagner et al. presented a number of static performance estimation techniques to determine the relative execution frequency of program regions, and measured their accuracy by comparing them to profiling [Wagner et al. 1994]. They found that in most cases static estimators provided sufficient accuracy for their tasks. Knijnenburg et al. [2000] used static models to reduce the number of program executions needed by iterative compilation. Our approach of static performance estimation is most similar to the approach of *virtual execution* used by Cooper et al. [Cooper et al. 2005] in their ACME system of compilation. In the ACME system, Cooper et al. strived to execute the application only once (for the un-optimized code) and then based on the execution counts of the basic blocks in that function instance, and careful analysis of transformations applied by their compiler, determine the dynamic instruction counts for other events, such as function instances. With this approach, ACME has to maintain detailed state, which introduces some amount of additional complexity in the compiler. In spite of detailed analysis, in a few cases ACME is not able to accurately determine the dynamic instruction count due to the types of optimizations been applied, occasionally resulting in small errors in their computation.

3. EXPERIMENTAL FRAMEWORK

The research in this paper uses the Very Portable Optimizer (VPO) [Benitez and Davidson 1988], which was a part of the DARPA and NSF co-sponsored National Compiler Infrastructure project. VPO is a compiler back end that performs all its optimizations on a single low-level intermediate representation called RTLs (Register Transfer Lists). Since VPO uses a single representation, it can apply most analysis and optimization phases repeatedly and in an arbitrary order. VPO compiles and optimizes one function at a time. This is important for the current study since restricting the phase ordering problem to a single function, instead of the entire file, helps to make the optimization phase order space more manageable. 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.

Even though native execution of the benchmarks on the ARM system to measure dynamic runtime performance would be ideal, we were not able to do so due to resource constraints. Mainly, we did not have access to an ARM machine that runs Linux, and which also supports our compilation framework. Secondly, ARM machines are considerably slower than state-of-the-art x86 machines, so performing hundreds of long-running experiments will require a significant number of custom ARM machines, which was infea-

sible for us to arrange. Therefore, we used the SimpleScalar set of functional and cycle-accurate simulators [Burger and Austin 1997] for the ARM to get dynamic performance measures.

Table I describes each of the 15 *optional* code-improving phases that we used during our exhaustive exploration of the optimization phase order search space. In addition, VPO also employs two compulsory phases, *register assignment* and *fix entry-exit*, that must be performed. *Register assignment* assigns pseudo registers to hardware registers. [1] In our experiments VPO implicitly performs register assignment before the first code-improving phase in a sequence that requires it. *Fix entry-exit* calculates required stack space, local/argument offsets, and generates instructions to manage the activation record of the runtime stack. The compiler applies *fix entry-exit* after performing the last optional code-improving phase in a sequence.

Two other optimizations, *merge basic blocks* and *eliminate empty blocks*, were removed from the optional optimization list used for the exhaustive search since these optimizations only change the internal control-flow representation as seen by the compiler, do not touch any instructions, and, thus, do not directly affect the final generated code. These optimizations are now implicitly performed after any transformation that has the potential of enabling them. Finally, after applying *fix entry-exit*, the compiler also performs predication and instruction scheduling before the final assembly code is produced. These last two optimizations should be performed late in VPO's compilation process, and so are not included in the set of phases used for exhaustive optimization space enumeration.

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*. *Evaluation order determination* 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*. Many of these phases depend on the detection of basic induction variables and VPO requires these to be in registers before they are detected. These phases can be performed in any order after *register allocation* is applied. *Register allocation* itself can only be effective after *instruction selection* so that candidate load and store instructions can contain the addresses of arguments or local scalars. Finally, there are a set of phases that require the allocation of registers and must be performed after *register assignment*.

VPO is a compiler back end. 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 VPO. We also do not perform ILP (frequent path) optimizations since the ARM architecture, our target for this study, is typically implemented as 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 this study. In this study we are investigating only 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 apply it with a loop unroll factor of two since we are generating code for an embedded processor

[1] In VPO, pseudo registers only represent temporary values and not variables. Before register allocation, all program variables are assigned space on the stack.

Optimization Phase	Gene	Description
branch chaining	b	Replaces a branch/jump target with the target of the last jump in the chain.
common subexpression elimination	c	Performs global analysis to eliminate fully redundant calculations, which also includes global constant and copy propagation.
unreachable code elimination	d	Removes basic blocks that cannot be reached from the function entry block.
loop unrolling	g	To potentially reduce the number of comparisons and branches at run time and to aid scheduling at the cost of code size increase.
dead assignment elimination	h	Uses global analysis to remove assignments when the assigned value is never used.
block reordering	i	Removes a jump by reordering blocks when the target of the jump has only a single predecessor.
loop jump minimization	j	Removes a jump associated with a loop by duplicating a portion of the loop.
register allocation	k	Uses graph coloring to replace references to a variable within a live range with a register.
loop transformations	l	Performs loop-invariant code motion, recurrence elimination, loop strength reduction, and induction variable elimination on each loop ordered by loop nesting level.
code abstraction	n	Performs cross-jumping and code-hoisting to move identical instructions from basic blocks to their common predecessor or successor.
evaluation order determination	o	Reorders instructions within a single basic block in an attempt to use fewer registers.
strength reduction	q	Replaces an expensive instruction with one or more cheaper ones. For this version of the compiler, this means changing a multiply by a constant into a series of shift, adds, and subtracts.
branch reversal	r	Removes an unconditional jump by reversing a conditional branch when it branches over the jump.
instruction selection	s	Combines pairs or triples of instructions that are are linked by set/use dependencies. Also performs constant folding.
useless jump removal	u	Removes jumps and branches whose target is the following positional block.

Table I. Candidate Optimization Phases Along with their Designations

where code size can be a significant issue.

It is important to realize that all optimization phases in VPO, except *loop unrolling* can be *successfully* applied only a limited number of times. Successful application of each phase depends on the presence of both the program inefficiency targeted by that phase, as well as the presence of architectural features required by the phase. Thus, (1) *register allocation* is limited (in the number of times it can be successfully applied) by the number of live ranges in each function. (2) *Loop invariant code motion* is limited by the number of instructions within loops. (3) *Loop strength reduction* converts regular induction variables to basic induction variables, and there are a limited number of regular induction variables. (4) There are a set of phases that eliminate jumps (*branch chaining, block reordering, loop jump minimization, branch reversal, useless jump removal*), and these are limited by the number of jumps in each function. (5) *Common subexpression elimination* is limited by the number of calculations in a function. (6) *Dead assignment elimination* is limited by the number of assignments. (7) *Instruction selection* combines instructions together and is limited by the number of instructions in a function. (8) *Induction variable elimination* is limited by the number of induction variables in a function. (9) *Recurrence elimination* removes unnecessary loads across loop iterations and is limited by the number of loads in

a function. (10) *Code abstraction* is limited by the number of instructions in a function.

Loop unrolling is an optimization that can be attempted an arbitrary number of times, and can produce a new function instance every time. We restrict loop unrolling to be attempted only once for each loop. This is similar to the restriction placed on loop unrolling in most compilers. Additionally, optimizations in VPO never undo the changes made by another phase. Even if they did, our approach could handle this since the function instance graph (explained in Section 4) would no longer be a DAG, and would contain cycles. Thus, for any function, the number of distinct function instances that can be produced by any possible phase ordering of any (unbounded) length is finite, and exhaustive search to enumerate all function instances should terminate in every case.

Note that some phases in VPO represent multiple optimizations in many compilers. However, there exist 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. For example, it has been reported that only 15 out of 60 possible optimization phases in GCC were included in an earlier work determining Pareto optimization levels in GCC [Hoste and khout 2008]. 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 applied in our compiler has never before been accomplished to the best of our knowledge.

For these experiments we used a subset of the benchmarks from the *MiBench* benchmark suite, which are C applications targeting specific areas of the embedded market [Guthaus et al. 2001]. We selected two benchmarks from each of the six categories of applications in MiBench. Table II contains descriptions of these programs. The first two columns in Figure II show the benchmarks we selected from each application category in MiBench. The next column displays the number of lines of C source code per program, and the last column in Figure II provides a short description of each selected benchmark. VPO compiles and optimizes individual functions at a time. The 12 benchmarks selected contained a total of 244 functions, out of which 88 were executed with the input data provided with each benchmark.

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

Table II. MiBench Benchmarks Used in the Experiments

4. APPROACH FOR EXHAUSTIVE EVALUATION OF THE PHASE ORDER SPACE

Complete phase order space evaluation to find the optimal phase ordering for each function would require enumeration and performance measurement of the application code generated by the compiler after applying each possible combination of optimization phases. Both of these tasks, *enumeration* and *performance evaluation*, have been generally considered infeasible over the complete phase order space. In the following sections we will explain our approach which makes these tasks possible in a reasonable amount of time for most of the functions that we studied.

4.1 Exploiting Redundancy in the Phase Order Space

In order to achieve high performance current compilers typically employ several different optimization phases, with few restrictions on the order of applying these phases. Furthermore, interactions between optimization phases cause some phases to be successful multiple times in the same optimization sequence. The conventional approach for exhaustive phase order enumeration attempts to fix the optimization sequence length, and then compile the program with all possible combinations of optimization phases of the selected length. The complexity of this approach is exponential and clearly intractable for any reasonable sequence length. It is also important to note that any such attempt to enumerate all combinations of optimizations is, in principle, limited by our lack of a priori knowledge of the best sequence length for each function.

Interestingly, another method 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. This approach to the same problem clearly makes the solution much more practical because there are far fewer distinct function instances than there are optimization phase orderings, since different orderings can generate the same code. Thus, the challenge now is to find accurate and efficient methods to detect identical function instances produced by distinct phase orderings.

Figure 1 illustrates the phase order space for four distinct optimization phases. At the root (level 0) we start with 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 made tractable without losing any information by using three pruning techniques which we describe in the next three sections.

4.1.1 *Detecting Dormant Phases.* The first pruning technique exploits the property that not all optimization phases are successful at all levels and in all positions. We call applied phases *active* when they produce changes to the program representation. A phase is said to be *dormant* if it could not find any opportunities to be successful when applied. Detecting dormant phases eliminates 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.

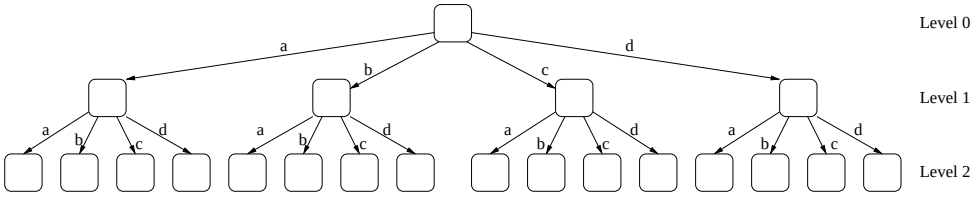


Fig. 1. Naive Optimization Phase Order Space for Four Distinct Optimizations

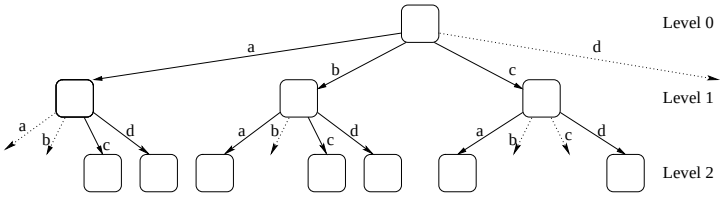


Fig. 2. Effect of Detecting Dormant Phases on the Search Space in Figure 1

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

Fig. 3. Diff. Opts. Having the Same Effect

4.1.2 *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 performed 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. One example is illustrated in Figure 3. *Instruction selection* merges the effects of 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.* Thus, if the current function instance is detected to be identical to some earlier function instance, then it can be safely eliminated from the space of distinct function instances.

4.1.3 *Detecting Equivalent Function Instances.* 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 [Kulkarni et al. 2004; Kulkarni et al. 2005]. 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 4(a). Figures 4(b) and 4(c) show two possible translations given two different orderings of optimization phases that consume registers and modify the control flow. To detect this situation we perform a remapping of hardware registers for each function instance, and again compare the current instance with all previous instances for a match. Thus, after remapping, code in Figures 4(b) and 4(c) are both transformed to the code in Figure 4(d). On most machines, which have identical access time for all registers, these two code sequences would have identical performance, and hence the function instances are termed *equivalent*. We only need to maintain one instance from each group of equivalent function instances, and the rest can be eliminated from the search space.

The effect of different optimization sequences producing identical or equivalent 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 5. By comparing Figures 1, 2 and 5, it is apparent how these three 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.

```
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

Fig. 4. Different Functions with Equivalent Code

4.2 Performance Estimation of Each Distinct Function Instance

Finding the dynamic performance of a function instance requires execution or simulation of the application. Executing the program typically takes considerably longer than it takes the compiler to generate each function instance. Moreover, simulation can be orders of magnitude more expensive than native execution, and is often the only resort for evaluating the performance of applications on embedded processors. Thus, in most cases, the enumerated distinct function instances for each function are still substantial enough to make executing

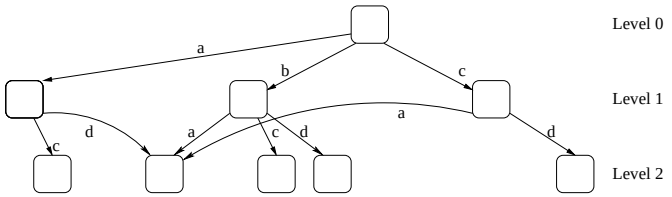


Fig. 5. Detecting Identical Code Transforms the Tree in Figure 2 to a DAG

the program for each function instance prohibitively expensive. Thus, in order to find the optimal function instance it is necessary to reduce the number of program executions, but still be able to accurately estimate dynamic performance for all function instances. In this section, we describe our approach to obtain accurate performance estimates.

In order to reduce the number of executions we use a technique that is based on the premise that two different function instances with identical control-flow graphs will execute the same basic blocks the same number of times. Our technique is related to the method used by Cooper et al. in their ACME system of adaptive compilation [Cooper et al. 2005]. However, our adaptations have made the method simpler to implement and more accurate for our tasks. During the exhaustive enumerations we observed that for any function the compiler only generates a very small number of distinct control-flow paths, i.e., multiple distinct function instances have the same basic block control-flow structure. For each such set of function instances having the same control flow, we execute/simulate the application only once to determine the basic block execution counts for that control-flow structure. For each distinct function instance we then calculate the number of cycles required to execute each basic block. The dynamic performance of each function instance can then be calculated as the sum of the products of basic block cycles times the block execution frequency over all basic blocks. We call this performance estimate our *dynamic frequency measure*. For the current study, the basic block cycle count is a static count that takes into account stalls due to pipeline data hazards and resource conflicts, but does not consider order dependent events, such as branch misprediction and memory hierarchy effects. Additionally, we reset all computation resources to be idle at the start of the static cycles calculation for each basic block. Other more advanced measures of static performance, using detailed cache and resource models, can be considered at the cost of increased estimation time. As we will show later in this paper, we found our simple estimation method to be sufficiently accurate for our needs on the in-order ARM processor.

5. IMPLEMENTATION DETAILS

In this section, we describe some of our implementation details for the techniques described in the previous section to find the optimal function instance by our measure of dynamic performance, and with respect to the possible phase orderings in our compiler.

The phase order space can be generated/traversed in either a breadth-first or a depth-first order. Each traversal algorithm has different advantages and drawbacks for our experiments. In both cases we start with the unoptimized function instance representing the root node of the DAG. Using breadth first search, nodes in Figure 5 would be generated in the order shown in Figure 6(a), while depth-first search would generate the nodes in the order shown in Figure 6(b).

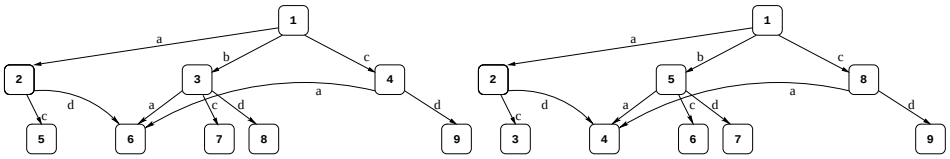


Fig. 6. Breadth-First and Depth-First DAG Traversal algorithms

During the search process we have to compile the same function with thousands of different optimization phase sequences. Generating the function instance for every new optimization sequence involves discarding the previous compiler state (which was produced by the preceding sequence), reading the unoptimized function back from disk, and then applying all the optimizations in the current sequence. We made a simple enhancement to keep a copy of the unoptimized function instance in memory to avoid disk accesses for all optimization sequence evaluations, except the first. Another enhancement we implemented to make the searches faster uses the observation that many different optimization sequences share common prefixes. Thus, for the evaluation of each phase sequence, instead of rolling back to the unoptimized function instance every time, we determine the common prefix between the current sequence and the previous sequence, and only roll back until after the last phase in the common prefix. This technique saves the time that would otherwise have been required to re-perform the analysis and optimizations in the common prefix. The number of intermediate function instances we need to store is limited by the depth of the DAG, so this technique does not cause any significant space overhead in practice, but proves to be very time efficient.

Table III shows the successful sequences during the generation of the DAG in Figure 5 during both breadth-first and depth-first traversals. Since phases in the common prefix do not need to be reapplied, the highlighted phases in Table III are the only ones which are actually attempted. Depth-first search keeps a stack of previous phase orderings, and is typically able to exploit greater redundancy among successive optimization sequences than breadth-first search. Therefore, to reduce the search time during our experiments we used the depth-first approach to enumerate the optimization phase order search space.

	Breadth-First Traversal			Depth-First Traversal		
1.	**a**		1.	**a**	
2.	**b**		2.	a	**c**
3.	**c**		3.	a	**d**
4.	**a**	**c**	4.	**b**	
5.	a	**d**	5.	b	**a**
6.	**b**	**a**	6.	b	**c**
7.	b	**c**	7.	b	**d**
8.	b	**d**	8.	**c**	
9.	**c**	**a**	9.	c	**a**
10.	c	**d**	10.	c	**d**

Table III. Applied Phases during Space Traversal

There is, however, a drawback to the depth-first traversal approach. Although a large majority of the functions that we encountered can be exhaustively evaluated in a reasonable amount of time, there are a few functions, with extremely large search spaces, which are intractable even after applying all our pruning methods. It would be beneficial if we could identify such cases before starting the search or early on in the search process so that we do not spend time and resources unnecessarily. This identification is easier to accomplish during the breadth-first traversal as we can see the growth in the search space at each level (refer Figure 5). If the growth in the first few levels is highly exponential, and difficult to tame, then we can stop the search on that function and designate that function as too large to exhaustively evaluate. In an earlier study, we used breadth-first search and stopped the search whenever the number of sequences to evaluate at any level grew to more than a million [Kulkarni et al. 2006a]. It is hard to find such a cut-off point during a depth-first traversal. For this study, we stop the exhaustive search on any function if the time required exceeded an approximate limit of 2 weeks. Please note that exhaustive phase order evaluation for most of the functions requires a few minutes or a few hours, with only the largest enumerated functions requiring a few days.

Figure 7 illustrates the steps followed during the exhaustive phase order evaluation for each function. We will now briefly describe the implementation details for each step.

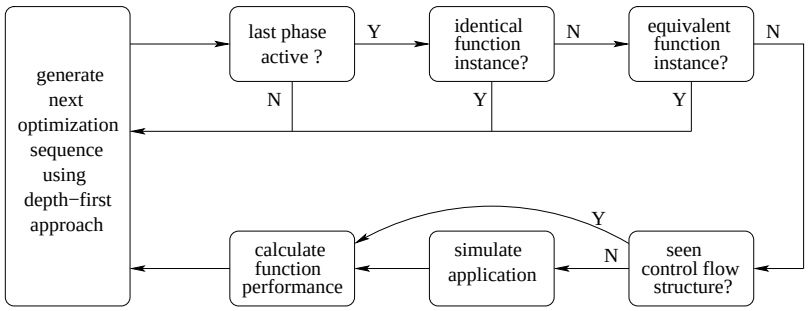


Fig. 7. Steps followed during an exhaustive evaluation of the phase order space for each function

5.1 Detecting Dormant Phases

There is no phase in our compiler that can be successfully applied more than once consecutively. Therefore, an active phase at one level is not even attempted at the next level. For all other attempted phases, we get feedback from the compiler reporting if the phase was active or dormant. Since dormant phases keep the function unchanged, we do not need to further generate that branch of the search space.

5.2 Detecting Identical Function Instances

A naive comparison of each newly generated function instance with all previous function instances will be slow and require prohibitive memory space. Therefore, to make the comparisons efficient, we calculate multiple hash values for each function instance and compare the hash values for a match. 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 was

also used in our previous studies to detect redundant sequences when applying a genetic algorithm to search for effective phase sequences [Kulkarni et al. 2004; Kulkarni et al. 2005]. 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 [Peterson and Brown 1961]. CRCs are useful in our case since function instances that are identical except for different order of instructions will be detected to be distinct. 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.

5.3 Detecting Equivalent Function Instances

To detect equivalent function instances we map each register and block label-number to a different number depending on when it is encountered in the control flow. Note that this mapping is only performed for the checksum calculation and is not used when additional phases are applied. We start scanning the function from the top basic block. Each time a register is first encountered we map it to a distinct number starting from 1. This register would keep the same mapping throughout the function. For instance, if register `r[10]` is mapped to `r[1]`, then each time `r[10]` is encountered it would be changed to `r[1]`. If `r[1]` is later found in some RTL, then it would be mapped to the remap number existing at that position during the scan. Note that this is different from register remapping of live ranges [Kulkarni et al. 2004; Kulkarni et al. 2005], 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. The algorithm for detecting equivalent function instances then proceeds similarly to the earlier approach of detecting identical function instances.

5.4 Obtaining Dynamic Performance Measures

As explained in Section 4.2, to determine dynamic performance we only need to simulate the application for new control flows. Thus, after generating each new function instance we compare its control-flow structure with all previously encountered control flows. This check compares the number of basic blocks, the position of the blocks in the control-flow graph, the positions of the predecessors and successors of each block, and the relational operator and arguments of each conditional branch instruction. *Loop unrolling* presents a complication when dealing with loops of single basic blocks. It is possible for loop unrolling to unroll such a loop and change the loop exit condition. Later if some optimization coalesces the unrolled blocks, then the control flow looks identical to that before unrolling, but due to different loop exit conditions, the block frequencies are actually different. We handle such cases by verifying the loop exit conditions and marking unrolled blocks differently from non-unrolled code. We are unaware of any other control flow changes caused by our set of optimization phases that would be incorrectly detected by our algorithm.

If the check reveals that the control flow of the new function instance has not as yet been encountered, then before producing assembly code, the compiler instruments the function with additional instructions using EASE [Davidson and Whalley 1991]. Upon simulation, these added instructions count the number of times each basic block is executed. The

functional simulator *sim-uop*, present in the SimpleScalar simulator toolset [Burger and Austin 1997] is used for the simulations. The dynamic performance for each function instance is estimated by multiplying the number of static cycles calculated for each block with the corresponding block execution counts.

6. EXPERIMENTAL RESULTS

In this study we have been able to exhaustively evaluate the phase order space for 234 out of a total of 244 functions over 12 applications selected from the MiBench benchmark suite [Guthaus et al. 2001]. Only 88 out of the 244 total functions were executed when using the input data sets provided with the MiBench benchmarks. Out of the 88 executed functions, we were able to evaluate 79 functions exhaustively. For space reasons, we only present the results for the executed functions in this submission version of the paper. Table IV presents the results for the executed enumerated functions. The functions in the table are sorted in descending order by the number of instructions in the un-optimized function instance. The table presents the results only for the top 50 functions in each category, along with the average numbers for the remaining functions.

The first three columns in Table IV, namely the number of instructions (*Inst*), branches (*Br*), and loops (*Lp*) in the unoptimized function instance, present some of the static characteristics for each function. These numbers provide a measure of the complexity of each function. As expected, more complex functions tend to have larger search spaces. The next two columns, number of distinct function instances (*Fn_inst*) and the maximum active sequence length (*Len*), reveal the sizes of the *actual* and *attempted* phase order spaces for each function. A maximum optimization sequence length of n gives us an attempted search space of 15^n, where 15 is the number of optimizations present in VPO. The numbers indicate that the median function has an optimization phase order search space of 15^{16}. Moreover, the search space can grow to 15^{44} (for the function *pfx_list_chk* in *ispell*) in the worst case for the compiler and benchmarks used in this study. Thus, we can see that although the attempted search space is extremely large, the number of distinct function instances is only a tiny fraction of this number. A more important observation is that unlike the attempted space, the number of distinct function instances does not typically increase exponentially as the sequence length increases. This is precisely the redundancy that we are able to exploit in order to make our approach of exhaustive phase order space enumeration feasible.

The number of distinct control flows, presented in the column labeled *CF*, is more significant for the performance evaluation of the executed functions in our benchmarks. We only need to simulate the application once for each distinct control flow. The relatively small number of distinct control flows as compared to the total number of unique function instances makes it possible to obtain the dynamic performance of the entire search space with only a relatively insignificant compile time overhead. The column labeled *CT* in Table IV gives an estimate of the compile time required for the exhaustive evaluation of each function on a 64-bit Intel Pentium-D 3.0Ghz processor. Our exhaustive experiments were conducted on five different machines of varying strengths, and so we only provide an estimate of the compile time on the granularity of several minutes (M), hours (H), days (D), or weeks (W). As expected, the compile time is directly proportional to the number of instructions, complexity of the control-flow, and the phase order space size of each function. Additionally, for the executed functions, the compile time is also dependent on the number

Function	Inst	Br	Lp	Fn_inst	Len	CF	CT	Leaf	within ? % of opt.			% perf dif.	
									opt	2%	5%	Batch	Worst
main(t)	1275	110	6	2882021	29	389	W	15164	1.1	26.3	41.1	0.0	84.3
parse_sw...(j)	1228	144	1	180762	20	53	D	2057	0.4	1.9	4.1	6.7	64.8
askmode(i)	942	84	3	232453	24	108	D	475	1.7	2.9	4.6	8.4	56.2
skiptoword(i)	901	144	3	439994	22	103	D	2834	0.5	5.6	29.6	6.1	49.6
start_in...(j)	795	50	1	8521	16	45	H	80	20.0	60.0	60.0	1.7	28.4
treeinit(i)	666	59	0	8940	15	22	H	240	3.3	40.0	100.0	0.0	3.4
pfx_list...(i)	640	59	2	1269638	44	136	D	4660	0.3	0.3	2.1	4.3	78.6
main(f)	624	35	5	2789903	33	122	W	4214	0.0	0.0	0.0	7.5	46.1
sha_tran...(h)	541	25	6	548812	32	98	H	5262	0.0	9.4	30.2	9.6	133.4
initckch(i)	536	48	2	1075278	32	32	D	4988	24.1	91.1	91.1	0.0	108.4
main(p)	483	26	1	14510	15	10	H	178	1.7	21.9	32.0	7.7	13.1
pat_insert(p)	469	41	4	1088108	25	71	D	3021	1.4	46.6	47.3	0.0	126.4
main(j)	465	28	1	25495	21	12	H	134	0.0	0.0	0.0	5.6	6.0
main(l)	464	51	4	1896446	25	920	W	5364	0.0	25.0	25.0	0.9	89.3
adpcm_co...(a)	385	35	1	28013	23	24	H	230	1.3	2.6	12.6	1.8	48.9
dijkstra(d)	354	22	3	92973	22	18	H	1356	0.3	22.1	26.8	0.0	51.1
good(i)	313	29	1	87206	22	32	H	370	4.3	17.3	48.9	0.0	14.6
chk_aff(i)	304	30	1	179431	21	160	H	2434	1.6	10.8	39.8	0.1	58.7
cpTag(t)	303	40	0	522	11	9	M	16	0.0	100.0	100.0	1.6	1.6
makeposs...(i)	280	33	1	70368	24	119	H	498	2.4	30.5	33.7	0.0	130.1
xgets(i)	273	37	1	37960	19	103	H	284	4.2	32.4	32.4	0.0	129.7
missings...(i)	262	28	2	23477	26	30	H	513	8.2	8.2	14.4	4.0	86.8
missingl...(i)	252	31	3	11524	16	40	H	180	0.6	0.6	3.3	12.9	79.2
chk_suf(i)	243	21	1	75628	21	29	H	2835	0.8	4.4	11.7	0.8	62.4
compound...(i)	222	30	1	78429	20	49	H	448	3.6	3.6	3.6	11.1	100.0
main(b)	220	15	2	182246	23	84	H	508	0.8	16.9	16.9	8.3	250.0
skipover...(i)	212	30	1	105353	29	110	H	413	0.5	7.3	47.0	7.7	75.4
lookup(i)	195	22	2	37396	20	38	H	114	0.0	0.0	0.0	7.7	75.9
wronglet...(i)	194	25	2	22065	17	25	H	430	0.5	0.5	4.2	15.0	89.8
ichartostr(i)	186	26	3	40524	21	40	H	304	1.6	27.3	52.6	0.0	236.0
main(s)	175	12	3	30980	23	10	M	163	4.9	7.4	8.6	0.0	67.4
main(d)	175	15	3	9206	20	22	M	85	2.4	3.5	49.4	4.3	75.3
main(q)	174	14	2	38759	23	121	H	160	2.5	25.0	25.0	0.0	214.3
treelookup(i)	167	23	2	67507	17	65	H	1992	15.3	15.3	15.3	0.0	66.7
insertR(p)	161	15	0	2462	14	6	M	22	18.2	36.4	72.7	0.5	97.9
sha_final(h)	155	4	0	2472	13	3	M	68	20.6	20.6	41.2	0.0	21.7
select_f...(j)	149	21	0	510	10	10	M	16	25.0	25.0	75.0	0.0	7.1
byte_rev...(h)	146	5	1	2715	19	13	M	54	7.4	61.1	74.1	0.4	42.4
main(a)	140	10	1	1676	16	8	M	12	0.0	66.7	66.7	0.0	16.9
strtoichar(i)	140	18	1	10721	19	17	H	109	7.3	11.0	26.6	0.0	100.5
ntbl_bit...(b)	138	1	0	48	7	1	M	8	25.0	25.0	75.0	0.0	10.7
read_pbm...(j)	134	21	2	4182	15	18	H	60	6.7	16.7	20.0	6.7	69.3
bitcount(b)	133	1	0	44	8	1	M	7	14.3	14.3	28.6	0.0	64.3
strsearch(s)	128	17	2	32550	17	48	H	972	0.3	0.9	5.2	1.5	135.2
enqueue(d)	124	10	1	488	13	4	M	12	16.7	75.0	100.0	0.2	4.5
sha_update(h)	118	7	1	5990	18	50	H	49	0.0	81.6	81.6	0.1	82.0
transpos...(i)	117	10	1	5310	16	19	H	44	4.5	13.6	25.0	4.5	98.7
pat_search(p)	110	14	1	5052	15	33	H	98	4.1	16.3	16.3	0.6	66.8
init_sea...(s)	103	9	2	1430	15	11	M	30	3.3	53.3	53.3	0.4	459.3
main(h)	101	11	1	22476	20	129	H	320	0.0	0.0	0.0	7.1	100.0
remaining(29)	51.5	4.1	0.3	442.6	9.1	4.4	13.0	-	41.8	47.8	52.8	7.8	34.0
average(79)	234.3	21.7	1.2	174574.8	16.1	47.4	813.4	-	18.7	32.6	41.8	4.8	65.4

(Function - function name followed by benchmark indicator [(a)-adpcm, (b)-bitcount, (d)-dijkstra, (f)-fft, (h)-sha, (i)-ispell, (j)-jpeg, (l)-blowfish, (q)-qsort, (p)-patricia, (t)-tiff, (s)-stringsearch]), (Inst - number of instructions in unoptimized function), (Br - number of conditional and unconditional transfers of control), (Lp - number of loops), (Fn_inst - number of distinct control-flow instances), (Len - largest active optimization phase sequence length), (CF - number of distinct control flows), (CT - compile time [M-minutes, H-hours, D-days, W-weeks]), (Leaf - Number of leaf function instances), (within ? % of optimal - what percentage of leaf function instances are within "?"% from optimal), (% perf dif. - % performance difference between *Batch* and *Worst* leaf from Optimal).

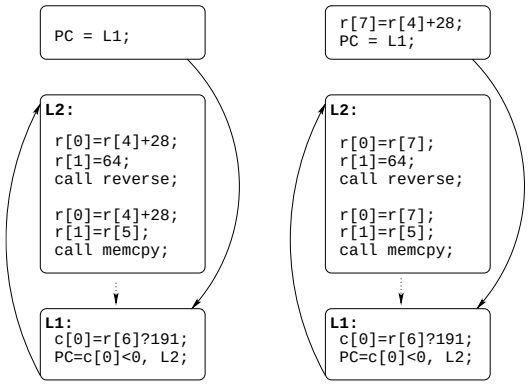
Table IV. Optimization Phase Order Evaluation Results for the Executed Functions in the MiBench Benchmarks

of distinct control-flow paths of the function, since these determine the number of application simulations required to evaluate the performance of all function instances. Please note that the compile time for the undisplayed functions varied from several seconds in most cases to a few minutes.

The next column, labeled *Leaf*, gives a count of the *leaf* function instances. These are function instances for which no additional phase is able to make any further changes to the program representation. The small number of leaf function instances imply that even though the enumeration DAG may grow out to be very wide, it generally starts converging towards the end. Leaf instances are an interesting and important sub-class of possible function instances. They are the only instances that can be generated by an aggressive compiler like VPO that repetitively apply optimizations until there are no more that can be applied. At the same time, leaf instances generally produce good performance. It is easy to imagine why this is the case, since all optimizations are designed to improve performance, and leaf instances apply all that are possible in some particular ordering. For most of our benchmark functions, at least one leaf function instance is able to reach optimal. Note again that we are defining optimal in this context to be the function instance that results in the best dynamic frequency measures. The set of the next three columns in Table IV, labeled *within ?% of opt.*, reveal that over 18% of all leaf instances, on average, reached optimal, with over 40% of them within 5% of optimal. Additionally, we calculated that for 86% of the functions in our test suite at least one leaf function instance reached optimal.

We analyzed the few cases for which none of the leaf function instances achieved optimal performance. The most frequent reason we observed that caused such behavior is illustrated in Figure 8. Figure 8(a) shows a code snippet which yields the best performance and 8(b) shows the same part of the code after applying *loop-invariant code motion*. $r[0]$ and $r[1]$ are passed as arguments to both of the called functions. Thus, it can be seen from Figure 8(b) that *loop-invariant code motion* moves the invariant calculation, *r[4]+28*, out of the loop, replacing it with a register to register move, as it is designed to do. But later, the compiler is not able to collapse the reference by *copy propagation* because it is passed as an argument to a function. The implementation of *loop-invariant code motion* in VPO is not robust enough to detect that the code will not be further improved. In most cases, this situation will not have a big impact, unless this loop does not typically execute many iterations. It is also possible that *loop-invariant code motion* may move an invariant calculation out of a loop that is never entered during execution. In such cases, no leaf function instance is able to achieve the best dynamic performance results.

The last two columns in Table IV compare the performance of the conventional (batch) compiler and the worst performing leaf function instance with the function instance(s) having the optimal ordering. The conventional VPO compiler iteratively applies optimization phases until there are no additional changes made to the program by any phase. As a result, the fixed (batch) sequence in VPO always results in a leaf function instance. In contrast, many other compilers (including GCC) cannot do this effectively since it is difficult to re-order optimization phases in these compilers. The fixed batch optimization sequence in VPO has been tuned over several years. The maturity of VPO is responsible for the batch sequence finding the optimal function instance for 33 of the 79 executed functions. However, in spite of this aggressive baseline, the batch compiler produces code that is 4.8% worse than optimal, on average. The worst performing leaf function instance can potentially be over 65% worse than optimal on average.



(a) Before Code Motion (b) After Code Motion

Fig. 8. Case When No Leaf Function Instance Yields Optimal Performance

Function	Insts	Blk	Trans. of Cntr.		Loops		
			Cond.	Uncond.	Depth1	Depth2	Depth3
main(i)	3335	369	93	142	3	1	0
linit(i)	1842	180	51	98	5	3	0
checkline(i)	1387	203	69	96	5	2	0
save_root..(i)	1140	133	38	73	4	4	0
suf_list..(i)	823	102	33	48	1	1	0
fft_float(d)	680	45	11	21	3	1	1
treeoutput(i)	767	114	29	60	7	3	3
flagpr(i)	581	86	21	46	8	0	0
cap_ok(i)	521	95	24	54	1	5	0
prr_pre..(i)	470	65	17	33	3	0	0

(Function - function name followed by benchmark indicator [(d)-dijkstra, (i)-ispell]), (Inst - number of instructions in unoptimized function), (Blk - number of basic blocks in unoptimized function), (Trans. of Cntr. - number of conditional and unconditional transfers of control), (Loops - number of loops at nesting depths 1, 2, and 3),

Table V. Static Features of Functions which We Could Not Exhaustively Evaluate

Table V displays the static features of the functions that we were unable to exhaustively enumerate according to our stopping criteria. As explained earlier, we terminate the exhaustive evaluation for any function when the time required for the algorithm exceeds two weeks. Out of the 244 possible functions, we were unable to exhaustively evaluate only 10 of those functions. It is difficult to identify one property of each function that leads to such uncontrollable growth in the phase order space. Instead, we believe that some combination of the high number of loops, greater loop nesting depths, branches, number of instructions, as well as the instruction mix are responsible for the greater phase order space size.

7. CORRELATION BETWEEN DYNAMIC FREQUENCY MEASURES AND PROCESSOR CYCLES

Even after pruning the optimization phase order search space, evaluating the performance of all distinct function instances using *simulation* is prohibitively expensive. Therefore, we used a measure of estimated performance based partly on static function properties. Our performance estimate accounts for stalls resulting from pipeline data hazards, but does not consider other penalties encountered during execution, such as branch misprediction and cache miss penalties. In spite of the potential loss in accuracy, we expect our measure of performance to be sufficient for achieving our goal of finding the optimal phase ordering with only a handful of program simulations. Our expectation is especially true for embedded applications. Unlike general-purpose processors, the cycles obtained from a simulator can often be very close to executed cycles on an embedded processor since these processors may have simpler hardware and no operating system. For similar reasons, dynamic frequency measures on embedded processors will also have a much closer correlation to simulated cycles, than for general-purpose processors. Indeed, in the next section, we perform some studies to show that there is a strong correlation between dynamic frequencies and simulator cycles. Even though our measure of dynamic frequencies do not exactly match the simulator cycles we believe that this issue is less critical. An exact match is less important since we are mainly interested in sorting the function instances according to their performance, and in particular in determining the best function instance. A strong correlation between dynamic frequencies and simulator cycles allows us to achieve that objective.

7.1 Complete Function Correlation

The SimpleScalar simulator toolset [Burger and Austin 1997] includes many different simulators intended for different tasks. For these experiments we used SimpleScalar's cycle-accurate simulator, *sim-outorder*, modified to measure cycles only for the function of interest. This enhancement makes the simulations faster, since most of the application is simulated using the faster *functional* simulation mode. We have verified that our modification did not cause any noticeable performance deviation [Kulkarni et al. 2006b].

Clearly, simulating the application for all function instances in every function is very time-consuming as compared to simulating the program only on encountering new control-flows. Therefore, we have simulated all instances of only a single function completely to provide an illustration of the close correlation between processor cycles and our estimate of dynamic frequency counts. Figure 9 shows this correlation for all the function instances for the *init_search* function in the benchmark *stringsearch*. This function was chosen mainly because it is relatively small, but still has a sufficient number of distinct function instances to provide a good example.

All the performance numbers in Figure 9 are sorted on the basis of dynamic frequency counts. Thus, we can see that our estimate of dynamic frequency counts closely follows the processor cycles most of the time. It is also seen that the correlation gets better as the function is better optimized. We believe that this improvement in correlation may be due to the reduction in memory accesses after optimizations such as *register allocation*, in turn reducing the number of cache misses, the penalty for which we did not consider during our performance estimates. The improved correlation for optimized function instances is important since the best function instance will generally reside in this portion of the

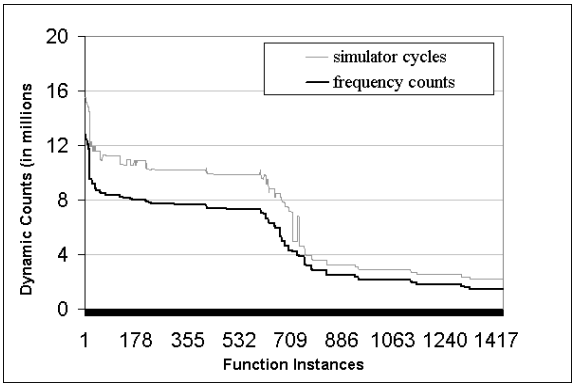


Fig. 9. Correlation between Processor Cycles and Frequency Counts for *init_search*

phase ordering space. The excellent correlation between dynamic frequency estimates and simulator cycles for the optimized function instances allows us to predict the function instances with good/optimal cycle counts with a high level of confidence.

7.2 Correlation for Leaf Function Instances

Figure 10 shows the distribution of the dynamic frequency counts as compared to the optimal counts for all distinct function instances, averaged over all 79 executed functions. From this figure we can see that the performance of the leaf function instances is typically very close to the optimal performance, and that leaf instances comprise a significant portion of optimal function instances as determined by the dynamic frequency counts. From the discussion in Section 6 we know that for more than 86% of the functions in our benchmark suite there was at least one leaf function instance that achieved the optimal dynamic frequency counts.

Since the leaf function instances achieve good performance across all our functions, it is worthwhile to concentrate our correlation study on leaf function instances. These experiments require hundreds of program simulations, which are very time consuming. So, we have restricted this study to only one application from each of the six categories of MiBench benchmarks. For all the executed function from the six selected benchmarks we get simulator cycle counts for only the leaf function instances and compare these values to our dynamic frequency counts. In this section we show the correlation between dynamic frequency counts and simulator cycle counts for only the leaf function instances for all executed functions over six different applications in our benchmark suite.

The correlation between dynamic frequency counts and processor cycles can be illustrated by various techniques. A common method of showing the relationships between variables (data sets) is by calculating Pearson's correlation coefficient for the two variables [Weisstein 2006]. The Pearson's correlation coefficient can be calculated by using the formula:

$$Pcorr = \frac{\sum xy - \frac{\sum x \sum y}{n}}{\sqrt{(\sum x^2 - \frac{(\sum x)^2}{n}) * (\sum y^2 - \frac{(\sum y)^2}{n})}} \quad (1)$$

In Equation 1 x and y correspond to the two variables, which in our case are the dynamic

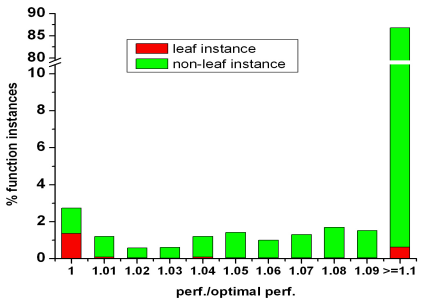


Fig. 10. Average Distribution of Dynamic Frequency Counts

frequency counts and simulator cycles, respectively. Pearson's coefficient measures the strength and direction of a linear relationship between two variables. Positive values of *Pcorr* in Equation 1 indicate a relationship between *x* and *y* such that as values for *x* increase, values of *y* also increase. The closer the value of *Pcorr* is to 1, the stronger is the linear correlation between the two variables.

It is also worthwhile to study how close the processor cycle count for the function instance that achieves the best dynamic measure is to the best overall cycle count over all the leaf function instances. To calculate this measure, we first find the best performing function instance(s) for dynamic frequency counts and obtain the corresponding simulator cycle count for that instance. In cases where multiple function instances provide the same best dynamic frequency count, we obtain the cycle counts for each of these function instances and only keep the best cycle count amongst them. We then obtain the simulator cycle counts for all leaf function instances and find the best cycle count in this set. We then calculate the following ratio for each function:

$$Lcorr = \frac{best\ overall\ cycle\ count}{cycle\ count\ for\ best\ dynamic\ freq\ count} \tag{2}$$

The closer the value of Equation 2 comes to 1, the closer is our estimate of optimal by dynamic frequency counts to the optimal by simulator cycles.

Table VI lists our correlation results for the leaf function instances over all studied functions in our benchmarks. The column, labeled *Pcorr* provides the Pearson's correlation coefficient according to Equation 1. An average correlation coefficient value of 0.96 implies that there is excellent correspondence between dynamic frequency counts and cycles. The next column shows the value of *Lcorr* calculated by Equation 2. The following column gives the number of distinct leaf function instances which have the same best dynamic frequency counts. These two numbers in combination indicate that an average simulator cycle performance of *Lcorr* can be reached by simulating only *nLf* number of the best leaf function instances as determined by our estimate of dynamic frequency measure. Thus, it can be seen that an average performance within 2% of the optimal simulator cycle performance can be reached by simulating, on average, less than 5 good function instances having the best dynamic frequency measure. The next two columns show the same measure of *Lcorr* by Equation 2, but instead of considering only the best leaf instances for dynamic frequency counts, they consider all leaf instances which come within 1% of the

Function	Pcorr	Lcorr 0%		Lcorr 1%		Function	Pcorr	Lcorr 0%		Lcorr 1%	
		Diff	nLf	Diff	nLf			Diff	nLf	Diff	nLf
AR_btbl_b...	1.00	1.00	1	1.00	1	BW_btbl_b...	1.00	1.00	2	1.00	2
bit_count	1.00	1.00	2	1.00	2	bit_shifter	1.00	1.00	2	1.00	2
bitcount	0.89	0.92	1	0.92	1	main	1.00	1.00	6	1.00	23
ntbl_bitc...	0.99	0.95	2	0.95	2	ntbl_bitcnt	1.00	1.00	2	1.00	2
dequeue	0.99	1.00	6	1.00	6	dijkstra	1.00	0.97	4	1.00	269
enqueue	1.00	1.00	2	1.00	4	main	0.98	1.00	4	1.00	4
print_path	1.00	1.00	2	1.00	2	qcount	1.00	1.00	1	1.00	1
CheckPoin...	0.95	1.00	2	1.00	5	IsPowerOf...	0.93	0.98	3	1.00	24
NumberOfB...	0.84	1.00	1	1.00	20	ReverseBits	1.00	1.00	2	1.00	2
byte_reve...	0.89	1.00	1	1.00	3	main	0.71	1.00	25	1.00	74
sha_final	0.72	0.82	26	1.00	50	sha_init	0.98	1.00	4	1.00	9
sha_print	0.95	0.88	1	1.00	6	sha_stream	1.00	1.00	1	1.00	8
sha_trans...	0.97	1.00	2	1.00	35	sha_update	0.98	1.00	14	1.00	32
finish_in...	1.00	1.00	1	1.00	1	get_raw_row	1.00	1.00	7	1.00	7
jinit_rea...	1.00	1.00	2	1.00	2	main	1.00	0.99	2	1.00	153
parse_swi...	0.95	1.00	8	1.00	16	pbm_getc	0.99	1.00	2	1.00	2
read_pbm_...	0.73	0.98	2	0.98	2	select_fi...	0.97	0.90	3	1.00	12
start_inp...	0.95	0.99	12	0.99	15	write_std...	1.00	1.00	1	1.00	1
init_search	1.00	1.00	1	1.00	14	main	1.00	1.00	8	1.00	12
strsearch	1.00	1.00	3	1.00	3						
average	0.96	0.98	4.38	0.996	21						

Pcorr - Pearson's correlation coefficient, Lcorr - ratio of cycles for dynamic frequency to best overall cycles (0% - optimal, 1% - within 1 percent of optimal frequency counts), Diff - ratio for *Lcorr*, nLf - number of leaves achieving the specified dynamic performance

Table VI. Correlation Between Dynamic Frequency Counts and Simulator Cycles for Leaf Function Instances

best dynamic frequency estimate. This allows us to reach within 0.4% of the optimal performance, on average, by performing only 21 program simulations per function. In effect, we can use our dynamic frequency measure to prune most of the instances that are very unlikely to achieve the fewest simulated cycles.

The conclusions of this study are limited since we only considered leaf function instances. It would not be feasible to get cycle counts for all function instances over all functions. In spite of this restriction, the results are interesting and noteworthy since they show that a combination of static and dynamic estimates of performance can predict pure dynamic performance with a high degree of accuracy. This result also leads to the observation that we should typically need to simulate only a very small percentage of the best performing function instances as indicated by dynamic frequency counts to obtain the optimal function instance by simulator cycles. As a final point, it should be noted that although our simple estimation technique is seen to work well on our simple ARM processor, it may not perform as well on more complicated architectures, and in such cases other more detailed estimation techniques should be explored.

8. ANALYSIS OF THE PHASE ORDER SPACE

Exhaustive evaluation of the phase order space for a large number of functions has provided us with a huge data-set which we have analyzed to determine some interesting properties of the optimization phase order space. In this section we will describe some of the characteristics of the optimization phase order space.

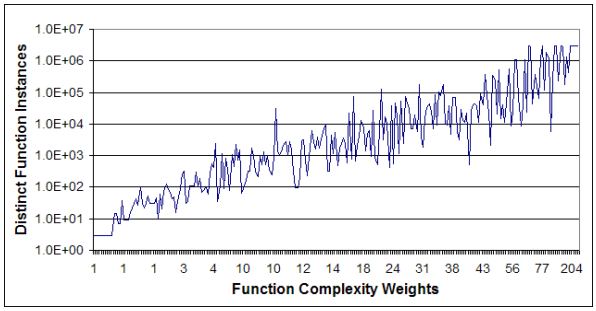


Fig. 11. Function Complexity Distribution

8.1 Statically Predicting Optimization Phase Order Space Size

Out of the 244 functions from the MiBench benchmark suite we were able to completely enumerate the phase order space for 234 of these functions. Although we were unable to exhaustively enumerate the space for the remaining 10 functions, we still had to spend considerable time and resources for their partial search space evaluation before realizing that their function complexity was greater that the capability of our current space pruning techniques to contain the space growth. For such cases, it would be very helpful if we could a priori determine the complexity of the function and estimate the size of the search space statically. This estimation is, however, very hard to achieve. The growth in the phase order space is dependent on various factors such as the number of conditional and unconditional transfers of control, loops, and loop nesting depths, as well as the number of instructions and instruction mix in the function.

We attempted to quantify the function complexity based on static function features such as branches, loops, and number of instructions. All transfers of control are assigned a unit value. Loops at the outermost level are assigned a weight of 5 units. All successive loop nesting levels are weighted two times the weight of the preceding loop level. Functions with the same weight are sorted based on the number of instructions in the unoptimized function instance. The 10 unevaluated functions are assumed to have 3,000,000 distinct function instances, which is more than the number of instances for any function in our set of evaluated functions. Figure 11 shows the distribution of the number of distinct function instances for each function as compared to its assigned complexity weight.

Figure 11 shows a marked increase in the number of distinct function instances with increase in the assigned complexity weights. A significant oscillation of values in the graph reconfirms the fact that it is difficult to accurately predict function phase order space size based on static function features. It is, however, interesting to note that out of the 10 unevaluated functions, five are detected to have the highest complexity, with eight of them within the top 13 complexity values. Thus, a static complexity measure can often aid the compiler in effectively prioritizing the functions for exhaustively evaluating the phase order space.

8.2 Redundancy Found by Each Space Pruning Technique

As described in Section 4.1 we employ three techniques to exploit redundancy in the optimization phase order space: detecting dormant phases, detecting identical function in-

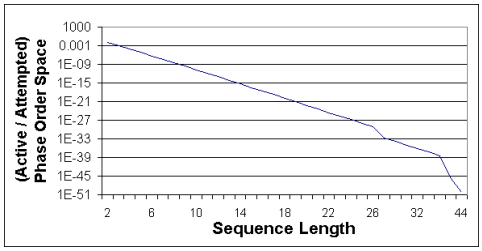


Fig. 12. Ratio of the Active to Attempted Phase Order Search Space for Different Sequence Lengths

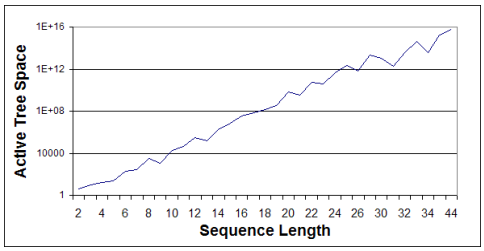


Fig. 13. Active Search Space for Different Sequence Lengths

stances, and detecting equivalent function instances. The *attempted search space* for any function in our compiler with 15 optimization phases is 15^n, where n is the maximum active sequence length for that function. The length of the longest active sequence for various executed functions is listed in Table IV. As explained earlier, phases that are unsuccessful in changing the function when applied are called *dormant* phases. Eliminating the dormant phases from the attempted space results in the *active search space*. Thus, the active search space only consists of phases that are *active*, i.e., successful in changing the program representation when attempted. Figure 12 shows the ratio of the active search space to the attempted search space sorted by different sequence lengths. The comparison of the active space to the attempted space, plotted on a logarithmic scale in Figure 12, shows that the ratio is extremely small, particularly for functions with larger sequence lengths. Figure 13 shows the average number of function instances in the active space for each maximum active sequence length. These two figures show the drastic reduction in attempted search space that is typically obtained by detecting and eliminating dormant phases from the search space.

The remaining pruning techniques find even more redundancy in the active search space. These pruning techniques detect identical and equivalent function instances, which causes different branches of the active search space tree to merge together. Thus, the active search space tree is converted into a DAG. Figure 14 shows the average ratio of the number of distinct function instances to the number of nodes in the active search space tree. Thus, this figure illustrates the redundancy eliminated by detecting identical and equivalent function instances. The fraction of the tree of function instances that is distinct and represented as a DAG decreases as the active sequence length increases. This property of exponential increase in the amount of redundancy detected by our pruning techniques as sequence lengths are increased is critical for exhaustively exploring the search spaces for larger functions.

8.3 Performance Comparison with Optimal

Figure 15(a) shows the distribution of the dynamic frequency count performances of all function instances as compared to the optimal performance. Figure 15(b) illustrates a similar distribution over only the leaf function instances. For both of these graphs, the numbers have been averaged over all the 79 executed functions in our benchmark suite. Function instances that are over 100% worse than optimal are not plotted. On average, 22.27% of total function instances, and 4.70% of leaf instances fall in this category.

Figure 15(a) rates the performances of all function instances, even the un-optimized and partially optimized instances. As a result very few instances compare well with optimal.

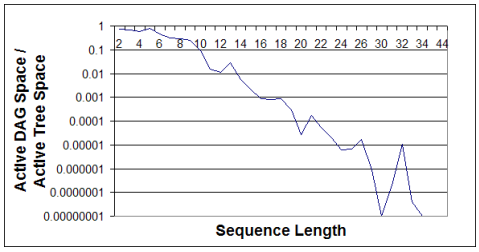
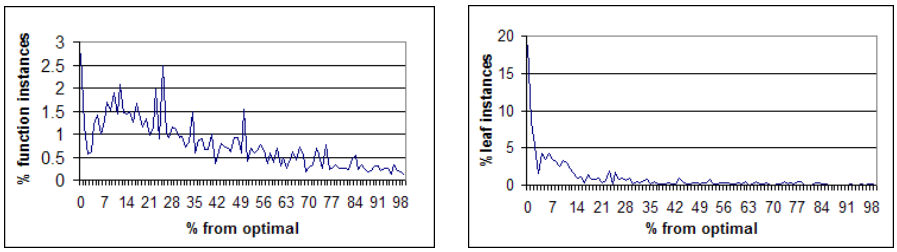


Fig. 14. Ratio of Distinct Function Instances in Active Space



(a) (b)

Fig. 15. Distribution of Performance Compared to Optimal

We expect leaf instances to perform better since these are fully optimized function instances that cannot be improved upon by any additional optimizations. Accordingly, over 18% of leaf instances on average yield optimal performance. Also, a significant number of leaf function instances are seen to perform very close to optimal. This is an important result, which can be used to seed heuristic algorithms, such as genetic algorithms, with leaf function instances to induce them to find better phase orderings faster.

8.4 Optimization Phase Repetition

Figure 16 illustrates the maximum number of times each phase is active during the exhaustive phase order evaluation over all studied functions. The order of the boxes in the legend in Figure 16 corresponds to the order of the phases plotted in the graph, which is also the order in which the phases are described in Table I. Functions with the same maximum sequence length are grouped together, and the maximum phase repetition number of the entire group is plotted. The functions in the figure are sorted on the basis of the maximum sequence length for that function. The optimization phases in Figure 16 are labeled by the *codes* assigned to each phase in Table I.

Common subexpression elimination(c) and *instruction selection*(s) are the phases that are typically active most often in each sequence. These phases clean up the code after many other optimizations, and hence are frequently enabled by other phases. For example, *instruction selection*(s) is required to be performed before *register allocation*(k) so that candidate load and store instructions can contain the addresses of arguments or local scalars. However, after allocating locals to registers, *register allocation*(k) creates many additional opportunities for *instruction selection*(s) to combine instructions. For functions with loops, *loop transformations*(l) may also be active several times due to freeing up of

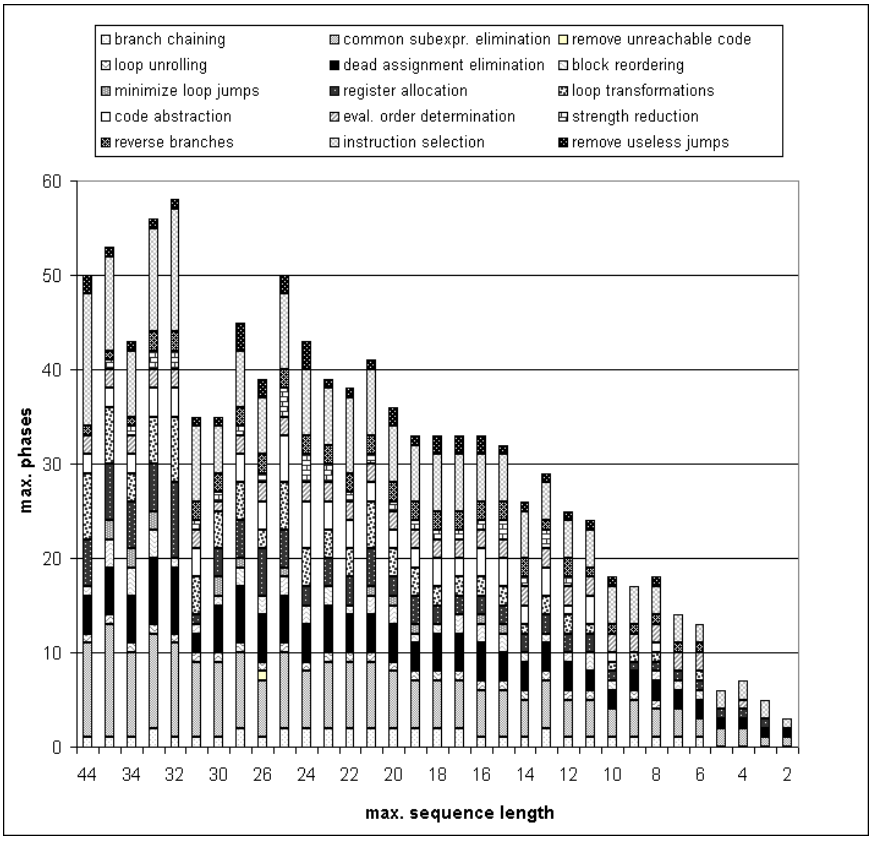


Fig. 16. Repetition Rate of Active Phases

registers, or suitable changes to the instruction patterns by other intertwining optimizations. Most of the branch optimizations, such as *branch chaining*(b), *branch reversal*(r), *block reordering*(i), and *useless jump removal*(u), are not typically enabled by any other optimizations, and so are active at most once or twice during each optimization sequence. As mentioned, *loop unrolling* was restricted to be active at most once in each sequence.

8.5 Analyzing the Best Optimization Sequences

A big motivation for this research is to analyze the optimization sequences resulting in good performance to determine recurring patterns of optimization phases in such sequences. Such analysis can help compiler writers to come up with good phase orderings for conventional compilers. However, any such analysis is made difficult by the sheer number of *good* optimization sequences. Figure 17 shows the number of active phase orderings achieving optimal performance for all 79 *executed* functions. Note that the *number of best active phase orderings* is plotted on a logarithmic scale along the Y-axis in Figure 17. The functions in this figure are sorted from left to right in the order of decreasing number of instructions in the unoptimized version of the function.

Figure 18 plots the percentage of optimal phase orderings amongst all possible active

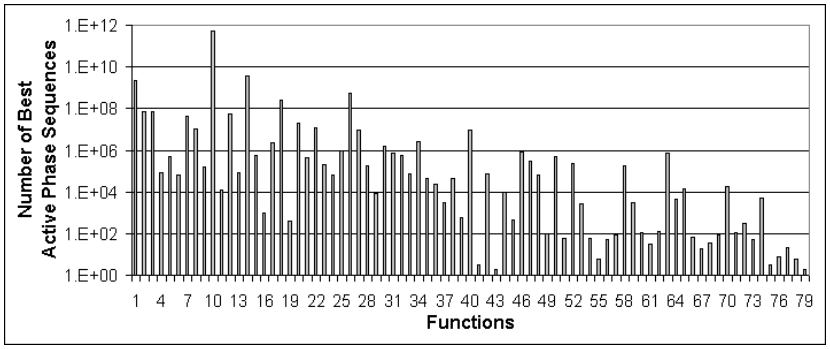


Fig. 17. Number of Optimal Phase Orderings Among All Orderings

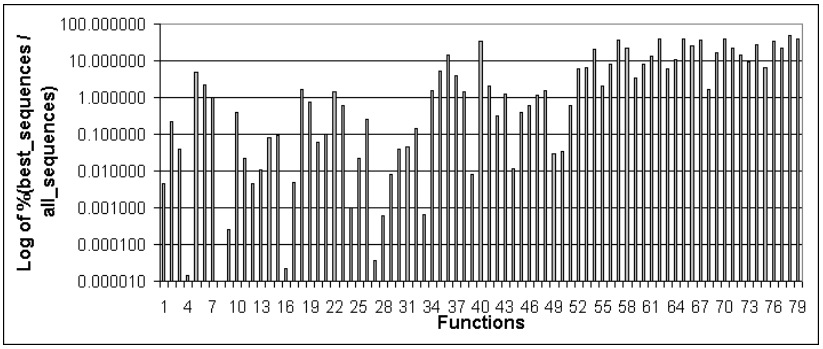


Fig. 18. Ratio of Optimal to All Possible Phase Orderings

phase orderings for the functions shown in Figure 17. This graph also uses a logarithmic scale to plot the percentages along the Y-axis. This figure again reiterates that the total number of active phase orderings for each function can be many orders of magnitude greater than the number of distinct function instances. Consequently, the number of active phase orderings resulting in the best performance is extremely large as well. An interesting trend that can be easily noticed is that this ratio tends to be very small for the larger functions, while a significant percentage of active phase orderings for the smaller functions are in fact optimal.

Figure 19 displays the number of *active* sequences that generated an optimal function instance for multiple functions. For each phase ordering that generated an optimal function instance for one function, we applied that ordering on all 78 other executed functions and compared the resulting performance with the best performance for the corresponding function. [2] Thus, we can see that no single phase ordering produced the best performance for more than 33 (out of 79) distinct executed functions. As mentioned earlier, the batch VPO compiler also reaches optimal performance for 33 functions. However, note that the

[2]Due to an exceptional number of active phase sequences ($5.03 * 10^{11}$) in the function *initckch* in *ispell*, we were unable to complete our analysis of this function for Figure 19.

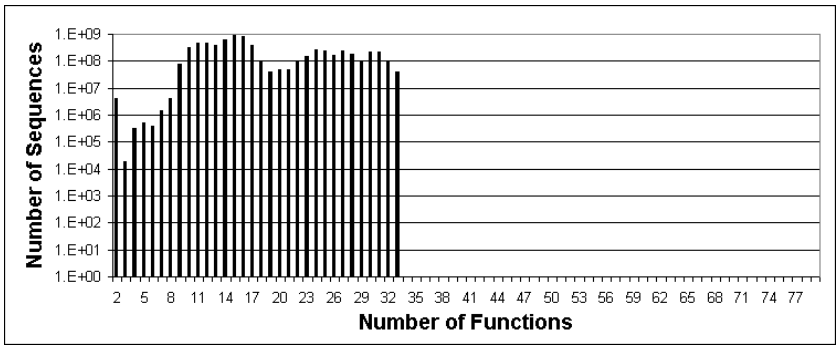


Fig. 19. Number of Active Sequences that Produced an Optimal Function Instance for Multiple Functions

sequence applied by the current batch compiler is much more aggressive. By applying optimization phases in a loop until there are no more program changes allows the batch compiler to generate different phase sequences for different functions depending on correspondingly different active phases. More detailed analysis of the characteristics of good phase orderings is very interesting future work that we plan to pursue.

8.6 Effectiveness of Heuristic Approaches to Address Phase Ordering

In some earlier work we performed a study to determine the effectiveness of popular heuristic approaches, in particular genetic algorithms, in finding *good* per-function optimization sequences efficiently [Kulkarni et al. 2006b]. Since we now know the optimal function instance for each function, our study was the first to determine how close the solutions provided by heuristic algorithms come to the performance provided by the optimal function instance in each case. We found that a basic genetic algorithm is able to quickly converge to its solution in only a few generations in most cases. At the same time, the performance of the delivered function instance is often very close to our optimal solution, with a performance difference of only 0.51% on average. Even more interesting, we were able to extract information regarding the *enabling* and *disabling* interaction between optimization phases from the data provided by our exhaustive phase order exploration runs. We used the phase enable/disable interaction information to modify the baseline genetic algorithm so that it is now able to find the optimal performing function instance in all but one cases, resulting in a performance difference of only 0.02% on average.

We, along with some other researchers, have also compared the performance and efficiency of several different heuristic approaches to address optimization phase ordering [Kulkarni et al. 2007; Almagor et al. 2004]. The conclusions suggest that most heuristic approaches yield comparable performance and efficiency, with more mature approaches based on genetic algorithms, simulated annealing, or hill climbing generally performing much better than a random sampling of the optimization phase order space. We have avoided furnishing detailed analysis of such comparative studies in the current paper, since the earlier publications provide good references.

9. FUTURE WORK

To our knowledge, this work is the first successful attempt at exhaustive optimization phase order space evaluation over all the optimization phases in a mature compiler, and as such has opened up many interesting avenues for future research. We have planned a variety of enhancements to this work. First, we would like to examine methods to 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 can be used to merge phases into mutually independent groups, to reduce the number of effective optimizations during the enumeration process. Remapping of register numbers can also be employed to detect greater equivalence between function instances. Second, we plan to develop additional techniques to further reduce the number of simulations required during exhaustive evaluations. Such reductions can be achieved by unambiguous and accurate prediction of block execution frequencies. We would also need to develop very different techniques to estimate performance for architectures where our measure of dynamic frequency counts does not correlate well with processor cycles. Third, we plan to work on parallelizing our exhaustive enumeration algorithm to make this approach more practical.

We have shown that exhaustive information regarding the optimization phase order space and phase interaction can be used to improve non-exhaustive searches of the phase order space. We plan to continue our work in this area to make heuristic approaches to address optimization phase ordering more widely acceptable. A broader aim of this work is to gain additional insight into the phase ordering problem in order to improve conventional compilation in general. We believe that a priori generation of program specific optimization phase sequences, achieving optimal or near optimal performance, may be possible after a more thorough knowledge of the phase interactions within themselves, as well as with features of the program such as loops and branches.

10. CONCLUSIONS

The compiler phase ordering problem has been a difficult issue to systematically address, and has found widespread attention by the compiler community over the past several decades. Until now it was assumed that the optimization phase order space is too large to exhaustively enumerate on acceptable benchmarks, using all the phases in a mature compiler. In this paper we have effectively solved the phase ordering problem for VPO on the ARM platform, and for the benchmark and input data set studied in this work. The problem required solutions to two related sub-problems. The first sub-problem is enumerating the phase order space for each function. 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 orders of magnitude smaller than the attempted space. The other sub-problem is to determine the dynamic performance of all the enumerated function instances for each function, in order to find the optimal solution. We have demonstrated how we can use properties of the phase order space to drastically reduce the number of simulations required to efficiently and accurately estimate performance for all distinct function instances. We further showed that our estimate of performance bears excellent correlation with simulator cycles in our experimental environment. Our results show that we have been able to completely evaluate the phase order space for 234 out of the 244 functions in our benchmark suite. Our correlation numbers further show that we

can obtain performance very close to optimal by performing a very small number of simulations per function. We have also analyzed the phase order space to identify interesting properties, which may prove helpful for many other related issues in compilers.

11. ADDITIONAL INFORMATION

Instructions regarding downloading the proper version of the VPO compiler to reproduce these results, links to download the MiBench benchmarks, and other relevant information is provided at the following web-page: http://www.ittc.ku.edu/∼kulkarni/research/taco08/taco08.html

REFERENCES

AGAKOV, F., BONILLA, E., CAVAZOS, J., FRANKE, B., FURSIN, G., O'BOYLE, M. F. P., THOMSON, J., TOUSSAINT, M., AND WILLIAMS, C. K. I. 2006. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Washington, DC, USA, 295–305.

ALMAGOR, L., COOPER, K. D., GROSUL, A., HARVEY, T. J., REEVES, S. W., SUBRAMANIAN, D., TORCZON, L., AND WATERMAN, T. 2004. Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM Press, New York, NY, USA, 231–239.

BARR, M. AND MASSA, A. 2006. *Programming Embedded Systems: With C and GNU Development Tools*, Second ed. O'Reilly Media, Inc.

BENITEZ, M. E. AND DAVIDSON, J. W. 1988. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*. ACM Press, 329–338.

BODIN, F., KISUKI, T., KNIJNENBURG, P., O'BOYLE, M., AND ROHOU, E. 1998. Iterative compilation in a non-linear optimisation space. Proceedings of the Workshop on Profile and Feedback Directed Compilation.

BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar tool set, version 2.0. *SIGARCH Computer Architecture News 25*, 3, 13–25.

CAVAZOS, J. AND O'BOYLE, M. F. P. 2006. Method-specific dynamic compilation using logistic regression. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 229–240.

COOPER, K., GROSUL, A., HARVEY, T., REEVES, S., SUBRAMANIAN, D., TORCZON, L., AND WATERMAN, T. 2005. ACME: Adaptive compilation made efficient. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 69–78.

COOPER, K. D., SCHIELKE, P. J., AND SUBRAMANIAN, D. 1999. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*. 1–9.

DAVIDSON, J. W. AND WHALLEY, D. B. 1991. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems 15*, 9 (November), 459–472.

ENGBLOM, J. 2007. Using simulation tools for embedded systems software development. http://www.embedded.com/columns/technicalinsights/199501500?_requestid=111267.

GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*.

HEWLETT-PACKARD. 2000. Parallel programming guide for hp-ux systems. http://docs.hp.com/en/B3909-90003/B3909-90003.pdf (page 74).

HOSTE, K. AND KHOUT, L. E. 2008. Cole: compiler optimization level exploration. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*. ACM, New York, NY, USA, 165–174.

KISUKI, T., KNIJNENBURG, P., AND O'BOYLE, M. 2000. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 237–246.

KISUKI, T., KNIJNENBURG, P., O'BOYLE, M., BODIN, F., AND WIJSHOFF, H. 1999. A feasibility study in iterative compilation. In *Proceedings of International Symposium of High Performance Computing, volume 1615 of Lecture Notes in Computer Science*. 121–132.

KNIJNENBURG, P., KISUKI, T., GALLIVAN, K., AND O'BOYLE, M. 2000. The effect of cache models on iterative compilation for combined tiling and unrolling. In *Proceedings of 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*. 31–40.

KULKARNI, P., HINES, S., HISER, J., WHALLEY, D., DAVIDSON, J., AND JONES, D. 2004. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*. 171–182.

KULKARNI, P., WHALLEY, D., TYSON, G., AND DAVIDSON, J. 2006a. Exhaustive optimization phase order space exploration. In *Proceedings of the Fourth Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 306–318.

KULKARNI, P., WHALLEY, D., TYSON, G., AND DAVIDSON, J. 2006b. In search of near-optimal optimization phase orderings. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers and tool support for embedded systems*. ACM Press, New York, NY, USA, 83–92.

KULKARNI, P., WHALLEY, D., TYSON, G., AND DAVIDSON, J. 2007. Evaluating heuristic optimization phase order search algorithms. In *to appear in the IEEE/ACM International Symposium on Code Generation and Optimization*.

KULKARNI, P., ZHAO, W., MOON, H., CHO, K., WHALLEY, D., DAVIDSON, J., BAILEY, M., PAEK, Y., AND GALLIVAN, K. 2003. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM Press, 12–23.

KULKARNI, P. A., HINES, S. R., WHALLEY, D. B., HISER, J. D., DAVIDSON, J. W., AND JONES, D. L. 2005. Fast and efficient searches for effective optimization-phase sequences. *ACM Transactions on Architecture and Code Optimization 2*, 2, 165–198.

PETERSON, W. AND BROWN, D. 1961. Cyclic codes for error detection. In *Proceedings of the Institute of Radio Engineers*. Vol. 49. 228–235.

REDHAT. 2004. Gnupro 04r1 tools for embedded systems development. http://www.redhat.com/f/pdf/GNUPro-04r1-Embedded-AnnouncementLetter.pdf.

TOUATI, S.-A.-A. AND BARTHOU, D. 2006. On the decidability of phase ordering problem in optimizing compilation. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*. ACM Press, New York, NY, USA, 147–156.

TRIANTAFYLLIS, S., VACHHARAJANI, M., VACHHARAJANI, N., AND AUGUST, D. I. 2003. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 204–215.

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

VIRTUTECH. 2008. Virtualized software development white paper. http://www.virtutech.com/files/whitepapers/wp_simics.pdf.

WAGNER, T. A., MAVERICK, V., GRAHAM, S. L., AND HARRISON, M. A. 1994. Accurate static estimators for program optimization. *SIGPLAN Notices 29*, 6, 85–96.

WEISSTEIN, E. W. 2006. Correlation coefficient. from MathWorld - A Wolfram Web Resource. http://mathworld.wolfram.com/CorrelationCoefficient.html.

WHITFIELD, D. L. AND SOFFA, M. L. 1990. An approach to ordering optimizing transformations. In *Proceedings of the second ACM SIGPLAN symposium on Principles & Practice of Parallel Programming*. ACM Press, 137–146.

WHITFIELD, D. L. AND SOFFA, M. L. 1997. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems 19*, 6, 1053–1084.

ZHAO, M., CHILDERS, B., AND SOFFA, M. L. 2003. Predicting the impact of optimizations for embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN Conference on Language, compiler, and tool for embedded systems*. ACM Press, New York, NY, USA, 1–11.

ZHAO, M., CHILDERS, B., AND SOFFA, M. L. 2005. A model-based framework: An approach for profit-driven optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA, 317–327.