Finding Good Optimization Sequences Covering Program Space

SURESH PURINI and LAKSHYA JAIN, International Institute of Information Technology

The compiler optimizations we enable and the order in which we apply them on a program have a substantial impact on the program execution time. Compilers provide default optimization sequences which can give good program speedup. As the default sequences have to optimize programs with different characteristics, they embed in them multiple subsequences which can optimize different classes of programs. These multiple subsequences may falsely interact with each other and affect the potential program speedup achievable. Instead of searching for a single *universally* optimal sequence, we can construct a small set of *good sequences* such that for every program class there exists a near-optimal optimization sequence in the good sequences set. If we can construct such a good sequences set which covers all the program classes in the program space, then we can choose the best sequence for a program by trying all the sequences in the good sequences set. This approach completely circumvents the need to solve the program classification problem. Using a sequence set size of around 10 we got an average speedup up to 14% on PolyBench programs and up to 12% on MiBench programs. Our approach is quite different from either the iterative compilation or machine-learning-based prediction modeling techniques proposed in the literature so far. We use different training and test datasets for cross-validation as against the Leave-One-Out cross-validation technique.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Algorithms, Design, Experimentation, Languages, Performance

Additional Key Words and Phrases: Compiler optimizations, phase ordering problem, optimization sequence clustering, program clustering

ACM Reference Format:

Purini, S. and Jain, L. 2013. Finding good optimization sequences covering program space. ACM Trans. Architect. Code Optim. 9, 4, Article 56 (January 2013), 23 pages.
DOI = 10.1145/2400682.2400715 <http://doi.acm.org/10.1145/2400682.2400715>

1. INTRODUCTION

1.1. Motivation and Problem Definition

Modern optimizing compilers are organized as a sequence of three modules: *frontend*, *optimizer (middle end)*, and *backend* (see Figure 1). The frontend generates an *intermediate representation* (IR) of the source program after checking for its syntactic and semantic correctness. The optimizer component which constitutes the middle end of the compiler applies a series of transformations on the IR so as to optimize it with respect to parameters like speed, memory footprint, power, etc. The backend translates the optimized IR into target machine-code and during that process it applies machine dependent code optimizations [Torczon and Cooper 2011]. The optimizer component is organized as a sequence of *analysis* and *transformation* passes. Analysis passes analyze the program and collect necessary information required by the transformation passes. A *compiler optimization* typically consists of one or more analysis passes

Authors' addresses: S. Purini (corresponding author), L. Jain, International Institute of Information Technology, Hyderabad, India; email: suresh.purini@gmail.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/01-ART56 \$15.00

DOI 10.1145/2400682.2400715 <http://doi.acm.org/10.1145/2400682.2400715>

Fig. 1. Structure of a three phase compiler.

Fig. 2. Runtimes of the program dynprog from the PolyBench suite when optimized using different optimization sequences. X-axis is labeled with sequence numbers which are sorted according to the respective program runtimes.

followed by a transformation pass. Any sequence of compiler optimizations that can act on the program IR is called as an *optimization sequence*. The collection of all optimization sequences is called the *optimization sequence space* and is infinitely large. The runtime¹ of a program is a function of the optimization sequence applied on its IR. Figure 2 shows the runtime for the program dynprog from the PolyBench 3.0 benchmark suite [Polybench 2012] on a set of 290 optimization sequences in a sorted order. The methodology for choosing these 290 optimization sequences will be discussed in detail in Section 3.3. The impact of the optimization sequences on the program runtimes can also be observed by computing the variance in the program runtimes when optimized by bounded length sequences generated by a uniform random sampling of the optimization sequence space.

The optimization sequence space size grows exponentially as a function of sequence length. If there are k optimizations, then there are k^l optimization sequences of length l . We say that an optimization sequence is *optimal* for a program if it leads to smallest program runtime when compared to all other sequences. This definition is incomplete as the program runtime and its behavior could be a function of the input. So an optimal sequence for a program depends on the *program characteristics* as defined by the input to the program. However in this work we assume that a program has a canonical input distribution and the program characteristics with respect to that input distribution almost remains constant [Chen et al. 2010]. If this assumption is not valid, then the sequence chosen using a particular input may not be good when the program is run using inputs which induce different program characteristics. The problem of finding an optimal optimization sequence for a program from the infinitely large optimization

¹In this article, program runtime is the only performance parameter we consider.

sequence space is called as the *phase order search* problem. As the phase order search problem is a combinatorial search problem with no linearity or convexity properties, we cannot say whether a given sequence is optimal. All we can say is how well the sequence is performing with respect to a default compiler optimization sequence (like -O2). When we say that a sequence is good for a program, it only means that it is giving a noticeable performance improvement over default optimization sequences. A good sequence is not necessarily an optimal sequence.

1.2. Summary of Proposed Approach

Searching a good optimization sequence for a program in the optimization sequence space is like searching for a needle in a haystack. Instead of solving this problem on a *per program* basis, compiler writers construct good optimization sequences through experience and experimentation on benchmark programs. The constructed sequences are associated with default optimization options like -O1, -O2, and -O3². These sequences may be globally optimal with respect to the *program space*³, but are suboptimal for the individual programs. The simple reason being what is a good sequence for one program may not be good for another. So looking for an *universal good sequence* is a futile exercise. An alternate viable approach which we proposed in this article is, to build a set of *few good sequences*, so that for every *class of programs* there exists a good optimization sequence in the sequence set catering to that class. Then given a new program we can choose the best sequence by trying out all the sequences from the good sequences set. This approach completely bypasses the *program classification* problem. Using the LLVM compiler framework [Lattner and Adve 2004] we constructed a good sequences set. In Section 3, the approach for constructing the good sequences set which forms the crux of the proposed technique will be presented. Here we highlight the main contributions of this article.

- (1) We present a *downsampling technique* to reduce the infinitely large optimization sequence space to a small set (of size in the order of hundreds). During the downsampling process, the diversity of the optimization sequences catering to different program classes will be retained.
- (2) We provide a sequence extraction algorithm to construct a small good sequences set from the downsampled optimization sequence set, such that for every program class there exists at least one good sequence in the reduced set. The size of the good sequences set in our experiment is around 10.
- (3) We give a sequence similarity metric and a sequence clustering algorithm leading to an alternate approach for the good sequence set extraction from the downsampled sequence set.
- (4) We present a *program characterization* approach using optimization sequences as against using static program features or dynamic features like hardware performance counters. This alternate perspective gives us a handle to understand the effectiveness of optimization sequence prediction models and the relevant program features determining the parameters of those models.

Our technique is quite different from the iterative compilation and machine-learning-based prediction modeling methods proposed in the literature. In iterative compilation techniques, a heuristic search algorithm⁴ is used to prune the optimization sequence

²These sequences have to be reconstructed whenever the backend of a compiler is retargeted to a new processor architecture. This becomes a frequently recurring issue for compilers targeting embedded architectures.

³Program space consists of all syntactically and semantically valid programs.

⁴Genetic algorithms, simulated annealing, steepest descent methods are examples of heuristic search techniques.

Table I. List of 62 Machine-Independent Optimizations

List of Optimizations			
-adce	-always-inline	-argpromotion	-codegenprepare
-constmerge	-constprop	-correlated-propagation	-dce
-deadargelim	-die	-dse	-early-cse
-globaldce	-globalopt	-gvn	-indvars
-inline	-instcombine	-instsimplify	-internalize
-ipconstprop	-ipsccp	-jump-threading	-licm
-loop-deletion	-loop-idiom	-loop-instsimplify	-loop-reduce
-loop-rotate	-loop-simplify	-loop-unroll	-loop-unswitch
-loops	-lower-expect	-loweratomic	-lowerinvoke
-lowerswitch	-memcpyopt	-mergfunc	-mergereturn
-partial-inliner	-prune-eh	-reassociate	-scallrepl
-sccp	-simplify-libcalls	-simplifycfg	-sink
-tailcallelim	-targetlibinfo	-no-aa	-tbaa
-basicaa	-basicccg	-functionattrs	-scallrepl-ssa
-domtree	-lazy-value-info	-lcssa	-scalar-evolution
-memdep	-strip-dead-prototypes		

space to converge onto a good sequence. Even for very good heuristic techniques, the number of required program evaluations would be prohibitively large to use in nontrivial application programs. In the machine-learning-based approach, during an *offline training phase* the parameters of a chosen *prediction model* will be learnt by observing the runtimes of a set of sample programs⁵ when optimized using randomly selected sequences. Then the prediction model would be used to determine the set of optimizations that should be enabled on a given source program. A compiler-specific default ordering for the enabled optimizations would be used to optimize the program. The principal advantage of this approach is that the number of required program evaluations would come down to feasible limits. Applying machine learning techniques to directly predict the optimization sequences is not feasible because multi-label classification models like decision trees, neural networks, etc., are suitable when there are only few class labels. Thomson et al. [2009] proposed a *nearest cluster* approach to overcome this limitation. The set of available training programs are clustered using their feature vectors and for each cluster center the best optimization sequence is obtained using an iterative compilation technique. For a new program the sequence corresponding to the nearest cluster centroid is chosen as the best sequence. Park et al. [2011] proposed a technique using *tournament predictors* for the optimization sequence selection problem.

2. EXPERIMENTAL SETUP

We use the LLVM⁶ 3.0 compiler infrastructure [Lattner and Adve 2004] to validate the proposed approach. LLVM compiler infrastructure uses a *persistent IR*, and consists of a collection of compiler tools and reusable libraries. We can store the IR of a program on disk either in human-readable text format or machine-readable *bitcode* format. The C language frontend clang translates the source program into LLVM IR. The middle end applies an optimization sequence on the IR based on the optimization option (like -O1, -O2, -O3) chosen by the user. Then the backend applies machine-dependent optimizations on the optimized IR and finally generates the target assembly. The optimizations, register allocation, and other algorithms invoked by the backend are also dependent on the optimization level chosen by the user. Table I lists the

⁵These set of sample programs constitute the *training data*.

⁶LLVM is an acronym for *Low-Level Virtual Machine*.

Fig. 3. Program compilation setup.

machine-independent compiler optimizations invoked at -O2 level. Refer to LLVM [2012] for a description of the optimizations listed in the Table I. In LLVM, the optimization level -O2 almost always gives either better or equal performance as that of -O3. So in all our experiments the performance comparison is with respect to -O2.

LLVM compiler infrastructure provides two tools, `opt` and `llc`, which are useful in conducting our experiments. The `opt` program takes as input an optimization sequence on the command line and applies it to the LLVM IR which is stored in a bitcode format file. After applying the specified optimization sequence, the transformed IR is again written back to a file in bitcode format. `llc` translates an LLVM IR in a bitcode file into target assembly code. We can invoke the backend of the compiler at various optimization levels in isolation using `llc`. In this article we focus on the interactions between machine-independent optimizations acting on the LLVM IR. The source program is translated into unoptimized LLVM IR by first by compiling it with `clang` at -O0 level. Then `-scalarrepl` optimization is applied on the unoptimized IR. The scalar replacement optimization transforms the LLVM IR into its SSA form suitable for any other optimization to act upon. The `opt` tool applies the input optimization sequence⁷ on the IR and thereby generates the optimized IR. Finally `llc` transforms the LLVM IR generated by the optimizer into target assembly code using -O2, level optimization. For parametric optimizations like loop unrolling, we let the compiler choose the unroll factor according to its own internal heuristics. Figure 3 depicts the summary of the compilation process.

We used three benchmark suites in our experiments. The first benchmark suite consists of a collection of 61 programs from the LLVM test suite. We call this as the Microkernels benchmark suite. All the programs in the Microkernels suite are single-file short programs. We require the programs in this suite to have diverse characteristics and short running times so that iterative compilation methods can be applied on them to find near-optimal optimization sequences. So this suite contains sorting programs, programs involving floating point arithmetic, programs containing tail recursion, etc. We use this suite in the offline phase to prune the exponentially large optimization sequence space down to a *few good sequences*. We test the effectiveness of our approach on the MiBench [Guthaus et al. 2001] and the PolyBench [Polybench 2012] programs. We use the dataset one while running MiBench programs and the large dataset for the PolyBench programs. We would like to highlight here that ours is the first approach which uses completely different training and test datasets for cross-validation as against the widely used Leave-One-Out approach. All the experiments are carried out on an Intel Xeon Processor W3550 machine with 12GB RAM.

3. PROPOSED APPROACH

3.1. Motivation

The set of good sequences for a program is a function of its characteristics (and the target architecture). The key issue in machine-learning-based approaches is to identify

⁷There are no ordering constraints on the optimization passes in LLVM.

the relevant static and dynamic program features characterizing a program with respect to the optimization sequence selection problem and, an appropriate model correlating the selected program features and the optimization sequences. Although some compiler frameworks like Milepost GCC provide plugins to extract some static program features, exploring the static program feature space is hard as addition of a new static program feature to the existing set requires the source-code modification of either an appropriate plugin or the compiler itself. Instead we can use the hardware performance counters provided by modern processors to capture the dynamic behavior of a program with relative ease. There could be sampling errors though due to the large number of possible attributes to measure when compared with the available registers to accumulate the event counts. For accurate measurements we have to run a program multiple times to reduce the sampling errors. However the fundamental question what static, dynamic and possibly any other type of program features completely characterize a program with respect to the optimization sequence selection problem still remains largely open. Iterative compilation techniques which do not use program features per se require prohibitively large number of program runs. The well-known approach taken by all the major compilers is to precompute good sequences and associate them with compilation options like -O1, -O2, and -O3. These precomputed sequences are intended to be near optimal for diverse program classes. In order to accomplish this, multiple subsequences catering to different program classes are embedded into a single run universally optimal sequence. However, the subsequences within the bigger sequence may end up having false interactions and thereby adversely impact the program runtime. In essence a single size fits all approach does not work effectively.

An alternate approach is to identify the program classes and for each class we construct a good optimization sequence either manually or by using an iterative compilation technique like genetic algorithms. Then we can use a program classification algorithm to identify the class label for a new program and use the associated optimization sequence for its compilation. The difficulty here, as discussed earlier, is in identifying the program features and the program classification model. Another problem in constructing good sequences for each of the program classes is in identifying the criteria for program class representative selection in order to construct a good sequence for the respective class using an iterative compilation technique. The proposed approach entirely bypasses these hurdles by starting with the optimization sequences and then identifying the program classes they cover. In effect we completely reverse the sequence selection technique using program classification.

3.2. Intuition

We want to construct a set of sequences such that for every program class there exists at least one good sequence in the set. We call this set as the *good sequences set*. If the size of the good sequences set is small, then for a given program we can try all the sequences in the set and choose the sequence which gives the best performance. We avoid the program classification problem by exhaustively trying all the sequences. This technique will be effective if the good sequences set covers all the program classes and will be practical only if the size of the good sequences set is small. So the crux of the problem is to construct a tight but rich good sequences set.

We call the infinite collection of programs as *program space* drawing an analogy from *vector spaces* of linear algebra. We call the set of programs for which an optimization sequence gives good performance as the *hyperplane* passing through the respective programs in the program space and it is associated with the respective sequence. The objective now is to select as few hyperplanes as possible so that the entire program space is covered. The sequences corresponding to the chosen hyperplanes constitute the good sequences set.

We can hypothetically envisage an algorithm wherein we choose a hyperplane which covers maximum number of programs in the program space. Then we choose another hyperplane which covers maximum number of programs not covered by the first hyperplane. We keep iterating this step until the entire program space is covered. If we assume that the program space is finite (which it is not) and if in each iteration a hyperplane covers a constant fraction of the remaining programs, then the algorithm converges in logarithmic steps yielding a small good sequences set.

We cannot implement the above algorithm unless we find a way to handle the infinite size of the program space. This issue is addressed by choosing a finite collection of benchmark programs which is representative of the program class distribution in the program space. We choose the Microkernels benchmark suite as a representative sample of the program space. It has to be noted that how well the Microkernels benchmark suite represents the actual program class distribution needs to be studied. Assuming that the Microkernels benchmark suite approximates the actual program class distribution, we use that suite to downsample the infinitely large optimization sequence space to a small set of good sequences. This small set of good sequences covers all the program classes. The size of this set turned out to be 290 in our experiments. However exhaustively trying out all these sequences is not practical, so we reduce this sequence set further down to 10 by using two techniques. We call the first technique as the Best-10 approach and the second technique as the Sequence Clustering approach. In the next subsection we present the optimization space downsampling approach using the Microkernels benchmark suite and in the later subsections we present the Best-10 and Sequence Clustering algorithms.

3.3. Optimization Sequence Space Downsampling

Optimization sequence space consists of all optimization sequences and is a countable infinite set. Corresponding to every program class we could think of a subspace consisting of sequences which are good for that program class. We assume that all the programs can be coarsely grouped into a small number of finite classes and hence there are small number of good sequence subspaces catering to these program classes. If we have a diverse collection of programs that cover all the program classes, then by finding good sequences for each of these programs we get an optimization sequence set which covers all the optimization sequence subspaces. This is crux of the our downsampling technique.

As mentioned earlier we choose the Microkernels benchmark suite for optimization sequence space downsampling. We used six different iterative compilation techniques to construct good sequences corresponding to each of the Microkernel programs. Since there are 61 Microkernel programs, a good sequence set of size 366 is generated. Due to the redundancies in the good sequence set, the final downsampled optimization sequence set size came down to 290. The following are the three iterative compilation techniques used in the downsampling procedure.

- (1) *Genetic Algorithm with a Rank Selector*. We used a genetic algorithm with the following parametric choice: Chromosome size = 50, population size = 60, number of generations = 100, mutation rate = 0.02, crossover rate = 0.9. If the standard deviation of the fitness score for a population is less than 0.01 or the best fitness score does not improve for 3 consecutive generations, the algorithm will terminate.
- (2) *Genetic Algorithm with a Tournament Selector*. This version of the genetic algorithm is exactly same as the above except that we used a Tournament Selector instead of a Rank Selector.
- (3) *Uniform Random Search*. In this algorithm, we generate 500 sequences of length 50 each by uniform random sampling and pick the best sequence.

Fig. 4. Program compilation setup with optimization sequence updates.

The other 3 iterative compilation techniques are exactly the same except that we update an optimization sequence before it is applied on the LLVM IR. A quick look at the LLVM manual [LLVM 2012] suggests what optimizations could precede and succeed an optimization for its effectiveness. For example the manual suggests that after `-constprop` (constant propagation) pass, it is a good idea to apply `-die` (dead instruction elimination) pass. So whenever `-constprop` pass appears in an optimization sequence, we modify the sequence by inserting `-die` pass immediately after that. By modifying the optimization sequences in this manner, we incorporate human knowledge and experience in an iterative compilation technique, so that the algorithm can build upon it to converge to an optimal sequence faster. Figure 4 shows the modified program compilation set.

While applying an iterative compilation technique there are optimization sequences which crash the compiler. We simply ignore those sequences. It is also possible that some of the sequences may generate incorrect code due to possible bugs in the compiler. However we do not check for correctness in the downsampling phase. This is due to lack of proper test suite infrastructure for the Microkernels benchmarks which could be developed in due time. But the correctness of the sequences on the test programs from the MiBench and PolyBench suites is verified.

Each of the sequences present in the downsampled optimization sequence set contains optimizations not contributing to the program speedup or have a negative impact on the program speed up due to false interactions with other optimizations. We eliminate these unnecessary optimizations in a sequence by applying the *Sequence Reduction* algorithm (see Algorithm 1). The *Sequence Reduction* algorithm keeps dropping optimizations from a sequence as long as the runtime of the program does not increase. The algorithm terminates when dropping any optimization from the remaining sequence increases the program runtime. The length of the sequences in the downsampled optimization sequence set reduced by almost 80% after applying the *Sequence Reduction* algorithm.

3.4. Best-10 Approach

After reducing the sequences in the downsampled sequence set, we construct a performance summary matrix PM . The rows of the matrix PM are labeled with the Microkernel programs and the columns with the sequences from the downsampled sequence set. If a row is labeled with a program p and a column with a sequence s , then the matrix entry $PM[p, s]$ contains the runtime of the program p when optimized using the sequence s . We then apply a *Sequence Extraction* algorithm on the performance summary matrix to extract the Best-10 sequences covering the Microkernels benchmark suite. The *Sequence Extraction* algorithm identifies the sequence which improves the performance of maximal number of Microkernel programs over `-O2`. In other words, if we put a cross mark over the matrix entries which are strictly less

ALGORITHM 1: Sequence Reduction Algorithm**Input:** Program and an optimization sequence**Output:** Reduced optimization sequencebestseq $\leftarrow$ optseq ;besttime $\leftarrow$ program runtime using bestseq ;change $\leftarrow$ True ;**while** change = True **do** change $\leftarrow$ False; bestseqlen $\leftarrow$ length of bestseq ; **for** $i=1$ **to** bestseqlen **do** currseq $\leftarrow$ bestseq - { i^{th} optimization of bestseq } ; currseqtime $\leftarrow$ program runtime using currseq ; **if** currseqtime $\leq$ besttime **then** besttime $\leftarrow$ currseqtime ; bestseq $\leftarrow$ currseq ; change $\leftarrow$ True ;

break from the for loop ;

end **end****end**

return bestseq ;

/* return the reduced sequence */

than the corresponding program's -O2 runtime, then the algorithm picks a sequence corresponding to a column with maximal number of cross marks. Ties are resolved by picking the sequence which gives maximum average speedup. This sequence is included in the good sequences set and the corresponding column will be eliminated from the matrix. The rows corresponding to the programs which got covered by this sequence will also be eliminated. Now the same procedure is applied on the reduced matrix until either the rows of the matrix are completely exhausted or a predetermined bound on the good sequences set size is reached. The construction of the good sequences set is a one-time offline procedure. After that given a new program we can try all the sequences from the good sequences set and choose the sequence which gives the best speedup. Refer Algorithm 2 for the *Sequence Extraction* algorithm pseudocode.

3.4.1. Best-12 – A Modified Best-10 Approach. Best-12 is a variant of the Best-10 technique. In this approach, we apply the Best-10 algorithm and out of the ten extracted sequences we pick the top six sequences. Then we remove these six sequences from the downsampled sequence set and repeat the Best-10 algorithm on the rest of the downsampled sequence set. Out of the new best ten sequences we pick the top six sequences again. The top six sequences from the two phases put together gives the Best-12 sequences. This approach uncovers the good sequences in the second application of Best-10 algorithm that are eliminated by the top sequences during the first application of Best-10 algorithm. This approach which is slightly counterintuitive can be partially explained by saying that it is better to have more optimization sequence samples from dominant program classes than to have optimization sequences corresponding to small program classes.

3.5. Sequence Clustering Approach

The downsampled optimization sequence set is a union of optimization sequence subspaces corresponding to various program classes. If we can identify these optimization sequence subspaces, then we can pick representative sequences from each of the subspaces to construct the final reduced set of good sequences. In this section

ALGORITHM 2: Best-10 Sequence Extraction Algorithm**Input:** List of 290 sequences, list of Microkernel programs, performance summary matrix (PM).**Output:** Best-10 sequences.seq290 $\leftarrow$ List of all 290 sequences ;prgms $\leftarrow$ List of all Microkernel programs ;**for** each seq in seq290 **do** **for** each prgm in prgms **do** **if** $PM[prgm, seq] < clang -O2$ runtime of prgm **then**

Dict[seq].append([prgm, speedupoverO2]) ;

end **end****end**Best10 $\leftarrow \{ \}$;**for** $i = 1$ to 10 **do**

/* Pick the sequence for which the length of the list Dict[s] is maximum. If there are ties, pick the one with maximum total speedup among the tied sequences. */

S = getmaximum(Dict) ;

Best10.append(S) ;

/* Delete all programs that are covered by S */

for seq in Dict.keys() **do** **for** prgm in Dict[S] **do** **if** $prgm \in Dict[seq]$ **then**

Dict[seq].delete(prgm) ;

end **end** **end****end**

we propose a sequence clustering algorithm to identify the optimization sequence subspaces and a way to pick representative sequences from each of the subspaces.

Let m be the number of programs in the Microkernels benchmark suite. Corresponding to a sequence s we define a *sequence vector* V_s such that the i^{th} entry in the vector V_s is 1 if the sequence s improves the performance (over -O2) of the i^{th} program in the Microkernel benchmark suite. Otherwise the i^{th} entry is equal to 0. We construct a sequence similarity matrix SS such that the rows and the columns of the matrix are labeled with the sequences from the downsampled optimization set. If p and q are two sequences, then the matrix entry $SS[p, q]$ contains the Euclidean distance between the vectors V_p and V_q . We then apply the *Sequence Clustering* algorithm (refer to Algorithm 3) to partition the downsampled sequence space into 10 clusters. The sequence clustering algorithm is minor variant of the standard k -means clustering algorithm. The algorithm starts with randomly initializing 10 sequence vectors as the cluster centers. The rest of the sequence vectors are assigned to the closest cluster center. Now the mean vector for each cluster is computed. This mean vector will not correspond to any of the sequence vectors usually. So the closest sequence vector to the mean vector is made cluster center in the next iteration. This process repeats until the clusters converge or a predefined number of iterations are reached. It has to be noted that because of the way we shift the cluster center, the algorithm may not converge like the usual k -means algorithm.

After the sequence clusters are constructed, we pick an arbitrary cluster and from that cluster pick a sequence which covers maximum number of Microkernel benchmark programs when compared to the rest of the sequences in the cluster. Note that

ALGORITHM 3: Sequence Clustering Algorithm**Input:** Sequence Similarly Matrix (SS).**Output:** Ten clusters.seq290 $\leftarrow$ List of all 290 sequences ;prgms $\leftarrow$ List of all Microkernel programs ;Pick 10 sequences $s_1, \dots, s_{10}$ uniformly at random. ;centroid = $[s_1, \dots, s_{10}]$;

/* List of cluster centers */

oldcentroid = $[-1, \dots, -1]$;**while** oldcentroid $\neq$ centroid **do**

oldcentroid = centroid ;

for each s **in** seq290 **do**

min = -1 ;

for j **in** centroid **do** **if** min > SS[s][j] **then** min = SS[s][j] ; clr = j ; **end** clusters[clr].append(s) **end** **end** **for each cluster** **in** clusters **do**

mean = mean of all the sequence vectors in this cluster;

center = the point in the cluster that is closest to mean ;

 update centroid[i] to center ; **end****end**

the sequence selected using this approach may not be the cluster centroid. We then remove the set of programs covered from the Microkernels suite and repeat the same process by choosing yet another cluster arbitrarily until all clusters are exhausted. The ten sequences thus selected are not deterministic as the initial cluster centers in the cluster selection algorithm are chosen at random. The quality of the clusters constructed and hence the quality of the final selected sequences depends on the initial choice of cluster centers. So we ran the Sequence Clustering algorithm 100 times and computed the corresponding sequence sets each of size 10 sequences. Then among those hundred sequence sets we pick the sequence set which covered the maximum number of Microkernel programs. It is important to note that we did not use the test benchmark programs to select the final set of 10 sequences.

4. EXPERIMENTAL RESULTS

As discussed earlier we used the LLVM compiler infrastructure to test the proposed approach. Microkernels benchmark suite is used to construct the good sequences set and the resulting good sequences are tested on the MiBench [Guthaus et al. 2001] and Polybench 3.0 programs. We used the dataset one for the MiBench programs and the large dataset for the PolyBench programs. All the experiments are carried out on an Intel Xeon Processor W3550 machine with 12GB RAM and Linux operating system running on it. The workload on the system is kept to bare minimum to reduce the variance in measured program runtimes. We also checked the variance in the program runtimes by running the programs for multiple times and found the variance to be negligible. The performance of the good sequences is compared with the LLVM -O2 option instead of -O3 as it gives better performance than -O3 on most programs.

Table II. LLVM -O2 Sequence

[targetlibinfo, -targetdata, -no-aa, -tbaa, -basicaa, -globalopt, -ipsccp, -deadargelim, -instcombine, -simplifycfg, -basiccg, -prune-eh, -inline, -functionattrs, -scalarrpl-ssa, -domtree, -early-cse, -simplify-libcalls, -lazy-value-info, -jump-threading, -correlated-propagation, -simplifycfg, -instcombine, -tailcallelim, -simplifycfg, -reassociate, -domtree, -loops, -loop-simplify, -lcssa, -loop-rotate, -licm, -lcssa, -loop-unswitch, -instcombine, -scalar-evolution, -loop-simplify, -lcssa, -indvars, -loop-idiom, -loop-deletion, -loop-unroll, -memdep, -gvn, -memdep, -memcpyopt, -sccp, -instcombine, -lazy-value-info, -jump-threading, -correlated-propagation, -domtree, -memdep, -dse, -adce, -simplifycfg, -instcombine, -strip-dead-prototypes, -constmerge, -preverify, -domtree, -verify]

Table III. Best-10 Sequences and Their Respective Lengths

S. No	Best-10 Sequences	Length
1	[-instcombine, -simplifycfg, -gvn, -indvars, -loop-rotate, -unroll-allow-partial, -loop-unroll, -loop-rotate, -loop-unroll, -gvn, -inline, -early-cse, -basicaa, -reassociate, -instcombine]	15
2	[-simplify-libcalls, -simplifycfg, -unroll-allow-partial, -internalize, -gvn, -instcombine, -inline, -globalopt, -scalarrpl, -loop-rotate, -loop-unroll, -loop-instsimplify]	12
3	[-gvn, -loop-rotate, -internalize, -inline, -licm, -tailcallelim, -instcombine, -basicaa, -indvars]	9
4	[-codegenprepare, -loop-rotate, -loop-idiom, -loop-deletion, -gvn]	5
5	[-loop-rotate, -inline, -simplify-libcalls, -basicaa, -licm, -constmerge, -indvars, -unroll-allow-partial, -loop-unroll, -reassociate, -gvn]	11
6	[-simplifycfg, -tailcallelim, -early-cse, -loop-rotate, -internalize, -partial-inliner, -instcombine, -inline, -ipsccp]	9
7	[-internalize, -inline, -globalopt, -early-cse, -licm, -basicaa, -indvars, -loop-reduce]	8
8	[-codegenprepare, -gvn, -tailcallelim, -reassociate, -basicaa, -inline, -loop-rotate, -indvars, -reassociate]	9
9	[-simplifycfg, -tailcallelim, -instcombine, -early-cse]	4
10	[-early-cse, -simplifycfg, -gvn, -inline, -loop-reduce, -licm, -instcombine, -gvn, -reassociate]	9
Optimizations not figuring in any of the Best-10 Sequences		
-adce, -always-inline, -argpromotion, -constprop, -correlated-propagation, -dce, -deadargelim, -die, -dse, -globaldce, -instsimplify, -ipconstprop, -jump-threading, -loop-simplify, -loop-unswitch, -loops, -lower-expect, -loweratomic, -lowerinvoke, -lowerswitch, -memcpyopt, -mergefunc, -mergereturn, -prune-eh, -sccp, -sink, -targetlibinfo, -no-aa, -tbaa, -basiccg, -functionattrs, -scalarrpl-ssa, -domtree, -lazy-value-info, -lcssa, -scalar-evolution, -memdep, -strip-dead-prototypes		

4.1. Sequence Analysis

Table II gives the optimization sequence corresponding to -O2. Tables III, IV, and V give the good sequence sets obtained by using the Best-10, Best-12, and Cluster-10 approaches respectively. These tables also give the optimizations not occurring in any of the the sequences in the good sequences set. Table VI gives the frequency with which various optimizations occurred in the All-290 and Best-10 sequence sets. The frequency tables corresponding to Best-12 and Cluster-10 sequences look similar. Table VI statistically explains why certain optimizations are not figuring in any of the good sequences. However to give a qualitative reasoning requires an understanding of the optimizations, their implementation in the compiler and for what kind of programs those optimizations are targeted. For example -prune-eh optimization may be valid only for C++ programs whereas all the Microkernel benchmark programs are C programs.

Table IV. Best-12 Sequences and Their Respective Lengths

S. No	Best12 Sequences	Length
1	[-instcombine, -simplifycfg, -gvn, -indvars, -loop-rotate, -unroll-allow-partial, -loop-unroll, -loop-rotate, -loop-unroll, -gvn, -inline, -early-cse, -basicaa, -reassociate, -instcombine]	15
2	[-simplify-libcalls, -simplifycfg, -unroll-allow-partial, -internalize, -gvn, -instcombine, -inline, -globalopt, -scalarrepl, -loop-rotate, -loop-unroll, -loop-instsimplify]	12
3	[-gvn, -loop-rotate, -internalize, -inline, -licm, -tailcallelim, -instcombine, -basicaa, -indvars]	9
4	[-codegenprepare, -loop-rotate, -loop-idiom, -loop-deletion, -gvn]	5
5	[-loop-rotate, -inline, -simplify-libcalls, -basicaa, -licm, -constmerge, -indvars, -unroll-allow-partial, -loop-unroll, -reassociate, -gvn]	11
6	[-simplifycfg, -tailcallelim, -early-cse, -loop-rotate, -internalize, -partial-inliner, -instcombine, -inline, -ipsccp]	9
7	[-internalize, -inline, -globalopt, -early-cse, -licm, -basicaa, -indvars, -loop-reduce]	8
8	[-loop-rotate, -internalize, -instcombine, -globalopt, -gvn, -inline, -simplifycfg, -scalarrepl-ssa, -loop-deletion, -inline, -adce, -loop-unroll, -globalopt, -basicaa, -licm]	15
9	[-codegenprepare, -instcombine, -loop-rotate, -loop-idiom, -loop-deletion, -gvn, -sink]	7
10	[-loop-reduce, -inline, -loop-unroll, -indvars, -loop-rotate, -simplify-libcalls, -early-cse]	7
11	[-inline, -loop-rotate, -basicaa, -gvn, -loop-unroll, -simplify-libcalls, -licm]	7
12	[-instcombine, -simplifycfg]	2
Optimizations not figuring in any of the Best-12 Sequences		
-always-inline, -argpromotion, -constprop, -correlated-propagation, -dce, -deadargelim, -die, -dse, -globaldce, -instsimplify, -ipconstprop, -jump-threading, -loop-simplify, -loop-unswitch, -loops, -lower-expect, -loweratomic, -lowerinvoke, -lowerswitch, -memcpypopt, -mergfunc, -mergereturn, -prune-eh, -sccp, -targetlibinfo, -no-aa, -tbaa, -basiccg, -functionattrs, -domtree, -lazy-value-info, -lcssa, -scalar-evolution, -memdep, -strip-dead-prototypes.		

4.2. Performance Analysis

Program speedup and percentage improvement are calculated as follows.

$$Speedup = \frac{O2_{runtime}}{new_{runtime}}$$

$$Percentage\ Improvement = (Speedup - 1) * 100$$

Tables VII and VIII give the speedup of the MiBench and PolyBench programs using Genetic, All290, Best-10, Best-12 and Cluster-10 approaches⁸. Genetic algorithm being an iterative compilation technique applied on a per program basis gives an approximate upper bound on the achievable speedup. The column labeled All290 gives the speedup using all the sequences in the downsampled sequence set. All290 column gives the maximum speedup possible for any sequence extraction algorithm using the downsampled sequence set. The speedup due to All290 does not indicate the best possible speedup achievable as we could be missing the optimal sequence for a program completely from the downsampled sequence set. However our experimental results show that All290 performs on par with the genetic algorithm and in some cases even

⁸Due to space constraints we are not able to represent all the data graphically. However bar graphs comparing the performance of Cluster-10 approach with All290 which is doing better than genetic are provided.

Table V. Cluster-10 Sequences and Their Respective Lengths

S. No	Cluster-10 Sequences	Length
1	[-inline, -basicaa, -loop-rotate, -unroll-allow-partial, -loop-unroll, -jump-threading, -gvn, -licm]	8
2	[-loop-rotate, -internalize, -instcombine, -globalopt, -gvn, -inline, -simplifycfg, -scalarrepl-ssa, -loop-deletion, -inline, -adce, -loop-unroll, -globalopt, -basicaa, -licm]	15
3	[-early-cse, -simplifycfg, -tailcallelim, -reassociate, -lower-switch, -reassociate]	6
4	[-instcombine, -loop-rotate, -inline, -simplify-libcalls, -basicaa, -functionattrs, -loop-unswitch, -indvars, -loop-unroll, -gvn]	10
5	[-instcombine, -indvars, -simplify-libcalls, -internalize, -simplifycfg, -reassociate, -ipconstprop, -instcombine, -reassociate, -loop-rotate, -unroll-allow-partial, -loop-unroll]	12
6	[-internalize, -inline, -globalopt, -early-cse, -licm, -basicaa, -indvars, -loop-reduce]	8
7	[-simplifycfg, -inline, -internalize, -instcombine, -simplify-libcalls, -indvars, -inline, -loop-reduce, -ipscpp, -licm, -simplifycfg]	11
8	[-codegenprepare, -instcombine, -loop-rotate, -loop-idiom, -loop-deletion, -gvn, -sink]	7
9	[-simplifycfg, -tailcallelim, -jump-threading, -loop-rotate]	4
10	[-simplifycfg, -tailcallelim, -instcombine, -early-cse]	4
Optimizations not figuring in any of the Cluster-10 Sequences		
-always-inline, -argpromotion, -constmerge, -constprop, -correlated-propagation, -dce, -deadargelim, -die, -dse, -globaldce, -instsimplify, -loop-instsimplify, -loop-simplify, -loops, -lower-expect, -loweratomic, -lowerinvoke, -memcpyopt, -mergefunc, -mergereturn, -partial-inliner, -prune-eh, -scalarrepl, -sccp, -targetlibinfo, -no-aa, -tbaa, -basiccg, -domtree, -lazy-value-info, -lcssa, -scalar-evolution, -memdep, -strip-dead-prototypes		

does better than genetic. We can say that the downsampled sequence set contains rich and diverse sequences from this observation.

The MiBench suite consists of 17 unique programs. Some programs like `automotive_susan` call different functions based on the command line option being passed to the program. For example the options `-e`, `-c` and `-s` respectively invoke the functions `susan_edges`, `susan_corners` and `susan_smoothing`. The program pretty much spends all the time in one of these functions. The program versions as defined by the command line options are called uniquely as `automotive_susan_e`, `automotive_susan_c` and `automotive_susan_s` respectively. Each of these program versions may have a different optimal optimization sequence. Whereas the Table VII shows the speedup of the programs by treating each of the program versions as unique programs, Figure 5 shows the speedup of the programs by averaging the speedup of different program versions. The final average speedup shown in the graph is the average of averages. A maximum improvement of around 38% has been obtained for the programs `automotive_susan_s` and `security_sha`. Cluster-10 sequences are only able to give 23% improvement for `security_sha`. There is a minor loss in performance for the programs `network_patricia` and `consumer_tiffmedian`. For these programs there are no good sequences in the downsampled sequence set. For the program `office_stringsearch1`, the sequence `[-functionattrs, -loop-rotate, -licm, -basicaa]` in the All290 sequences optimizes it to run in 0.1 seconds as against the -O2 runtime of 3.1 seconds. However this sequence does not figure out in any of the Best-10, Best-12, and Cluster-10 sequence sets. The order in which the four optimizations occur in the aforementioned sequence is so important that any changes to it increases the runtime to 3 seconds. Further the optimization `-functionattrs` occurs

Table VI. Frequency with which Various Optimizations Occurred in All-290 and Best-10 Sequence Sets

Optimizations	All-290	Best-10
-loop-rotate	193	8
-inline	154	8
-indvars	151	5
-gvn	136	9
-instcombine	122	7
-basicaa	114	5
-licm	102	4
-internalize	98	4
-simplifycfg	72	5
-early-cse	61	5
-jump-threading	60	0
-simplify-libcalls	56	2
-reassociate	55	5
-globalopt	54	2
-loop-unroll	51	4
-loop-reduce	45	2
-tailcallelim	38	4
-loop-deletion	29	1
-ipsccp	24	1
-codegenprepare	17	2
-loop-idiom	17	1
-partial-inliner	15	1
-scalarrepl	15	1
-scalarrepl-ssa	15	0
-sink	14	0
-functionattrs	13	0
-loop-unswitch	12	0
-deadargelim	12	0
-sccp	11	0
-correlated-propagation	11	0
-dse	11	0

Optimizations	All-290	Best-10
-globaldce	10	0
-adce	10	0
-ipconstprop	8	0
-lowerswitch	7	0
-constmerge	7	1
-argpromotion	5	0
-loop-instsimplify	4	1
-lcssa	4	0
-basicccg	3	0
-dce	3	0
-strip-dead-prototypes	3	0
-instsimplify	2	0
-always-inline	2	0
-memcpyopt	1	0
-tbaa	1	0
-loop-simplify	1	0
-mergereturn	0	0
-lower-expect	0	0
-memdep	0	0
-lazy-value-info	0	0
-die	0	0
-constprop	0	0
-targetlibinfo	0	0
-mergefunc	0	0
-prune-eh	0	0
-scalar-evolution	0	0
-no-aa	0	0
-lowerinvoke	0	0
-loops	0	0
-loweratomic	0	0
-domtree	0	0

only in 13 of the 290 sequences. It is not clear whether we will be able to predict such sequences using machine-learning-based prediction models. Also due to this program All290 shows a higher average improvement when compared with other approaches.

Table VIII gives the speedup of the PolyBench programs using Genetic, All290, Best-10, Best-12, Cluster-10 sequences. Figures 6 and 7 show the performance comparison using Cluster-10 and All-290 sequences. The speedup for all the programs (except for `reg.detect`) approaches All290. Ten out of thirty programs show at least 15% improvement. The rest of the programs show minor or no improvement at all. However in all the cases the performance of the proposed approaches is almost the same as that of genetic. A maximum speedup of 75% is obtained for the program `dynprog`.

Table VII. Speedup Obtained on the MiBench Programs using Genetic, All290, Best-10, Best-12 and Cluster-10 Sequences

Program	Genetic	All290	Best-10	Best-12	Cluster-10
telecom_CRC32	1.01	1.01	1.01	1.01	1.01
automotive_qsort1	1.01	1.03	1.02	1.03	1.02
telecom_adpcm.c	1.17	1.16	1.15	1.02	1.16
automotive_susan_e	1.16	1.32	1.3	1.3	1.32
automotive_susan_c	1.2	1.32	1.27	1.27	1.3
telecom_adpcm.d	1.27	1.25	1.23	1.23	1.19
consumer_tiff2rgba	1.22	1.2	1.11	1.13	1.15
security_rijndael.d	1.03	1.02	1.03	1.02	1.01
security_rijndael.e	1.04	1.03	1.03	1.03	1
automotive_susan.s	1.23	1.4	1.37	1.38	1.38
office_stringsearch1	1	2	0.83	0.83	0.99
consumer_lame	1.01	1.01	0.97	1.01	1.01
consumer_tiffdither	1.12	1.09	1.07	1.08	1.06
network_dijkstra	1.03	1.11	1.03	1.05	1.05
telecom_gsm	1.17	1.17	1.15	1.15	1.14
automotive_bitcount	1.14	1.11	1.02	1.06	1.06
bzip2e	1.15	1.14	1.11	1.1	1.1
bzip2d	1	1.16	1.01	1.16	1.14
network_patricia	0.98	0.99	0.95	0.96	0.95
consumer_tiffmedian	1.06	1.02	0.94	1	0.9
security_sha	1.51	1.39	1.38	1.39	1.23
consumer_jpeg.d	1.02	1.01	1.01	1	1.01
consumer_jpeg.c	1.1	1.11	1.04	1.07	1.11
consumer_tiff2bw	1.22	1.14	1.1	1.1	1.12
security_blowfish.e	1.01	1.05	1.05	1.05	1.05
security_blowfish.d	1.05	1.09	1.09	1.09	1.09
Performance Results Summary					
API	11.2	16.7	8.73	9.7	9.8
APIE	12.7	17.4	11.7	12.4	12.3
NPS	23	25	22	22	22

API stands for average percentage improvement, APIE stands for average percentage improvement excluding the programs showing no speed up and NPS stands for the number of programs achieving speedup.

4.3. False Interactions between Optimizations in the LLVM -O2 Sequence

We applied the *Sequence Reduction Algorithm 1* on the -O2 sequence while optimizing the Microkernels programs in order to understand the impact of the optimizations on the program runtime and interactions between them. Table IX shows the -O2 runtimes for the Microkernels program and the runtimes after the sequence reduction. The maximum sequence length after reduction is 10 which is substantially small when compared to the original -O2 sequence length of 62. Further 16 out of 61 Microkernel programs show at least 10% improvement over their original -O2 runtimes and 8 out of those 16 programs show at least 20% improvement.

Table VIII. Speedup Obtained on the PolyBench Programs using Genetic, All290, Best-10, Best-12 and Cluster-10 Sequences

Program	Genetic	All290	Best-10	Best-12	Cluster-10
bicg	1.22	1.3	1.23	1.28	1.3
syrk	1.06	1.08	1.08	1.08	1.07
jacobi-1d-imper	1	1.05	1	1	1
trmm	1.05	1.05	1	1	1.01
syr2k	0.84	1.02	1.02	1.01	1.01
cholesky	1.02	1.02	1.01	1.01	1.01
fdtd-apml	1.01	1.03	1.03	1.01	1.01
3mm	1	1.02	1.02	1.01	1.01
lu	1	1	0.99	1	1
doitgen	1.04	1.07	1.03	1.02	1.02
seidel-2d	1.02	1.01	1	1	1
gemm	1	1.02	1.01	1.01	1.01
adi	0.97	1.01	1	1.01	1.01
gesummv	1.22	1.26	1.26	1.26	1.26
floyd-warshall	0.97	1.03	0.99	1	1
ludcmp	1.01	1.03	1.01	1.02	1.02
covariance	1	1.02	1.01	1.02	1.02
symm	1.04	1.05	1.03	1.04	1.04
atax	1.28	1.38	1.38	1.38	1.3
jacobi-2d-imper	1.24	1.27	1.21	1.24	1.27
trisolv	1.32	1.44	1.38	1.35	1.35
2mm	0.99	1.02	1.01	1.01	1.01
mvt	1.21	1.29	1.29	1.29	1.26
gemver	1.16	1.26	1.24	1.24	1.24
fdtd-2d	1.12	1.15	1.12	1.15	1.15
reg_detect	1.2	1.2	0.86	0.86	0.86
correlation	1	1.01	1	1.01	1.01
gramschmidt	1	1.02	1.02	1.01	1.01
dynprog	1.82	1.86	1.74	1.75	1.75
durbin	1.22	1.2	1.2	1.2	1.2
Performance Results Summary					
API	10.87	13.9	11.1	11.37	11.17
APIE	17.15	14.37	14.5	14.2	13.4
NPS	19	29	23	24	25

API stands for average percentage improvement, APIE stands for average percentage improvement excluding the programs showing no speed up and NPS stands for the number of programs achieving speedup.

5. RELATED WORK

Touati and Barthou [2006] showed that the phase ordering problem in its full generality is *undecidable*. They also showed that given an optimization sequence consisting of parametric optimizations, finding the optimal parameters for individual optimizations is also an *undecidable* problem. They proposed *decidable* variants for

Fig. 5. Performance comparison of the Cluster10 approach with All290 on MiBench programs.

both these problems. Kulkarni et al. [2006] proposed an exhaustive search strategy to find optimal compilation sequences for each of the functions in a program. Multiple optimization sequences can lead to the same target code. They use this idea to prevent the combinatorial explosion of the total number of sequences to be tested. Although this approach gives interesting information about optimization sequence space like number of local minima, etc., it may not be a practical approach. Pan and Eigenmann [2006] proposed a search algorithm called *combined elimination* which eliminates weakly interacting optimizations having a negative impact on the program in a single iteration. This allows the algorithm converge to a near-optimal solution fast. Cooper, Subramanian and Torczon [Cooper et al. 2002] performed experiments to characterize the optimization sequence space and suggested that the optimal solutions are rare with frequent local minima in the space. So simple hill-climbing search strategies may not work. However the characteristics of the optimization state space could vary a lot from compiler to compiler. Cooper, Schielke and Subramanian [Cooper et al. 1999] used a genetic-algorithm-based approach to find optimization sequences which reduce code size. Generating executables with small memory footprint is important while compiling embedded applications to be ported on devices with small flash memory. Triantafyllis et al. [2003] proposed an iterative compilation technique which factors in compiler writer's knowledge to reduce the search space. Further the search is performed only on the hot code segments. The search time in iterative compilation techniques could be reduced by using performance estimators as proposed in Cooper et al. [2005]. Agakov et al. [2006] proposed a biased search technique using Markov models to reduce the number of iterations required to converge to a near-optimal sequence. The Markov model used to bias the search space is learnt offline using a set of training programs.

Fig. 6. Performance comparison of the Cluster10 approach with All290 on PolyBench programs.

Fig. 7. Performance comparison of the Cluster10 approach with All290 on PolyBench programs.

Table IX. Comparison of Runtimes for -O2 and Reduced -O2 Sequences

Program	O2 time	O2-reduced time	O2-reduced length
flops-7.c	3.29	3.28	4
queens.c	1.81	1.77	7
whetstone.c	1.18	1.16	10
lists.c	5.36	4.44	3
n-body.c	1.05	1.02	0
flops-2.c	1.21	1.16	3
ReedSolomon.c	4.88	4.64	7
flops-3.c	1.59	1.56	6
flops-6.c	1.23	1.2	5
flops-5.c	1.3	1.3	8
perm.c	3.23	2.64	2
recursive.c	0.92	0.72	3
fbench.c	1.51	1.54	8
flops.c	7.55	7.49	6
richards_benchmark.c	0.77	0.69	8
perlin.c	4.42	3.9	7
flops-4.c	0.77	0.74	0
ary3.c	1.58	1.53	6
partialsums.c	0.52	0.52	0
lowercase.c	5.95	5.79	4
realmm.c	2.32	2.3	7
fib2.c	2.38	2.29	4
mandel-2.c	1.03	0.96	6
quicksort.c	3.19	2.99	8
hash.c	3.18	3.18	7
nsieve-bits.c	4.08	3.45	0
oscar.c	1.96	1.86	7
intmm.c	1.56	1.56	5
flops-1.c	2.05	2.05	1
ackermann.c	1.2	0.87	3
matrix.c	2.49	2.29	7
puzzle.c	0.49	0.04	3
queens-mcgill.c	1.83	1.75	7
fdry.c	1.74	1.68	10
fannkuch.c	2.08	2.1	4
strcat.c	1.31	1.19	0
chomp.c	1.61	1.51	1
misr.c	0.17	0.16	4
mandel.c	0.83	0.82	3
heapsort.c	2.12	2.08	4
salsa20.c	6.8	6.18	7
puzzle-stanford.c	2.91	2.88	9
lpbench.c	4.06	4.03	9
ourafft.c	4.08	4	9
flops-8.c	1.37	1.34	1
huffbench.c	4.91	4.12	16
pi.c	0.51	0.51	7
bubblesort.c	3.81	3.55	5
tre sort.c	6.78	6.73	9
towers.c	2.35	2.09	5
random.c	2.66	2.67	4
fp-convert.c	2.6	2.3	2
dry.c	1.75	0.47	13
spectral-norm.c	1.1	1.1	0
methcall.c	3.44	3.29	1
sieve.c	5.23	5.2	6
nestedloop.c	0.2	0	2
himenobmtxpa.c	0.79	1.6	12
objinst.c	3.66	0	8
ffbench.c	0.65	0.65	2
dt.c	1.76	1.76	2

As discussed earlier for the effectiveness of a machine learning technique, the choice of relevant features is vital. Leather et al. [2009] proposed an automatic feature generation technique which uses genetic programming. Cavazos et al. [2007] proposed a machine learning model using hardware performance counters as program features. They used a *logistic regression* model to predict the probability with which an optimization should be enabled. Once the probability prediction models for various optimizations are learnt, a biased search of the sample space is performed. Fursin et al. [2011] and Fursin and Temam [2010] proposed a probabilistic prediction model and they used static program features to represent a program. Park et al. [2011] proposed a sequence prediction model using tournament predictors whereas Thomson et al. [2009] proposed a nearest-cluster-based approach. Cavazos et al. [2006] proposed an approach for performance prediction when a particular program transformation is applied without actually running the programs. This approach can be used in the optimization sequence selection problem to reduce the number of program evaluations either in the iterative compilation or in the machine-learning-based techniques.

Machine-learning-based prediction models have been used in other parameter estimation problems like loop unrolling [Stephenson and Amarasinghe 2005]. Kisuki et al. [2000] tried to attack a similar problem using iterative compilation techniques. Wang and O'Boyle [2009] proposed a machine learning technique to predict an optimal scheduling policy for programs parallelized using *openmp*. There are many research efforts in applying predictive modeling techniques in adaptive virtual machines [Arnold et al. 2005; Cavazos and O'Boyle 2006; Gu and Verbrugge 2008].

6. CONCLUSIONS AND FUTURE WORK

In this article we proposed a practical approach to solve the optimization phase ordering problem. This approach is quite different from the iterative compilation and machine-learning-based prediction techniques. The idea is to downsample the infinitely large optimization sequence space using a benchmark suite representative of the program class distribution in the program. Then the downsampled sequence set can be further reduced to a very small set of good sequences such that it covers all program classes. Then given a new program we can try all the sequences from the good sequences set and choose the best sequence. There is no need to identify a program class during this process. The following are few interesting problems which are due to the proposed technique.

- (1) The effectiveness of the proposed approach depends on the diversity of the Microkernels benchmark suite. So program diversity analysis of the Microkernels benchmark suite has to be done and thereby enrich it with new programs.
- (2) The quality of the downsampled sequence set depends on the iterative compilation techniques applied on the Microkernels benchmark suite. Better iterative compilation techniques lead to higher-quality sequences in the downsampled set.
- (3) We can possibly reverse-engineer our approach to solve the program classification problem by finding correlations between the class labels given by optimization sequences to the programs and their static/dynamic features.

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 *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*.
- ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. 2005. A survey of adaptive optimization in virtual machines. *Proc. IEEE* 93, 2, 449–466.
- CAVAZOS, J., DUBACH, C., AGAKOV, F., BONILLA, E., O'BOYLE, M. F. P., FURSIN, G., AND TEMAM, O. 2006. Automatic performance model construction for the fast software exploration of new hardware designs.

- In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*. ACM, New York, 24–34.
- CAVAZOS, J., FURSIN, G., AGAKOV, F., BONILLA, E., O'BOYLE, M. F. P., AND TEMAM, O. 2007. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. IEEE Computer Society, Washington, DC, 185–197.
- CAVAZOS, J. AND O'BOYLE, M. F. P. 2006. Method-specific dynamic compilation using logistic regression. *SIGPLAN Not.* 41, 229–240.
- CHEN, Y., HUANG, Y., ECKHOUT, L., FURSIN, G., PENG, L., TEMAM, O., AND WU, C. 2010. Evaluating iterative optimization across 1000 datasets. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, 448–459.
- COOPER, K. D., GROSUL, A., HARVEY, T. J., REEVES, S., SUBRAMANIAN, D., TORCZON, L., AND WATERMAN, T. 2005. Acme: Adaptive compilation made efficient. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '05)*.
- COOPER, K. D., SCHIELKE, P. J., AND SUBRAMANIAN, D. 1999. Optimizing for reduced code space using genetic algorithms. *SIGPLAN Not.* 34, 1–9.
- COOPER, K. D., SUBRAMANIAN, D., AND TORCZON, L. 2002. Adaptive optimizing compilers for the 21st century. *J. Supercomput.* 23, 7–22.
- FURSIN, G., KASHNIKOV, Y., MEMON, A. W., CHAMSKI, Z., TEMAM, O., NAMOLARU, M., YOM-TOV, E., MENDELSON, B., ZAKS, A., COURTOIS, E., BODIN, F., BARNARD, P., ASHTON, E., BONILLA, E. V., THOMSON, J., WILLIAMS, C. K. I., AND O'BOYLE, M. F. P. 2011. Milepost gcc: Machine learning enabled self-tuning compiler. *Inte. J. Parallel Program.* 39, 3, 296–327.
- FURSIN, G. AND TEMAM, O. 2010. Collective optimization: A practical collaborative approach. *ACM Trans. Archit. Code Optim.* 7, 4, 20:1–20:29.
- GU, D. AND VERBRUGGE, C. 2008. Phase-Based adaptive recompilation in a jvm. In *Proceedings of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '08)*. 24–34.
- 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. In *Proceedings of the IEEE International Workshop on Workload Characterization*.
- KISUKI, T., KNIJNENBURG, P. M. W., AND O'BOYLE, M. F. P. 2000. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT '00)*. IEEE Computer Society, Washington, DC, 237.
- KULKARNI, P. A., WHALLEY, D. B., TYSON, G. S., AND DAVIDSON, J. W. 2006. Exhaustive optimization phase order space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*.
- LATTNER, C. AND ADVE, V. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '04)*.
- LEATHER, H., BONILLA, E., AND O'BOYLE, M. 2009. Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09)*. 81–91.
- LLVM. 2012. <http://llvm.org/docs/Passes.html>.
- PAN, Z. AND EIGENMANN, R. 2006. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*.
- PARK, E., KULKARNI, S., AND CAVAZOS, J. 2011. An evaluation of different modeling techniques for iterative compilation. In *Proceedings of the 14th international conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '11)*. ACM, New York, 65–74.
- POLYBENCH. 2012. <http://www.cse.ohio-state.edu/~pouchet/software/polybench/>.
- STEPHENSON, M. AND AMARASINGHE, S. 2005. Predicting unroll factors using supervised classification. In *Proceedings of the International Symposium on Code Generation and optimization (CGO '05)*. 123–134.
- THOMSON, J., O'BOYLE, M. F. P., FURSIN, G., AND FRANKE, B. 2009. Reducing training time in a one-shot machine learning-based compiler. In *Proceedings of the International Conference on Language and Compilers for Parallel Computing*. 399–407.
- TORCZON, L. AND COOPER, K. 2011. *Engineering A Compiler*, 2nd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA.
- TOUATI, S.-A.-A. AND BARTHOU, D. 2006. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd Conference on Computing Frontiers (CF '06)*.

- TRIANTAFYLIS, 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: Feedback-Directed and Runtime Optimization (CGO '03)*.
- WANG, Z. AND O'BOYLE, M. F. 2009. Mapping parallelism to multi-cores: A machine learning based approach. *SIGPLAN Not.* 44, 75–84.

Received June 2012; revised November 2012; accepted November 2012