Evaluating Heuristic Optimization Phase Order Search Algorithms

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

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

Abstract

Program-specific or function-specific optimization phase sequences are universally accepted to achieve better overall performance than any fixed optimization phase ordering. A number of heuristic phase order space search algorithms have been devised to find customized phase orderings achieving high performance for each function. However, to make this approach of iterative compilation *more widely accepted and deployed in mainstream compilers, it is essential to modify existing algorithms, or develop new ones that find near-optimal solutions quickly. As a step in this direction, in this paper we attempt to identify and understand the important properties of some commonly employed heuristic search methods by using information collected during an exhaustive exploration of the phase order search space. We compare the performance obtained by each algorithm with all others, as well as with the optimal phase ordering performance. Finally, we show how we can use the features of the phase order space to improve existing algorithms as well as devise new, and better performing search algorithms.*

1. Introduction

Current compilers contain numerous different optimization phases. Each optimization phase analyzes the program and attempts to change the code in some way to produce a semantically equivalent program that performs better for one or some combination of constraints, such as speed, code size and power. Most optimization phases share resources (such as registers), and require certain conditions in the code to be applicable before applying a transformation. Each optimization phase may consume or release resources, as well as create or destroy specific conditions. As a result, phases interact with each other, by generating or removing opportunities for application of other phases. In many performance-oriented application domains it is important to find the best order of applying optimization phases so that very high-quality code can be produced. This problem of finding the best sequence of optimization phases to apply is known as the *phase ordering problem* in compilers.

The space of all possible orderings of optimization phases is huge since most current compilers contain numerous different optimization phases with few restrictions imposed on the order of applying these phases. Prior research has found that no single ordering of phases can achieve optimal performance on all applications or functions [1, 2, 3, 4, 5, 6]. The phase ordering problem is difficult since even after decades of research the relationships and interactions between optimization phases remain ill-understood. The only consensus is that the phase relationships change with the function being optimized, the manner in which optimizations are implemented in the compiler, and the characteristics of the target architecture. Due to such factors, iterative compilers that evaluate numerous different orderings of optimization phases have found wide appeal to search for the best phase order as well as phase parameters on a per-function basis.

Exhaustive exploration of the optimization phase order space, although possible in a reasonable amount of time for a large majority of the functions [7, 8], takes prohibitively long for most large functions to make it suitable for use in typical iterative compilers. Instead, faster heuristic algorithms that scan only a portion of the phase order space are more commonly employed. However, such methods do not evaluate the entire space to provide any guarantees about the quality of the solutions obtained. Commonly used heuristic algorithms to address the phase ordering problem include genetic algorithms [3, 4], hill climbing algorithms [9, 10], as well as random searches of the space [11].

In this paper we evaluate many different heuristic search approaches to determine the important characteristics of each algorithm as related to the phase order space. We also evaluate their performance, comparing the performance with other heuristic approaches as well as with optimal orderings. We are able to perform a very thorough study since

we have completely enumerated the phase order spaces of our compiler for hundreds of functions and can simply lookup information instead of compiling and simulating each ordering of phases. Thus, the main contributions of this paper are:

1. This study is the most detailed evaluation of the performance and cost of different heuristic search methods, and the first to compare their performance with the optimal phase ordering.
2. We isolate and evaluate several properties of each studied heuristic algorithm, and demonstrate the significance and difficulty in selecting the correct optimization phase sequence length, which is often ignored or kept constant in most previous studies on optimization phase ordering.
3. This study identifies and illustrates the importance of leaf function instances, and shows how we can exploit the properties of leaf instances to enhance existing algorithms as well as to construct new search algorithms.

The paper is organized as follows. In the next section we will discuss research related to the current work. Our experimental setup will be described in Section 3. In Section 4 we will describe the results of our study to determine the properties of the phase order search space and evaluate various heuristic search algorithms. Finally, we will present our conclusions in Section 5.

2. Related Work

Optimization phase ordering has been a long standing and important issue in compiler optimizations, and as such has received much recent attention from researchers. Several researchers have attempted to formally specify compiler optimizations in order to systematically address the phase ordering problem. Whitfield and Soffa developed a framework based on axiomatic specifications of optimizations [2, 12]. This framework was employed to list the potential *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 of cyclic interactions between two optimizations, it was not possible to determine a good ordering automatically 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 [13], and the *profitability* of optimizations [14]. Some other work has proposed a Unified Transformation Framework (UTF) to provide a uniform and systematic representation of iteration reordering transformations and their arbitrary combinations [15]. It is possible using UTF to transform the optimization phase order space into a polyhedral space, which is considered by some researchers to be more convenient for a systematic exploration than the original space [16]. However, this work is restricted to loop optimizations, and needs to be extended to other optimizations before it could be adopted in typical iterative compilers.

Earlier work in iterative compilation concentrated on finding good parameter settings for a few optimizations, such as loop unrolling and loop tiling [10, 17, 6]. In cases where exhaustive exploration was expensive, researchers used heuristic algorithms, such as grid-based searches, hill climbers, and genetic algorithms, to scan only a portion of the search space. A common deduction is that typical program search spaces, on a variety of different architectures, generally contain enough local minima that biased sampling techniques should find good solutions. Iterative compilation usually comes at the cost of a large number of program executions. In order to reduce the cost, some studies have looked at incorporating static architectural models, particularly cache models, into their approach [18].

Research on the phase ordering problem over all or most of the optimization phases in a compiler has typically concentrated on finding effective (rather than optimal) phase sequences by evaluating only a portion of the phase order search space. A method, called Optimization-Space Exploration [5], uses static performance estimators to reduce the search time. In order to prune the search they limit the number of configurations of optimization-parameter value pairs to those that are likely to contribute to performance improvements. This area has also seen the application of other search techniques to *intelligently* search the optimization space. Hill climbers [9] and genetic algorithms [3, 4] have been employed during iterative algorithms to find optimization phase sequences better than the default one used in their compilers. Such techniques are often combined with aggressive pruning of the search space [19, 20] to make searches for effective optimization phase sequences faster and more efficient. Successful attempts have also been made to use predictive modeling and code context information to focus search on the most fruitful areas of the phase order space for the program being compiled [11]. Other approaches to reduce compilation time estimate the number of instructions executed and use that as a means to prune the number of versions of a program that need to be executed or simulated for evaluation [21, 8].

Researchers have also attempted a *near-optimal* resolution of the phase ordering problem considering all the phases present in their compilers. Kulkarni et al. demonstrated that for typical optimization phases applied in a compiler backend it is possible to exhaustively evaluate the phase order space for most of the functions in several embedded benchmarks [7, 8]. This was made possible by using

techniques to drastically prune the space of distinct function instances, as well as reducing the number of program executions to estimate the performances of all nodes in the space. However, they were unable to exhaustively explore the space for some of the largest functions, while some other large functions each took several days to evaluate. This shows that exhaustive space evaluation may still be infeasible for routine use in typical iterative compilers.

In spite of the wide-spread use of heuristic approaches, there have been few attempts to evaluate and compare their properties and performance in the context of the characteristics of the phase order space. Kisuki et al. analyzed the performance of five different search algorithms, and reported the observation that heuristic algorithms do not differ much in their efficiency [10]. However, this study was performed on a space of only two optimizations (with different parameters), and did not take in to account properties of the phase order space. A more detailed evaluation was performed by Almagor et al. [9], which attempted to relate features of the phase order space with the efficiency and performance of different heuristic algorithms. This study was, however, incomplete since they had restricted their sequence length to be of a fixed size. This earlier work also did not have access to the entire phase order space features, and lacked a knowledge of the optimal phase order performance.

3. Experimental Framework

In this section we first describe the compiler framework, and then explain our experimental setup.

3.1. Compiler Framework

The research in this paper uses the Very Portable Optimizer (VPO) [22], which 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 since different functions may have very different best orderings, so any strategy that requires all functions in a file to have the same phase order will almost certainly not be optimal. At the same time, restricting the phase ordering problem to a single function helps to make the phase order space more manageable. The compiler has been targeted to generate code for the StrongARM SA-100 processor using Linux as its operating system. We used the SimpleScalar set of functional simulators [23] for the ARM to get dynamic performance measures.

Table 1 describes each of the 15 candidate code-improving phases that were used during the exhaustive exploration of the optimization phase order search space. In addition, *register assignment*, which is a compulsory phase that assigns pseudo registers to hardware registers, must be performed. VPO implicitly performs register assignment before the first code-improving phase in a sequence that requires it. After applying the last code-improving phase in a sequence, VPO performs another compulsory phase that inserts instructions at the entry and exit of the function to manage the activation record on the run-time stack. Finally, the compiler also performs *predication* and *instruction scheduling* before the final assembly code is produced. These last two optimizations should only be performed late in the compilation process in the VPO compiler, and so are not included in the set of re-orderable optimization phases.

For the experiments described in this paper we used a subset of the benchmarks from the *MiBench* benchmark suite, which are C applications targeting specific areas of the embedded market [24]. We selected two benchmarks from each of the six categories of applications present in MiBench. Table 2 contains descriptions of these programs. VPO compiles and optimizes individual functions at a time. The 12 benchmarks selected contained a total of 244 functions, out of which 88 were executed (at least once) with the input data provided with each benchmark.

3.2. Experimental Setup

For this study we exhaustively enumerate the phase order space for a large number of functions. This enumeration enables us to accurately investigate the properties of the search space, to study heuristic search algorithms, to tune the algorithms, to suggest new algorithms, as well as to compare the efficiency of different heuristic algorithms in finding optimal phase ordering for each function. In this section we will describe our setup for the current research.

We use the algorithm proposed by Kulkarni et al. [7, 8] to exhaustively evaluate the phase order space for the functions in our benchmark suite. The algorithm is illustrated in Figure 1. The phase ordering problem is viewed as an evaluation of all possible distinct function instances that can be generated by changing the order of optimization phases in a compiler. This approach to the phase ordering problem makes no assumptions about the phase sequence length for each function, and allows phases to be repeated as many times as they can possibly be active in the same sequence. Thus, the phase ordering space is complete in each case. However, it is important to 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 VPO. Thus, *optimal* in the context of this work refers to the best code that can be produced by any optimization phase ordering in VPO and is not meant to imply a universally optimum solution. The algorithm can be briefly summarized as follows:

Optimization Phase	Gene	Description
branch chaining	b	Replaces a branch or jump target with the target of the last jump in the jump chain.
common subexpression elimination	c	Performs global analysis to eliminate fully redundant calculations, which also includes global constant and copy propagation.
remove unreach. code	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 runtime and to aid scheduling at the cost of code size increase.
dead assign. elim.	h	Uses global analysis to remove assignments when the assigned value is never used.
block reordering	i	Removes a jump by reordering blocks when the target of the jump has only a single predecessor.
minimize loop jumps	j	Removes a jump associated with a loop by duplicating a portion of the loop.
register allocation	k	Uses graph coloring to replace references to a variable within a live range with a register.
loop transformations	l	Performs loop-invariant code motion, recurrence elimination, loop strength reduction, and induction variable elimination on each loop ordered by loop nesting level.
code abstraction	n	Performs cross-jumping and code-hoisting to move identical instructions from basic blocks to their common predecessor or successor.
eval. order determ.	o	Reorders instructions within a single basic block in an attempt to use fewer registers.
strength reduction	q	Replaces an expensive instruction with one or more cheaper ones. For this version of the compiler, this means changing a multiply by a constant into a series of shift, adds, and subtracts.
reverse branches	r	Removes an unconditional jump by reversing a cond. branch when it branches over the jump.
instruction selection	s	Combines pairs or triples of instructions together where the instructions are linked by set/use dependencies. Also performs constant folding and checks if the resulting effect is a legal instruction before committing to the transformation.
remove useless jumps	u	Removes jumps and branches whose target is the following positional block.

Table 1. Candidate Optimization Phases along with Their Designations

1. The algorithm starts with the un-optimized function instance. A depth-first search algorithm is used to produce the next sequence to evaluate. Each new sequence appends one phase to a previous sequence resulting in a new function instance, in depth first order.
2. Phases not successful in changing the program representation do not need further evaluation.
3. The next stage uses CRC hash values [25], calculated on the entire function, to compare the current function instance with all previous distinct function instances. If the current function instance is identical to an instance previously produced by some other phase sequence, then only one needs to be evaluated, and so the current instance is discarded.
4. Even if two function instances are not identical, it is possible that the only differences may lie in the register numbers being used, or the labels assigned to the various basic blocks. In such cases the two function instances will still perform identically, and so the current function instance no longer needs further evaluation.
5. The next step determines the performance of each distinct function instance. In order to reduce the number of program simulations, the algorithm only simulates one function instance from the set of function instances having the same basic block control flow. The first function instance with a new block control flow is instrumented and simulated to obtain the number of times each block is executed. The *dynamic frequency measure* for the each function instance is determined by multiplying the block execution counts by the estimated number of static cycles for each block. These dynamic frequency measures have been shown to bear a strong correlation with simulated cycles.

Category	Program	Description
auto	bitcount	test proc. bit manipulation abilities
	qsort	sort strings using the quicksort algo.
network	dijkstra	Dijkstra's shortest path algorithm
	patricia	construct patricia trie for IP traffic
telecomm	fft	fast fourier transform
	adpcm	compress 16-bit linear PCM samples to 4-bit samples
consumer	jpeg	image compression and decomp.
	tiff2bw	convert color *tiff* image to b&w image
security	sha	secure hash algorithm
	blowfish	symmetric block cipher with variable length key
office	search	searches for given words in phrases
	ispell	fast spelling checker

Table 2. MiBench Benchmarks Used

The exhaustive enumeration of any function is stopped if the time required exceeds an approximate limit of two

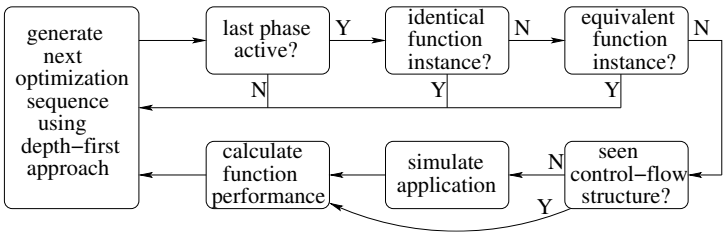
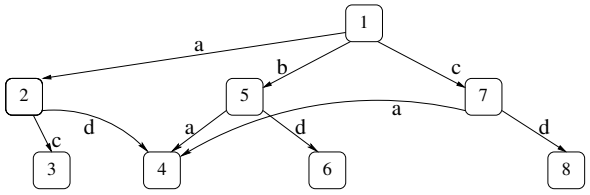


Figure 1. Steps During an Exhaustive Evaluation of the Phase Order Space for Each Function

weeks. Using the above algorithm and cut-off criteria we were able to enumerate completely the optimization phase order space for 234 out the 244 functions that we studied. Out of the 88 executed functions, we were able to completely enumerate 79. We represent the phase order space for every function in the form of a DAG (Directed Acyclic Graph), as shown in Figure 2. Nodes in the DAG represent distinct function instances and edges represent transitions from one node to the next on application of an optimization phase. The DAG then enables much faster evaluation of any search heuristic, since compilation as well as execution can be replaced with a simple table lookup in the DAG to determine the performance of each phase ordering. As a result, the study of the various algorithms is fast, and it is possible to evaluate various parameters of the algorithms as well as the search space.



(b) Depth−first Traversal

Figure 2. Directed Acyclic Graph Representing the Phase Order Space of a Function

4 Study of Common Heuristic Search Techniques

Over the past decades researchers have employed various heuristic algorithms to cheaply find effective solutions to the phase ordering problem. However, several issues regarding the relative performance and cost of each algorithm, as well as the effect of changing different algorithm parameters on that algorithm's performance are as yet uncertain and not clearly understood. In this section we will perform a thorough evaluation and comparison of commonly used heuristic methods. Based on the phase order space characteristics we will also develop new techniques and suggest several improvements to existing algorithms.

4.1. Local Search Techniques

Local search techniques, such as hill climbing and simulated annealing, can only migrate to neighboring points from one iteration to the next during their search for good solutions. Central to these algorithms is the definition of *neighbors* of any point in the space. For this study, we define the neighbors of a sequence to be all those sequences that differ from the base sequence in only one position. Thus, for a compiler with only three optimization phases *a, b* and *c*, the sequence shown in the first column of Table 3 will have the sequences listed in the following columns as its neighbors. The position that differs in each neighbor is indicated in bold. For a compiler with m optimization phases, a sequence of length n will have $(m-1)n$ neighbors. Unless the search space is extremely smooth, these local search algorithms have a tendency to get stuck in a local minimum, which are points that are not globally minimum, but have better fitness values than any of their neighbors. For a comprehensive study of these algorithms it is important to first understand relevant properties of the optimization phase order space. The results from this study are presented in the next section.

bseq	neighbors							
a	**b**	**c**	a	a	a	a	a	a
b	b	b	**a**	**c**	b	b	b	b
c	c	c	c	c	**a**	**b**	c	c
a	a	a	a	a	a	a	**b**	**c**

Table 3. Neighbors in Heuristic Searches

4.1.1 Distribution of Local Minima in the Phase Order Space

Earlier studies have attempted to probe the properties of the phase order space [17, 9]. Such studies, however, only

looked at a small portion of the space, and ignored important factors affecting the nature and distribution of local minima in phase order spaces. One such factor, commonly ignored, is the optimization sequence length. It is almost impossible to estimate the best sequence length to use due to the ability and tendency of optimization phases to enable other phases. During our experiments, maximum sequence lengths of active phases varied from 3 to 44 over different functions, with considerable variation within the same function itself for different phase orderings. The goal of analyzing the search space, in the context of local search techniques, is to find the properties and distribution of all local minima in each phase order search space. However, there are some difficult hurdles in achieving this goal:

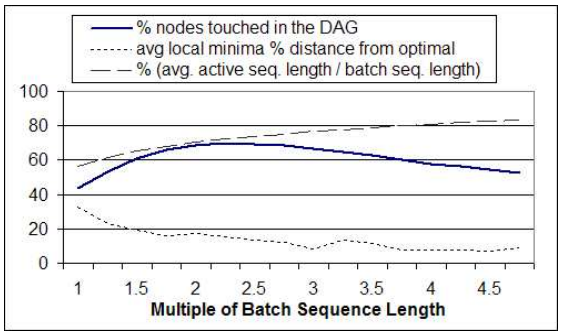
Variable sequence length: Since the best sequence length for each function is unknown, an ideal analysis would require finding the properties of local minima for all possible sequence lengths. This requirement is needed because any sequence of attempted phases of any length defines a point in the search space DAG. Conversely, a single point in the space can be defined by, potentially, infinite number of attempted sequences of different lengths. This is important, since different sequences defining the same point will have different neighbors. This implies that some of those sequences may be locally optimum, while others may be not, even though they define the same point in the phase order space. For example, the attempted sequences $\{\mathbf{b} \rightarrow \mathbf{a}\}$, $\{c \rightarrow \mathbf{b} \rightarrow \mathbf{a}\}$, and $\{d \rightarrow \mathbf{b} \rightarrow c \rightarrow \mathbf{a}\}$ all define the same node 4 in the DAG in Figure 2 (Note that, the phases **a** and **b**, indicated in bold, are active, while c and d are dormant). Thus, we can see that it is possible to have sequences of different lengths pointing to the same node. Thus, this ideal goal of finding the local minima for all possible sequence lengths is clearly impossible to achieve.

Fixed sequence length: A conceptually simpler approach would be to use some *oracle* to give us the best sequence length to use for each function, and then only analyze the space for this single sequence length. The minimum reasonable length to use, so that all nodes in the DAG can be reached, would be the maximum active sequence length for each function. For an average maximum active sequence length of 16, over all 234 enumerated functions, we would need to evaluate 15^{16} different phase orderings for each function. Evaluation of any phase ordering to determine if that ordering is a local optimum would in turn require us to lookup the performance of that ordering as well as that of its $15 * 16$ neighbors. This, also, is clearly a huge undertaking considering that the maximum active sequence length we encountered during our exhaustive phase order enumeration study was 44.

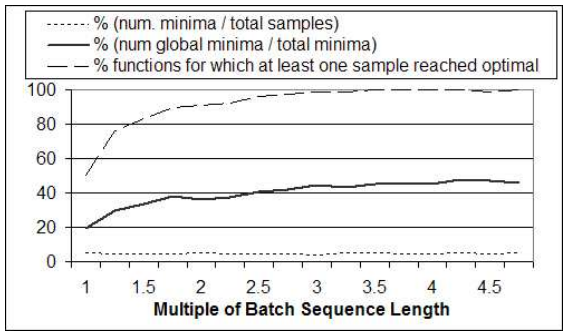
Due to such issues, in our present experiments, we decided to use *sampling* to probe only a reasonable portion of the phase order search space for some number of different sequence lengths for each function. We use 16 different sequence lengths. The initial length is set to the length of the sequence of active phases applied by the conventional VPO compiler in batch mode. The remaining sequence lengths are successive increments of one-fourth of the initial sequence length used for each function. The larger sequence lengths may be needed to accommodate phases which may be dormant at the point they are attempted. For each set of experiments for each function, we first randomly generate a sequence of the specified length. We then compare the performance of the node that this sequence defines with the performance of all of its neighbors to find if this sequence is a local optimal. This base node is marked as *done*. All later sequences are constrained to define different nodes in the space. As this sampling process progresses it will require an increasing number of attempts to find a sequence corresponding to an unevaluated node in the search space. The process terminates when the average number of attempts to generate a sequence defining a new node exceeds 100.

Figures 3(a) and 3(b) illustrate the average phase order space properties over all the executed functions that we studied. The plot labeled *% nodes touched in the DAG* from Figure 3(a) shows the percentage of nodes that were evaluated for local minimum from amongst all nodes in the space DAG. This number initially increases, reaches a peak, and then drops off. This graph, in effect, shows the nature of typical phase order spaces. Optimization phase order space DAGs typically start out with a small width, reach a maximum around the center of the DAG, and again taper off towards the leaf nodes as more and more function instances generated are detected to be redundant. Smaller attempted sequence lengths in Figure 3(a) define points higher up in the DAG, with the nodes defined dropping down in the DAG as the length is increased. The next plot labeled *avg local minima % distance from optimal* in Figure 3(a) measures the average difference in performance from optimal over all the samples at each length. As the sequence lengths increased the average performance of the samples gets closer and closer to optimal, until after a certain point the performance remains more or less constant. This is expected, and can be explained from the last plot in Figure 3(a), labeled *%(avg.active seq. length / batch seq. length)*, which shows the percentage increase in the average length of active phases as the attempted sequence length is increased. The ability to apply more active phases implies that the function is better optimized, and thus we see a corresponding increase in performance and a smaller percentage of active phases.

The first plot in Figure 3(b) shows the ratio of sequences reaching local minima to the total sequences probed. This ratio seems to remain more or less constant for different lengths. The small percentage of local minima in the total samples indicates that there are not many local minima



(a) Local Minima Information



(b) Global Minima Information

Figure 3. Search Space Properties

in the space. The next plot, *%(num global minima / total minima)*, in this figure shows that the percentage of locally minimum nodes achieving global minima grows with increase in sequence length. This increase is more pronounced initially, but subsequently becomes steadier. In the steady state around 45% of local minima display globally optimum performance. This characteristic means that for longer sequence lengths there is a good chance that the local minimum found during local search algorithms will have globally optimal performance. The final plot in Figure 3(b), *%(functions for which at least one sample reached optimal)*, presents the percentage of functions for which the probe is able to find optimal in at least one of its samples. This number shows a similar characteristic of continuously increasing with sequence lengths, until it reaches a steady state at close to 100% for larger sequence lengths. Hence, for small multiples of the batch sequence length the local search algorithms should be able to find global minima with a high probability for most of the functions.

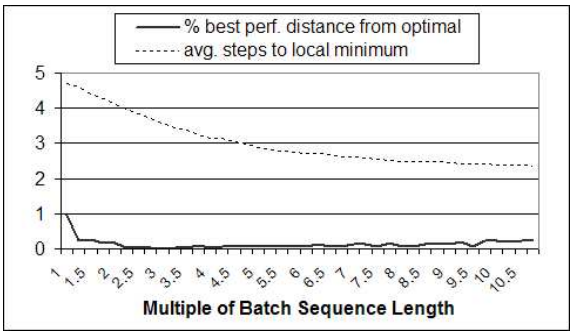
Thus, this study illustrates that it is important to find the correct balance between increase in sequence length, performance obtained, and the time required for the search. Although larger sequence lengths tend to perform better they are also more expensive to evaluate, since they have more neighbors, and evaluation of each neighbor takes longer. It is worthwhile to note that we do not need to increase the sequence lengths indefinitely. After a modest increase in the sequence lengths, as compared to the fixed batch sequence length, we are able to obtain most of the potential benefits of any further increases in sequence lengths.

4.1.2 Hill Climbing

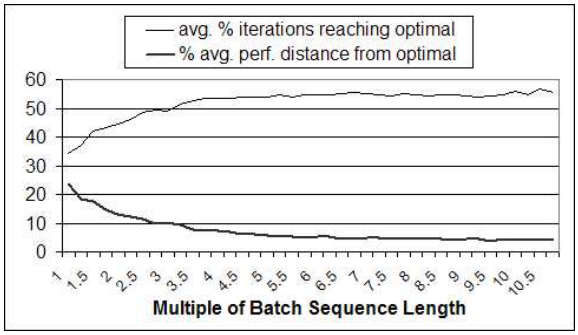
In this section we evaluate the performance of the *steepest descent hill climbing* heuristic algorithm for different sequence lengths [9]. The algorithm is initiated by randomly populating a phase sequence of the specified length. The performance of this sequence is evaluated, along with that of all its neighbors. If the best performing neighbor has equal or better performance than the base sequence, then that neighbor is selected as the new base sequence. This process is repeated until a local optimum is reached, i.e., the base sequence performs better than all of its neighbors. For each sequence length, 100 iterations of this algorithm are performed by selecting random starting points in the search space. The sequence lengths were incremented 40 times starting from the length of the active batch sequence, with each increment equal to one-fourth the batch length.

Figures 4(a) and 4(b) illustrate the results of the hill climbing experiments. The plot marked *% best perf. distance from optimal* in Figure 4(a) compares the best solution found by the hill climbing algorithm with optimal, averaged over the 79 executed functions, and over all 100 iterations for each sequence length. We can see that even for small sequence lengths the algorithm is able to obtain a phase ordering whose best performance is very close to optimal. For lengths greater than 1.5 times the batch sequence length, the algorithm is able to reach optimal in most cases. The plot *avg. steps to local minimum* in Figure 4(a) shows that the simple hill climbing algorithm requires very few steps to reach local optimal, and that the average distance to the local optimal decreases with increase in sequence lengths. This decrease in the number of steps is caused by better performance delivered by each typical sequence when the initial sequence length is increased, so that in effect the algorithm starts out with a better initial sequence, and takes fewer steps to the local minimum.

As mentioned earlier, the hill climbing algorithm is iterated 100 times for each sequence length and each function to eliminate the noise caused by the random component of the algorithm. The first plot in Figure 4(b), *avg. % iterations reaching optimal*, illustrates that the average number of iterations reaching optimal increases with increase in the sequence length up to a certain limit, after which it remains more or less constant. A related measure *% avg. perf.*



(a) Local Minima Information



(b) Global Minima Information

Figure 4. Properties of the Hill Climbing Algorithm

distance from optimal, shown in the second plot in Figure 4(b), is the average function performance over all the iterations for each sequence length. This measure also shows a marked improvement as the sequence length increases until the average performance peaks at around 4% worse than optimal. These results indicate the significance of selecting a correct sequence length during the algorithm. Larger sequence lengths lead to larger active sequences that result in the initial performance improvement, but increasing the length incessantly gives diminishing returns while making the algorithm more expensive.

4.1.3 Simulated Annealing

Simulated annealing can be defined as a technique to find a good solution to an optimization problem by trying random variations of the current solution. A worse variation is accepted as the new solution with a probability that decreases as the computation proceeds. The slower the cooling schedule, or rate of decrease, the more likely the algorithm is to find an optimal or near-optimal solution [26]. In our implementation, the algorithm proceeds similarly to the hill climbing algorithm by starting from a random initialization point in the phase order space. The sequence length is fixed for each run of the algorithm. During each iteration the performance of the base sequence is evaluated along with that of all its neighbors. Similar to the hill climbing method, if the performance of the best performing neighbor is better than the performance of the base sequence, then that neighbor is selected as the base sequence for the next iteration. However, if the current iteration is not able to find a neighbor performing better than the base sequence, the algorithm can still migrate to the best neighbor based on its current *temperature*. The worse solution is generally accepted with a probability based on the Boltzmann probability distribution:

$$prob = exp(-\frac{\delta f}{T}) \quad (1)$$

where, δf is the difference in performance between the current base sequence and the best neighbor, and T is the current temperature. Thus, smaller the degradation and higher the temperature the greater the probability of a worse solution being accepted.

An important component of the simulated annealing algorithm is the *annealing schedule*, which determines the initial temperature and how it is lowered from high to low values. The assignment of a good schedule generally requires physical insight and/or trial and error experiments. In this paper, we attempt to study the effect of different annealing schedules on the performance of a simulated annealing algorithm. For this study, the sequence length is fixed at 1.5 times the batch compiler length of active phases. As seen in the hill climbing experiments, this is the smallest sequence length at which the average performance reaches a steady state that is very close to optimal. We conducted a total of 400 experimental runs by varying the initial temperature and the annealing schedule. The temperature was varied from 0 to 0.95 in steps of 0.5. For each temperature we defined 20 different annealing schedules, which control the temperature in steps from 0.5 to 0.95 per iteration. The results for each configuration are averaged over 100 runs to account for noise caused by random initializations.

Our results, shown in Figure 5, indicate that for the phase ordering problem, as seen by our compiler, the initial temperature as well as the annealing schedule do not have a significant impact on the performance delivered by the simulated annealing algorithm. The best performance obtained over all the 400 experimental runs is, on average, 0.15% off from optimal, with a standard deviation of 0.13%. Likewise, other measures obtained during our experiments are also consistent across all 400 runs. The average number of iterations achieving optimal performance during each run is 41.06%, with a standard deviation of 0.81%. The average performance for each run is 15.95% worse than optimal, with a deviation of 0.55%. However, as expected,

the number of steps to a local minimum during each iteration for each run increases with increase in the initial temperature and the annealing schedule step. As the starting temperature and annealing schedule step are increased, the algorithm accepts more poorly performing solutions before halting. However, this increase in the number of steps to local optimal does not translate into any significant performance improvement for our experiments,

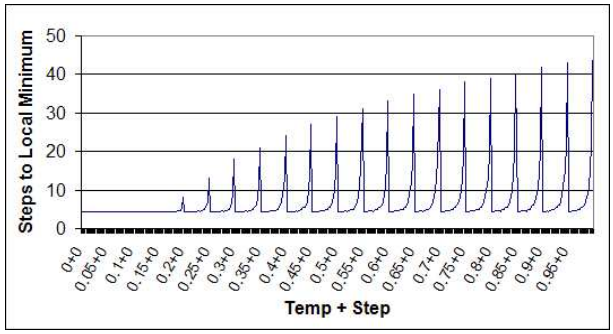


Figure 5. Increase in the Number of Steps to Local Minimum with Increases in Initial Temperature and Annealing Schedule Step

4.2. Greedy Algorithm

Greedy algorithms follow the policy of making the locally optimum choice at every step in the hope of finally reaching the global optimum. Such algorithms are commonly used for addressing several optimization problems with huge search spaces. For the phase ordering problem, we start off with the empty sequence as the *base* sequence. During each iteration the algorithm creates new sequences by adding each available phase first to the prefix and then as the postfix of the base sequence. Each of these sequences is evaluated to find the best performing sequence in the current iteration, which is consequently selected as the base sequence for the next iteration. If there are multiple sequences obtaining the same best performance, then one of these is selected at random. The algorithm is repeated 100 times in order to reduce the noise that can potentially be caused by this random component in the greedy method. Thus, in our case the algorithm has a bounded cost, as it performs a fixed number of (15+15=30) evaluations in each step, where 15 is the number of available optimizations in our compiler.

Our current implementation of the greedy algorithm is inspired by the approach used by Almagor et al. [9]. Similar to the hill climbing algorithm, the sequence lengths during the greedy algorithm are varied from the active batch sequence length for each function as the initial length to 11 times the batch length, in increments of one-fourth the batch length. To minimize the effect of the random component, the algorithm is repeated 100 times for each sequence length. The best and average performances during these 100 iterations for each sequence length, averaged over all executed functions, are illustrated in Figure 6. The plots show a similar pattern to the hill climbing performance graphs. However, it is interesting to note that the best achievable performance during the greedy algorithm is around 1.1% worse than optimal, whereas it is very close to optimal (0.02%) for the hill climbing algorithm. Also, the average performance during the greedy algorithm improves more gradually and continues to improve for larger sequence lengths as compared to hill climbing.

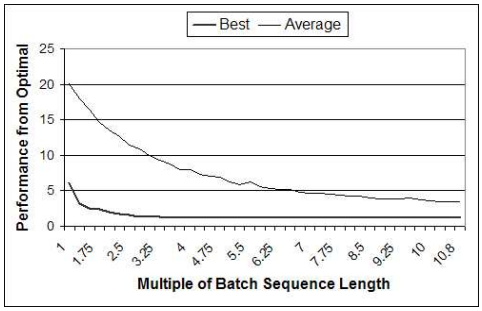


Figure 6. Greedy Algorithm Performance

4.3. Focusing on Leaf Sequences of Active Phases

Leaf function instances are those that cannot be further modified by the application of any additional optimizations phases. These function instances represent leaves in the DAG of the phase order space (e.g. nodes 3, 4, 6, and 8 in Figure 2). Sequences of active phases leading to leaf function instances are called *leaf sequences*. Working with only the leaf sequences has the advantage that the heuristic algorithm no longer needs to guess the most appropriate sequence length to minimize the algorithm running time, while at the same time obtaining the best, or at least close to the best possible performance. Since leaf function instances are generated by different lengths of active phase sequences, the length of the leaf sequences is variable. In this section we describe our modifications to existing algorithms, as well as introduce new algorithms that deal with only leaf sequences.

We first motivate the reason for restricting the heuristic searches to only leaf function instances. Figure 7 shows the distribution of the dynamic frequency counts as compared to the optimal for all distinct function instances obtained during our exhaustive phase order space evaluation, 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 with respect to the dynamic frequency counts. This fact is quite intuitive since active optimizations generally improve performance, and very rarely cause a performance degradation. The main drawback of this approach is that the algorithm will not find the optimal phase ordering for any function that does not have an optimal performing leaf instance. However, we have observed that most functions do contain optimal performing leaf instances. For more than 86% of the functions in our benchmark suite there is at least one leaf function instance that achieved optimal dynamic frequency counts. The average best performance for leaf function instances over all executed functions is only 0.42% worse than optimal. Moreover, leaf function instances comprise only 4.38% of the total space of distinct function instances, which is in turn a minuscule portion of the total phase order search space. Thus, restricting the heuristic search to only the leaf function instances constrains the search to only look at a very small portion of the search space that typically consists of good function instances, and increases the probability of finding a near-optimal solution quickly. In the next few sections we will describe some modifications to existing algorithms, as well as describe new algorithms that take advantage of the opportunity provided by leaf function instances to find better performance faster.

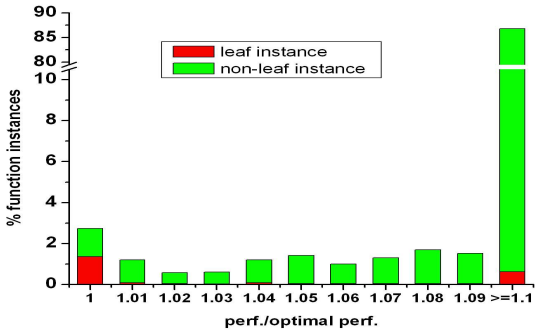
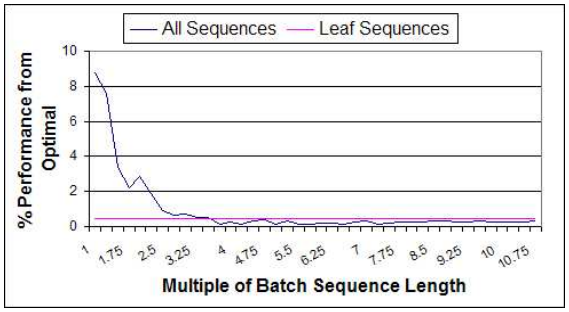


Figure 7. Average Distribution of Dynamic Frequency Counts

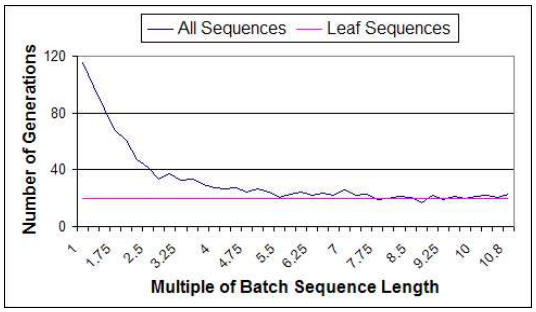
4.3.1 Genetic Algorithm

Genetic algorithms are adaptive algorithms based on Darwin's theory of evolution [27]. These algorithms have been successfully employed by several researchers to address the phase ordering problem and other related issues in compilers [4, 3, 28, 29]. *Genes* correspond to optimization phases and *chromosomes* correspond to optimization sequences in the genetic algorithm. The set of chromosomes currently under consideration constitutes a *population*. The number of *generations* is how many sets of populations are to be evaluated. Our experiments with genetic algorithms suggests that minor modifications in the configuration of these parameters do not significantly affect the performance delivered by the genetic algorithms. For the current study we have fixed the number of chromosomes in each population at 20. Chromosomes in the first generation are randomly initialized. After evaluating the performance of each chromosome in the population, they are sorted in decreasing order of performance. During crossover, 20% of chromosomes from the poorly performing half of the population are replaced by repeatedly selecting two chromosomes from the better half of the population and replacing the lower half of the first chromosome with the upper half of the second and vice-versa to produce two new chromosomes each time. During mutation we replace a phase with another random phase with a small probability of 5% for chromosomes in the upper half of the population and 10% for the chromosomes in the lower half. The chromosomes replaced during crossover are not mutated.

The only parameter that seems to significantly affect the performance of the genetic algorithm is the length of each chromosome. We conducted two different studies with genetic algorithms. In the first study we vary the length of the chromosomes (attempted sequence) starting from the batch sequence length to 11 times the batch sequence length, in steps of one-fourth of the batch length. For the second study we modified the genetic algorithm to only work with leaf sequences of active phases. This approach requires maintaining active leaf sequences of different lengths in the same population. After crossover and mutation it is possible that the new sequences no longer correspond to leaf function instances, and may also contain dormant phases. The modified genetic algorithm handles such sequences by first squeezing out the dormant phases and then extending the sequence, if needed, by additional randomly generated phases to get a leaf sequence. Figures 8(a) and 8(b) shows the performance results, as well as a comparison of the two approaches. Since the modified genetic algorithm for leaf instances does not depend on sequence lengths, the average performance and cost delivered by this new algorithm are illustrated by single horizontal lines in Figures 8(a) and (b).The number of generations is a measure of the cost of the algorithm. Thus, by concentrating on only the leaf function instances, the genetic algorithm is able to obtain close to the best performance at close to the least cost possible for any sequence length. Interestingly, performance of the genetic algorithm for leaf sequences (0.43%) is very close to the best achievable average leaf performance (0.42%).

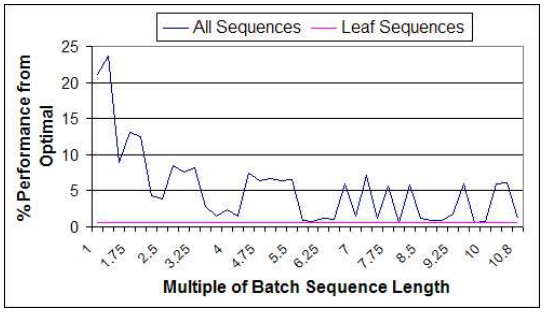


(a) Performance

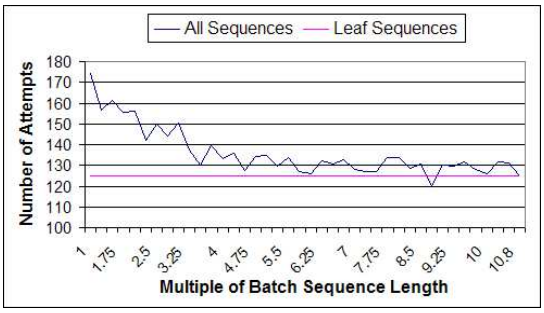


(b) Cost

Figure 8. Performance and Cost of Genetic Algorithms



(a) Performance



(b) Cost

Figure 9. Performance and Cost of Random Search Algorithms

4.3.2 Random Search

Random sampling of the search space to find good solutions is an effective technique for search spaces that are typically discrete and sparse, and when the relationships between the various space parameters are not clearly understood. Examples are the search spaces that we are dealing with to address the phase ordering problem. In this study we have attempted to evaluate random sampling, again by performing two different sets of experiments similar to the genetic algorithm experiments in the previous section. For the first set, randomly constructed phase sequences of different lengths are evaluated until 100 consecutive sequences fail to show an improvement over the current best. The second set of experiments is similar, but only considers leaf sequences or leaf function instances.

Figures 9(a) and 9(b) show the performance benefits as well as the cost for all our random search experiments. It is interesting to note that random searches are also able to achieve performance close to the optimal for each function in a few number of attempts. Since our algorithm configuration mandates the best performance to be held steady for 100 consecutive sequences, we see that the cost of our algorithm is always above 100 attempts. We again notice that leaf sequences consistently obtain good performance for the random search algorithm as well. In fact, for our current configuration, random search algorithm concentrating on only the leaf sequences is able to cheaply outperform the best achievable by any other random search algorithm for any sequence length.

4.3.3 N-Lookahead Algorithm

This algorithm scans N levels down the search space DAG from the current location to select the phase that leads to the best subsequence of phases to apply. The critical parameter is the number of levels to scan. For a N lookahead algorithm we have to evaluate 15^N different optimization phases to select each phase in the base sequence. This process can be very expensive, especially for larger values of the lookahead N. Thus, in order for this approach to be feasible we need to study if small values of the lookahead N can achieve near optimal performance for most of the functions.

For the current set of experiments we have constrained the values of N to be either 1, 2, or 3 levels of lookahead. Due to the exponential nature of the phase order search space, we believe that any further increase in the lookahead

value will make this method too expensive in comparison with other heuristic approaches. Table 4 shows the average performance difference from optimal for the three levels of lookahead over all the executed functions in our set. As expected, the performance improves as the levels of lookahead are increased. However, even after using three levels of lookahead the performance is far from optimal. This illustrates the ragged nature of typical phase order search spaces, where it is difficult to predict the final performance of a phase sequence by only looking at a few number of phases further down from the current location.

	Lookahead		
	1	2	3
% Performance	22.90	14.64	5.35

Table 4. Perf. of N-Lookahead Algorithm

5. Conclusions

In this paper we studied various properties of the optimization phase order space, and evaluated various heuristic search algorithms. Based on our observations regarding the search space properties, we further suggested and evaluated extensions to existing heuristic algorithms, and developed new heuristic methods. This study is also the first comparison of the performance delivered by the different heuristic algorithms with the optimal phase ordering. A study of this magnitude would have normally required several man-months to accomplish. However, the presence of the exhaustive phase order exploration data over a large number of functions meant that this study required no further compilation or simulation runs to determine the performance of each unique phase sequence during the heuristic algorithms.

We have a number of interesting conclusions from our detailed study: (1) Analysis of the phase order search space indicates that the space is highly discrete and very sparse. (2) The phase order space typically has a few local and global minima. More importantly, the sequence length of attempted phases defines the percentage of local and global minima in the search space. Larger sequence lengths increase the probability of finding a global minima, but can also increase the search time to find a good solution. Thus, it is important to find the correct sequence lengths to balance algorithm cost, and its ability to reach better performing solutions faster. (3) Due to the inherent difficulty in determining the ideal sequence length to use during any heuristic method, and the high probability of obtaining near-optimal performance from leaf function instances, we modified existing algorithms to concentrate on only leaf sequences and demonstrated that for many algorithms leaf sequences can deliver performance close to the best, and often times even better than that obtained by excessive increases in sequence lengths for the same algorithms. Moreover, this can be achieved at a fraction of the running cost of the original algorithm since the space of leaf function instances is only 4.38% of the total space of all function instances. (4) On comparing the performance and cost of different heuristic algorithms we find that simple techniques, such as local hill climbing allowed to run over multiple iterations, can often outperform more complex techniques such as genetic algorithms and lookahead schemes. The added complexity of simulated annealing, as compared to hill climbing, is found to not significantly affect the performance of the algorithm. Random searches and greedy search algorithms achieve decent performance, but not as good as the other heuristic approaches for the amount of effort expended. The unpredictable nature of phase interactions is responsible for the mediocre performance of the N-lookahead heuristic algorithm. (5) Interestingly, most of the heuristic algorithms we evaluated are able to achieve performance close to the best phase ordering performance in acceptable running times for all functions. Thus, in conclusion we find that differences in performance delivered by different heuristic approaches are not that significant when compared to the optimal phase ordering performance. Selection of the correct sequence length is important for algorithms that depend on it, but can be safely bypassed without any significant performance loss wherever possible by concentrating on leaf sequences.

6. Acknowledgments

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by NSF grants EIA-0072043, CCR-0208892, CCR-0312493, CCF-0444207, CNS-0072043, CNS-0305144, and CNS-0615085.

References

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

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

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

[4] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yun-

heung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, Compiler, and Tool for Embedded Systems*, pages 12–23. ACM Press, 2003.

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

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

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

[8] Prasad Kulkarni, David Whalley, Gary Tyson, and Jack Davidson. 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*, pages 83–92, New York, NY, USA, 2006. ACM Press.

[9] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*, pages 231–239, New York, NY, USA, 2004. ACM Press.

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

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

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

[13] Min Zhao, Bruce Childers, and Mary Lou Soffa. 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*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[14] Min Zhao, Bruce R. Childers, and Mary Lou Soffa. A model-based framework: An approach for profit-driven optimization. In *Proceedings of the international symposium on Code generation and optimization*, pages 317–327, Washington, DC, USA, 2005.

[15] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, 1993.

[16] Shun Long and Grigori Fursin. A heuristic search algorithm based on unified transformation framework. In *7th workshop on High Performance Scientific and Engineering Computing*, Norway, 2005. IEEE Computer Society.

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

[18] P.M.W. Knijnenburg, T. Kisuki, K. Gallivan, and M.F.P. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling. In *Proc. FDDO-3*, pages 31–40, 2000.

[19] Prasad Kulkarni, Steve Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, June 2004.

[20] Prasad Kulkarni, Steve Hines, David Whalley, Jason Hiser, Jack Davidson, and Douglas Jones. Fast and efficient searches for effective optimization-phase sequences. *ACM Trans. Archit. Code Optim.*, 2(2):165–198, 2005.

[21] K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Acme: Adaptive compilation made efficient. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 69–78, June 15-17 2005.

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

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

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

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

[26] Paul E. Black. Simulated annealing. Dictionary of Algorithms and Data Structures adopted by the U.S. National Institute of Standards and Technology, December 2004. http://www.nist.gov/dads/HTML/simulatedAnnealing.html.

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

[28] A. Nisbet. Genetic algorithm optimized parallelization. Workshop on Profile and Feedback Directed Compilation, 1998.

[29] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pages 77–90. ACM Press, 2003.