Amir Hossein Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, John Cavazos
MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning
ACM Transactions on Architecture and Code Optimization, TACO 2015

The published version is available online at: https://doi.org/10.1145/3124452

MiCOMP: Mitigating the Compiler Phase-ordering Problem using Optimization Sub-sequences and Machine Learning

AMIR HOSSEIN ASHOURI, Politecnico di Milano
ANDREA BIGNOLI, Politecnico di Milano
GIANLUCA PALERMO, Politecnico di Milano
CRISTINA SILVANO, Politecnico di Milano
SAMEER KULKARNI, University of Delaware
JOHN CAVAZOS, University of Delaware

Recent compilers offer a vast number of multilayered optimizations, capable of targeting different code segments of an application. Choosing among these optimizations can significantly impact the performance of the code being optimized. The selection of the right set of compiler optimizations for a particular code segment is a very hard problem, but finding the best ordering of these optimizations adds further complexity. . The traditional approach of constructing compiler heuristics to solve this problem simply can not cope with the enormous complexity of choosing the right ordering of optimizations for every code segment in an application.

The predictive model uses (i) a platform-independent dynamic features, (ii) an encoded version of the compiler sequence and (iii) an exploration heuristic to tackle the problem.

Experimental results using the LLVM compiler framework and the Cbench suite show the effectiveness of the clustering and encoding techniques to application-based reordering of passes while using a number of predictive models. We perform statistical analysis on the prediction space and compare against (i) standard optimization levels O2 and O3, (ii) random iterative compilation, and (iii) two recent non-iterative approaches. We demonstrate that our proposed methodology outperforms the performance of -O1, -O2, and -O3 optimization levels in just a few iterations, reaching an average performance speedup of 1.31 (up to 1.51) on the Cbench benchmark suite.

Additional Key Words and Phrases: compilers, machine learning, autotuning, recommender systems

ACM Reference format:
Amir Hossein Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John cavazos. 2017. MiCOMP: Mitigating the Compiler Phase-ordering Problem using Optimization Sub-sequences and Machine Learning. *ACM Transactions on Architecture and Code Optimization* , , Article 1 (September 2017), 22 pages.
DOI: 10.1145/3124452

1 INTRODUCTION

Compiler developers typically design optimization passes in order to transform each code segment of a program to produce an optimized version of an application. The optimizations can be applied at different stages of the compilation process. Optimizing source code by hand is a tedious task and therefore compiler optimizations are provided to automatically transform code. However, these code optimizations are programming language, application, and architecture dependent. Additionally, the word optimization is a misnomer and there is no guarantee the transformed code will perform better than the original version. Understanding the behavior of the optimizations and the actual effect on the source-code and the interaction of the optimizations with each other

This work is partially supported by the European Commission Call H2020-FET-HPC program under the grant ANTAREX-671623.

are complex modeling problems. The problem is particularly difficult because compiler developers have to deal with hundreds of different optimizations that can be applied during the different compilation phases and this creates the phase-ordering problem. The phase-ordering problem has been an open-problem in the field of compiler research for many decades [25, 42]. The inability of researchers to solve the phase-ordering problem has led to advances in the more simple problem of selecting the right set of optimizations, but even this problem has yet to be solved [9, 12].

This process of selecting the right optimizations for each code segment is typically done manually, and the sequence of optimizations is constructed with little insight into the interaction between the preceding compiler optimizations in the sequence. The task of manually constructing heuristics to select the right sequence of compiler optimizations is infeasible given the ever growing number of compiler optimizations being integrated into compiler frameworks. As an example, GCC has more than 200 compiler passes, and LLVM-clang and LLVM-opt each have more than 100 transformations. Additionally, these optimizations are applied at very different phases of the compilation, including analysis passes and loop-nest passes. Most optimization flags are turned off by default, and compiler developers rely on software developers to know which optimizations will benefit their code. Compiler developers provide standard optimization levels, e.g. -O1, -O2, -Os, etc. to introduce a fixed-sequence of compiler optimizations that, on average, bring good performance on a set of benchmarks the compiler developers tested. Finding the best ordering of compiler optimizations can have substantial benefits for performance metrics such as execution time, power consumption, and code-size. To this end, using predefined optimizations usually is not good enough to bring the best achievable application-specific performance. In this paper, we propose a framework in order to mitigate the complexity of the phase-ordering problem. So far, there are two potential techniques we could use to predict good optimization orders for code being optimized:

- (i) *Intermediate Sequence Prediction*: This technique uses a model to predict the current best optimization (from a given set of optimizations) that should be applied based on the characteristics of code in its present state. [5, 24].
- (ii) *Complete Sequence Prediction*: This technique uses a model to predict the complete sequence of optimizations that needs to be applied to the code just by looking at characteristics of the code in its original state [10, 34–36].

The framework proposed in this paper, MiCOMP, falls under the second category. It uses predictive models on complete optimization sequences rather than individual optimizations. We characterize applications as a vector of dynamic features that are independent from the target architecture. Predicting the complete optimization sequence to apply to a piece of code, i.e. complete sequence prediction, has the benefit of only requiring a single-round of feature collection of the code before any optimizations are applied to it. In order to use classic machine learning algorithms with the phase-ordering problem, we adapt an encoding scheme to transform variable-length vectors of optimizations into fixed-length vectors. Our prediction models are trained offline and program features and different compiler configurations are fed as inputs. As outputs, a prediction model produces a speedup number without the need to actually run the code on the target architecture. The dynamic characterization is independent from the architecture the code is running; thus, it can bring portability among different architectures. Additionally, we define exploration heuristics to find the best models in the shortest time. Our metric of time is defined as the minimum number of predictions from the model to obtain the best version of the code being optimized. The heuristic is based on Adjusted Cosine Similarity [38] to correlate different configurations of optimizations with their corresponding predicted speedups across all the training data. A recommendation algorithm enables us to explore only a fraction of the configuration space to reach the best speedups rather than a state-of-the-art sorting/ranking [10, 34–36]. In our experimental results, we show that our technique can outperform LLVM's highest optimization level of `-O3` by just a few predictions (up to 3 in the worst observed case). . We selected the full set of applications from the Ctuning

Cbench benchmark [14] to assess and evaluate the benefits of the proposed approach and to prove its feasibility. The main contributions of the proposed approach are as follows:

- An independent predictive-modeling framework, capable of capturing the correlation between different compiler optimizations and their predicted speedup without having to run optimized code variants on the target platform. Our autotuning framework can be paired with any desired predictive models.
- . We have clustered different compiler optimizations, all taken from LLVM's `O3` into 5 different groups. The order of optimizations within a group is internally fixed but the ordering of the groups can be altered. In this work, these groups are called sub-sequences and we exploit the phase-ordering by using these sub-sequences rather than the individual optimizations. By starting from no optimizations (as the baseline) and exploring different orderings of the sub-sequences using the same optimizations available to `-O3`, we outperformed `-O3`.
- Adapting a simple mapping technique to encode an optimization sequence into a bit string. The proposed technique transforms a variable-length representation to a fixed-length feature vector representation. It allows us to apply traditional machine learning algorithms since they are mostly designed to cope with fixed-length feature vectors.
- Adapting a Recommender System (RS) approach on the prediction space to use dynamic information.

The rest of the paper organized as follows: Section 2 presents related work. Section 3 introduces our proposed methodology including all its components. In Section 4, we present our experimental results and evaluate the results by means of several comparisons in the Section 5. We conclude the paper with future work and the conclusion.

2 RELATED WORK

Literature on the phase-ordering problem is closely related to the problem of selecting the best set of compiler optimizations in a fixed ordering. Recent literature can be classified into two main classes: (i) autotuning and iterative compilation approaches and (ii) applying machine learning to the problem of optimization selection.

Autotuning addresses automatic code-generation and optimization by using different scenarios and architectures. It involves building techniques for automatic optimization of different parameters in order to maximize or minimize the satisfaction of an objective function. One strategy in autotuning consists of coupling the approach with random generation of code-variants at each run. This technique can generally improve application performance in reference to static-handcrafted compiler optimization sequences [1]. Given the complexity of the iterative compilation problem [9], it has been shown that applying compiler optimization sequences at random can be as good as using other algorithms such as Genetic algorithms or Simulated Annealing to choose which optimizations to apply [1, 10, 12]. Other authors [3, 8] explored compiler Design Space Exploration (DSE) techniques jointly with architectural DSE for VLIW architectures.

Applying *machine learning* to the problem of selecting the best compiler optimizations has been extensively investigated by many researchers in the past. Proposed methodologies [11, 13, 22, 28, 29, 41] were among the first notable works introducing the use of machine learning to solve compilation problems. Recent related work [4, 6, 7, 34, 35] also tackled the problem of selecting the best compiler optimizations to apply by utilizing Bayesian Networks with an application-independent characterization technique, predictive modeling with dynamic characterization, and predictive modeling with compiler representations (Intermediate Representation (IR)) . There have been different objective functions used with machine learning on the problem: i) A *speedup predictor* takes as input both the characterization of the program being compiled and an optimization sequence, and it predicts as output the speedup when applying that optimization sequence relative to a default optimization setting. [10, 34, 35] ii) A *sequence predictor* characterizes a program being

compiled and uses it as input to a model, and the model predicts a probability distribution of optimizations to apply to that program. [1, 4, 6, 37]. iii) A *tournament predictor* [36]) takes as input a triple corresponding to the characterization of the program and two optimization sequences. This model predicts whether the speedup after applying the the first optimization sequence will be more or less than speedup if applying the second optimization sequence.

. However, there are a few notable published studies that attempted to solve the problem.

Kulkarni and Cavazos [24] have applied *Neuro-Evolution for Augmenting Topologies* (NEAT) in the Java JikesRVM compiler to phase-ordering by using intermediate sequence prediction. They built prediction models that use as input features of the current state of the transformed source-code and define certain stop-condition rules to complete the final predicted sequence at each iteration. They used source-code features and the Java JikesRVM JIT compiler to experimentally evaluate their approach. In contrast, we tackle the problem using predictive modeling and dynamic independent-characterization of the applications and our proposed methodology enables us to predict the full-sequence in one-shot.

Matrins et al. [27] tackled the problem of phase-ordering by a DSE approach that uses a clustering-based selection method for grouping functions with similarities and exploration of a reduced search space resulting from the combination of optimizations previously suggested for the functions in each group. Authors used DNA encoding where program elements (e.g., operators and loops in function granularity) are encoded in a sequence of symbols, and followed by calculating the distance matrix and a tree construction of the optimization set. Consequently, they applied the compiler optimization passes already included in the DSE to measure the reduction in the total exploration time of the search space such as Genetic algorithm. Our proposed approach on the other hand is mainly different, as we mitigate the phase-ordering problem by inducing a prediction model rather than a design space exploration scheme. Once our model is trained, it can be further used for any number of applications under analysis to induce a prediction inexpensively and we believe it will bring scalability in autotuning compilers.

Other related work has approached the problem by exhaustively exploring the optimization ordering space at the granularity of functions [23]. The exhaustive enumeration these authors proposed, constructs probabilities of enabling/disabling interactions between different optimization sequences, but these probabilities are not specific to any program. Jantz et al. [19] proposed two pruning techniques to downsample the optimization space. As a result the authors could employ faster exhaustive phase-ordering searches on the new space. Ashouri et al. introduced an approach that uses predictive modeling to construct an intermediate sequence of optimizations for code being compiled [5]. Other related work used iterative Design Space Exploration (DSE) and clustering-based approaches to down-sample and cluster the available optimizations targeting performance gain and power reduction [30–32].

Our approach, MiCOMP, is significantly different compared with those mentioned in the literature. Our work mostly resembles the approach of Park et al. [35, 36]. However, our techniques tackle the significantly harder problem of the phase-ordering. We introduce a mapping function that encodes an optimization sequence into a bit string. It preserves the ordering and the repetition of the optimizations. At the same time, the proposed work is able to predict the complete optimization sequence to apply to the unoptimized code, rather than predicting the best optimization to apply to the current state of the optimized code [5, 24]. . We use dynamic architecture independent features to feed into our model. Moreover, we used clustering over all passes in LLVM's `-O3`, that tended to perform well, to significantly outperform the single optimization sequence performed by `-O3` itself. We do that by re-ordering these sub-sequences automatically based on the type of the application under optimization. To summarize, the presented work is the first approach that uses machine-learning based techniques on the phase-ordering problem to predict the complete sequence of optimizations. In Section 4, we improve the machine-learning model through Recommender System

techniques and assess the experimental results we obtain against the state-of-the-art phase-ordering approaches.

3 THE PROPOSED METHODOLOGY

Compilers typically ship with standard optimization levels (e.g. `-O2, -O3 and -Ofast`) each tuned during compiler development to obtain a certain level of performance on a standard set of benchmarks. These optimization levels do not always translate to good performance on other applications. The main objective of the proposed methodology is to introduce a compiler autotuning framework, which is able to dynamically reorder the compiler passes within LLVM's optimization level `-O3` , to achieve the maximum speedup for the applications being optimized. We found that if we could reorder sub-sequences of optimizations that tended to perform well, we could significantly outperform the single optimization sequence performed by `-O3`. This process should be customized based on the features of the application under analysis. To mitigate the phase-ordering problem, a model has to be constructed in such a way that it can correlate the effect of using different compiler sequences and the corresponding achievable speedup. MiCOMP uses such a model, and it can (i) recommend good sequences of optimizations that maximize an application's performance (ii) with very few predictions.

Phase-ordering is also complicated by allowing the possibility of variable-length compiler sequences. State-of-the-art approaches for selecting the right set of optimizations used fixed length feature-vectors [6, 10, 35, 36] to induce a prediction model. During the prediction phase, MiCOMP proposes an iterative process in which different solutions are explored by evaluating different optimization sequences with the potential of leading to higher speedups. We predict optimization sequences that will perform well against using state-of-the-art *ranking* [35, 36] techniques.

Figure 1 illustrates the two main phases of MiCOMP: (i) *offline training* and (ii) *online prediction*.

The *offline training phase* is used to learn about the effects of compiler optimizations when compiling an application. In particular, this phase is used to induce a prediction model considering application features and applied optimizations (including order and repetitions). This phase is performed once for each compiler and the model is built on a set of representative applications. In this phase, each application is passed through a single round of feature collection to extract an application's characteristics. A dynamic profiler is used to generate a representation of the program in terms of its features. Since a very large set of features is extracted for each application, we apply a dimension-reduction technique to reduce the number of features that is fed as input to the prediction model (e.g. PCA âĂŞ Principal Component Analysis [21]). This speeds up the learning during the model construction process. Application-profiling and dimension-reduction techniques are extensively described in Section 3.1. Next, an application is compiled with different configurations of compiler optimizations, executed and profiled in terms of speedup with respect to LLVM 's `-O3`. The speedup values together with the reduced program features and an encoded version of the used compiler optimizations (characterized by a fixed-length binary output, see Section 3.3) are fed to a machine learning algorithm to induce the speedup predictor (see Section 3.4). This model can then used during the online phase.

The *online prediction phase*, is used every time a new application is optimized. We use the same feature extraction and dimension reduction techniques described in the offline training phase. The collected features are used to query the speedup prediction model to predict the best set of compiler sequences to apply to an application. The goal of our method is to discover the fewest number of predictions that will be needed to obtain the optimization sequence that gives the best speedup possible. Thus, MiCOMP has been coupled with a heuristic derived from the field of Recommender Systems (see Section 3.5). This technique is used to obtain a predicted set of optimization sequences where each sequence is as diverse as possible to the other sequences in the set, thus guaranteeing coverage of a large part of the optimization configuration space, consequently obtaining a set of optimization sequences that are robust to model inaccuracies.

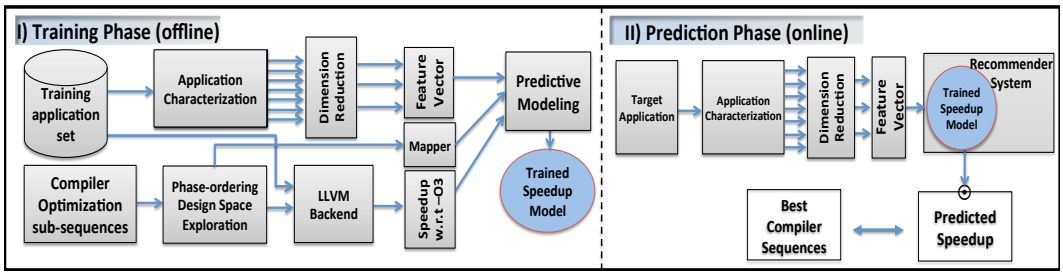


Fig. 1. Proposed framework. (i) offline-training phase which is done once and (ii) online-prediction phase for optimizing new unseen applications

3.1 Application Characterization

In this work, we used a PIN-based [26] dynamic instrumentation framework to analyze and characterize the behavior of applications at execution-time. In particular, our framework provides a high level Micro-architectural Independent Characterization of Applications (MICA) [17] suitable for characterizing applications in a target architecture agnostic manner. There is no static syntactic analysis, but the framework is solely based on dynamic MICA profiling. The MICA framework reports information about instruction types, memory and register access pattern, potential instruction level parallelism and a dynamic control flow analysis in terms of branch predictability. Overall, the MICA framework characterizes an application in reference to 99 different metrics (or *features*). Many of these 99 features are strongly correlated (e.g. the number of memory reads with stride smaller than 1K is bounded by the number of reads with stride smaller than 2K). To significantly improve the speed of model construction, we applied a dimension reduction by using Principal Component Analysis (PCA) [21] to reduce the number of features used to characterize an application. PCA is a technique to transform a set of correlated features into a set of orthogonal, i.e., uncorrelated principal components. The PCA transformation sorts the principal components by descending order based on their variance [20]. For instance, the first principal component includes the most input data variability, i.e., this component represents most of the information contained in the input data. To reduce the number of input features, while keeping most of the information contained in the input data, one simply needs to use the first k principal components as suggested in previous work [17]. In particular, we set $k = 5$, which captures more than 98% of the overall variance across all training data. .

3.2 Constructing Compiler Sub-sequences

In this section, we briefly explain our novel idea behind clustering certain compiler optimizations as *sub-sequences*. A phase-ordering optimization sequence represented by the vector $\boldsymbol{o}$ belongs to the n dimensional factorial space $|\Omega_{phases}| = n!$, where n represents the number of compiler optimizations under study. However, the mentioned bound is for a simplified phase-ordering problem having a fixed length optimization sequence length and no repetitive application of optimizations. Allowing optimizations to be repeatedly applied and a variable length sequence of optimizations will expand the problem space to:

$$|\Omega_{phases\_repetition}| = \sum_{i=0}^{m} n^i \tag{1}$$

Where n is the number of optimizations under study and m is the maximum desired length for the optimization sequence. Even for reasonable values for n and m, the entire search space is enormous. For example, assuming n and m are both equal to 10, this leads to an optimization search

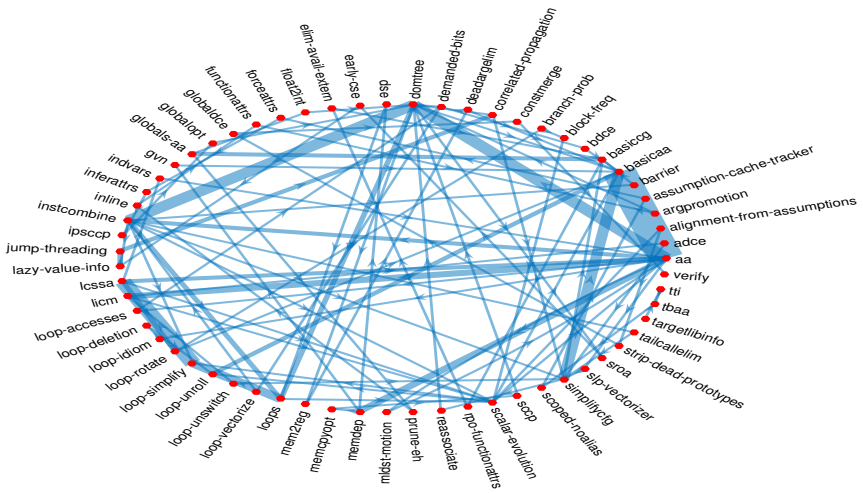


Fig. 2. Generated directed graph for LLVM's -O3. Each node in the graph represents an optimization pass. The edge thickness depicts the strengths in the connection between two nodes.

space of more than 11 billion different optimization sequences to select from for each piece of code being optimized [5] [1].

3.2.1 The Optimization Dependence Graph. Mitigating the phase-ordering problem with previous approaches is not practical due to the large number of different possible optimization sequences to select for each piece of code being optimized. MiCOMP proposes to group optimizations into clusters of sub-sequences that are known to perform well, . There are 157 compiler passes in LLVM optimization level `-O3` (more than 60 unique compiler passes) and selecting the most promising sub-sequences from these optimizations can positively affect the autotuning process. Among all these 157 compiler passes, some are analysis passes (i.e. basicaa, memdep, etc) which do not transform the code directly, but instead provide analysis information to other compiler passes that follow them. The rest are transformation passes, i.e., Aggressive Dead Code Elimination (adce), Loop Invariant Code Motion (licm), loop-rotate, etc., which perform optimizations on the code [2].

In this paper, we introduce the idea of clustering sub-sequences of all the passes available to the optimization level `-O3` and adapt prediction models to order these sub-sequences in ways that improve the performance of a particular application. We show that this technique can improve the performance of an application over using `-O3` by evaluating a few predicted orderings of the sub-sequences of optimizations.

This graph can be represented by a *Weighted Adjacency Matrix* that has the size of N×N. This matrix can be used for clustering optimizations into different sub-sequences. Figure 2 shows the constructed graph on `-O3`.

3.2.2 Graph and Sub-sequence Clustering. For our clustering of optimizations into sub-sequences, we could have used any number of the numerous clustering methods proposed in the literature related to Pattern Recognition (e.g. iterative, hierarchical, divisive, etc,) [39]. We selected *agglomerative clustering* [43] which is an iterative clustering technique that merges smaller clusters and improves the complexity of k-mean clustering on graphs [39]. A key insight of this method is that

[1]The problem of phase-ordering does not have deterministic upper-bound in the case of unbounded length.
[2]http://llvm.org/docs/Passes.html

it treats clusters as a dynamical system and its samples as states. The algorithm works as follows: Agglomerative clustering, receives as input the matrix of the graph G and the number of desired clusters (n_T) and builds (i) the graph G with k-nearest-neighbors upon computing its Weighted Adjacency Matrix (W). (ii) The algorithm then calculates the transition probabilities and (iii) forms sample clusters $C = \{c_1, ..., c_{nc}\}$. (iv) It enters a loop to iteratively try to add more sub-clusters to the already available clusters in C as long as the conditional sum of the all-path integrals within the new sub-clusters maximizes some objective function (*argmax*) [43]. A path integral is a metric to measure the stability of a dynamical system and is computed by summing the paths within the cluster on the directed graph weighted by transition probabilities. We used the algorithm and tentatively increased the number of max desired clusters until no clusters could be added. The final five clusters, namely, the best optimization sub-sequences the algorithm could find are reported in Section 4 Table 4.

3.2.3 Benefits of Sub-sequences. Clustering optimizations into sub-sequences makes sense. Certain analysis algorithms typically should be done before an optimization in order for the optimization to have any significant impact. For example, we may want to run analysis that performs basic block counts and predicts branch instruction outcomes before applying an optimization that reorders the code blocks in an application. Additionally, it is likely that `-O3` will contain optimizations that should follow other optimizations in order to obtain the best performance. Thus, forming a cluster of optimizations that should be applied together makes a lot of sense.

3.3 The Proposed Encoder

Constructing prediction models for the problem of selecting the right compiler optimizations with fixed-length feature vectors has been extensively studied [10, 34–36]. However, prediction models fall short when correlating program characterizations with the right compiler optimizations to apply when it comes to a variable optimization sequence length [2]. Therefore, we adapt a simple mapping technique to encode an optimization sequence into a bit string. The proposed technique transforms a variable-length representation to a fixed-length feature vector representation.

Let $\mathcal{A} = \{\alpha_1, \ldots, \alpha_N\}$ be the set of all variables, which can be thought of as an *alphabet*. Every α_i is a *letter*. A finite string of not necessarily distinct letters is called a *word*. Thus, each word is a concatenation of the form $\alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_k}$, where $i_1, i_2, \ldots, i_k \in \{1, \ldots, M\}$. The integer k is the *length* of the word. We will also allow the empty word which by definition has length zero.

There is a simple way of encoding the space $\mathcal{W}$ of all words of length at most M using the space described by $\{0,1\}^{N\times M}$ consisting of all binary strings of the fixed length $N \times M$. To see this, consider the mapping function $f : \mathcal{A} \to \{0,1\}^N$ which encodes each letter α_i to the binary string $f(\alpha_i) = b_1 \cdots b_M$, where

$$b_j = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{if } j \neq i. \end{cases}$$

Now we define the mapping function $F : \mathcal{W} \to \{0,1\}^{N\times M}$ by encoding each word $\alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_k}$ to the binary string

$$F(\alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_k}) = f(\alpha_{i_1})f(\alpha_{i_2})\cdots f(\alpha_{i_k}) \underbrace{\mathbf{0}\cdots\mathbf{0}}_{N-k \text{ times}},$$

where $\mathbf{0} = 0\cdots0$ is the zero string of length N. Evidently the map F is one-to-one. The image $F(\mathcal{W})$ is much smaller than the target space $\{0,1\}^{N\times M}$, as these sets have $\sum_{k=0}^{M} N^k$ and $2^{N\times M}$ elements, respectively.

If we identify each element of $\{0,1\}^{N\times M}$ with a concatenation $s_1 \cdots s_N$ of N elements of $\{0,1\}^N$, the image $F(\mathcal{W})$ can be simply characterized by the following two requirements:

1. Each s_i has at most one non-zero binary digit.
2. If $s_i = \mathbf{0}$ and $s_j \neq \mathbf{0}$, then $i > j$.

Given the proposed encoding, there exists a one-to-one (1:1) mapping, F, for every instance of $\mathcal{A} = \{\alpha_1, \ldots, \alpha_N\}$ with the binary size of $N \times M$ that has the same characteristics of the original presentation

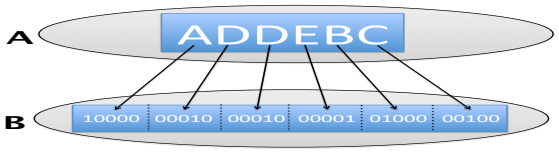


Fig. 3. An example of the proposed encoding scheme on the example where we have repetitions and $\{N = 5, M = 6\}$: Each letter represents a compiler sub-sequences containing different compiler optimizations.

with the benefit of having a fixed $N \times M$ length. An example of the proposed encoding scheme is shown on the Figure 3. Our adapted mapping function uses a *one-hot encoding* approach [33] for N=5 and M=6 to assign a single high (1) for each subsequences while other bits are turned off (0). This technique can inexpensively preserve the order and the repetitions of optimizations in a sequence, at the same time it assures the transformed feature vector has fixed-length size.

3.4 Predictive Modeling

To this end, the proposed methodology in Figure 1 illustrates the use of predictive modeling in both the offline (training) and online (testing) phases of the process. We used the predictive modeling in the offline training phase to (i) construct the model and in (ii) the online prediction phase we exploit the constructed model on the target application to predict the speedup of a complete optimization sequence without the need to actually apply the sequence of optimizations to the code.

3.4.1 Constructing the Prediction Model. Predictive modeling is the process of constructing, testing, and validating a model to predict an unobserved outcome based on characterization of a state from which to predict the outcome. In this paper, the state being characterized is the code being optimized, and the predicted outcome corresponds to the speedup metric calculated by normalizing the execution time of the current optimization sequence by the execution time of the baseline optimization sequence. The general formulation of the optimization problem is to construct a function that takes as input the features of the unoptimized program being compiled. In other words, this model takes as an input a tuple (F, T) where F is the feature vector of the collected instrumentation of the program being optimized; and T is one of the several possible compiler optimization sequences predicted to perform well on this program. Its output is a prediction of the speedup T should achieve when applied to the original code.

3.4.2 Analysis of Selecting the Compilation Baseline. As explained in Section 3.2, we do not use any of the default compilation optimization levels as a baseline to start from since we used all compiler optimizations passes that are used in `-O3` for our clustering purposes (see Section 3.2.2). Additionally, we found that using a baseline compiler optimization level to start from ultimately reduces the speedup achievable from the sequence we construct with predictive modeling.

Results suggests that using the MiCOMP optimization sequence without an optimization level as a baseline can lead to substantial benefits compared with using any of `-OX` optimization levels as a baseline. . Note that using a baseline of `-O1`, `-O2`, or `-O3` each converge to a sub-optimal speedup. Thus, applying certain sequences causes a degradation in performance as can be seen by using these standard optimization levels as a baseline. The better option is to not use a baseline sequence at all and to allow MICOMP to predict the best sequence to apply on its own.

The insights of this experiment are threefold: **(i)** The clustering technique is beneficial; first, to gain better speedup values and second, . **(ii)** The sub-sequences can be coupled with machine-learning techniques so they can be reordered based on the applications being optimized while outperforming the highest standard optimizations levels. **(iii)** Phase-ordering does matter in the field of compilers; i.e., using the same set of optimization flags available to `-O3`, MiCOMP can significantly outperform `-O3` itself.

3.4.3 Application-specific Prediction. Our machine-learning constructed models can be used for unseen target applications to predict the speedup when applying compiler sequences to them. The predicted speedup values correspond to the optimization sequence applied to the program. For a given input program, first a feature vector containing dynamic instrumentation is collected. Then our prediction model is fed the features of the program being compiled to predict the expected speedup if an optimization sequence T was applied to

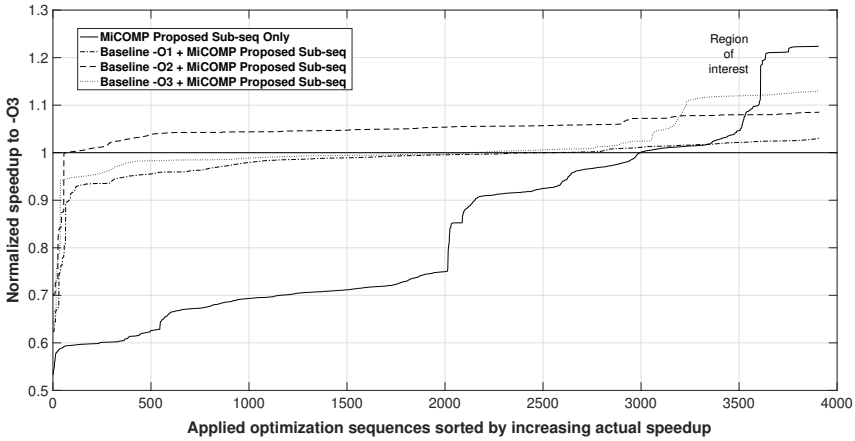


Fig. 4. Empirical analysis of having different compilation baseline across all CBench applications (Harmonic mean). Region of interest is depicted where MiCOMP sub-sequences outperformed other compiler sequences having a fixed standard compilation baseline.

it. By predicting the performance of each possible optimization sequence that can be applied, it is possible to rank the optimization sequences according to their expected speedup and only select the sequences to actually apply that are predicted to give the highest speedups.

A state-of-the-art ranking approach [34, 36] was used to rank optimization sequences in descending order, and we only select the top N optimization sequences to evaluate their actual optimization quality. In this work, we propose an iterative process in which different solutions are explored to find those leading to higher speedups. In other words, our proposed exploration technique uses the output of our prediction model to generate an initial exploration strategy, and the exploration strategy dynamically updates itself in order to reach the highest speedup values in the least number of predictions.

3.5 Recommender System Heuristic

In the initial steps taken by [5, 24], the authors defined iterative exploration heuristics, based on the current optimized state of the target application being compiled, to select the next best optimization to apply, which will bring the eventual best speedup. As the current state of the optimized application depends on the optimizations that were already applied, this previous approach required several rounds of feature collection. In this paper, we propose a predictive approach that generates the complete optimization sequence for a program that has not been optimized, thus it needs to collect features only once before any optimizations are applied.

3.5.1 Adjusted Cosine Similarity. Many of the aforementioned state-of-the-art approaches, tackling both the selection and the phase-ordering problem, define exploration strategies on the optimizations design space. Yet, to the best of the authors' knowledge, none of them make use of information in order to dynamically improve the strategy itself. Dynamic information, in our particular case, is the predicted speedup on the sequences already explored and evaluated. The knowledge can be effectively used to improve the initial exploration. The technique we propose, leverages the similarity between the unexplored and the explored optimization sequences. In particular, our proposed technique prioritizes the evaluation of solutions less similar to the ones already explored. This is especially important for the phase-ordering problem where there are a plethora of optimization sequences that need to be explored. The similarity measure is based on how close the achieved speedup is for predicted solutions across all the training data. As an example, let $S_{p,i}$ and $S_{p,j}$ be the predicted speedups of the sequences i and j when applied to program p in the set of programs P. We define an iterative process to look for predicted similarities in i and j when they were applied to different programs in P.

In recommender system (RS), an algorithm called Basic Cosine Similarity is used to correlate users and items. However, computing the similarity using this algorithm has one important drawback: the difference in rating scale are not taken into account. The Adjusted Cosine Similarity offsets this drawback by subtracting

the corresponding user-average from each co-rated pair and is shown to have the lowest error-rate amongst the different similarity measurement techniques [38]. . Using this method, we can compute the Adjusted Cosine Similarity between optimization sequence i and j as:

$$sim(i,j) = \frac{\sum_{p \in P}(S_{p,i} - \bar{S_p})(S_{p,j} - \bar{S_p})}{\sqrt{\sum_{p \in P}(S_{p,i} - \bar{S_p})^2}\sqrt{\sum_{p \in P}(S_{p,j} - \bar{S_p})^2}} \quad (2)$$

where $S_{p,i}$ is the speedup achieved by sequence i when applied to program p of all set of programs P, and $\bar{S_p}$ is the average speedup on program p. We use the computed measure to evaluate the correlation between a pair of optimization sequences to bias our exploration strategy.

We define the exploration strategy inspired by ACS as follows:

(1) Sort predicted speedup solutions in decreasing order in a list.
(2) Test solutions in order. If the solution to test is too similar to one already tested in the current list iteration, skip it.
(3) If the end of the list has been reached and there are still optimization sequences to test, go to 2. Start from the head of the list and exclude already tested solutions.

High values of ACS for a pair of optimization sequences are the consequence of achieving pairwise similar speedups across all training data. We employ this measure to hint exploration priority to the solutions that are less similar to the ones already tested. .

4 EXPERIMENTAL RESULTS

In this section we evaluate our proposed methodology on an Intel Xeon architecture. We adapted our instrumentation and architecture-independent tool (Section 3.1) to extract characteristics from a large set of benchmarks from the Ctuning CBench suite [14]. We have used LLVM compilation framework v3.8 (`Clang` for the frontend/backend and `Opt` for the optimization passes). The training set consists of different applications ranging from automotive, security, office, and telecom. The list of applications we evaluated is reported in Table 3. Table 4 illustrates the list of different compiler optimizations that are clustered into 5 different *sub-sequences* (refer to Section 3.2.2) that are derived from LLVM's `-O3`. We used the sub-sequences with no baseline in MiCOMP and generate the design space enabling orderings and repetitions of these sub-sequences. The optimizations are fixed within a sub-sequence, but sub-sequences are allowed to appear in any order in the full optimization sequence.

The application execution time has been estimated by using the Linux *Perf* tool. . The execution time is done by averaging three loop-wraps of the specific compiled binary with 1s of sleep in between three different executions of those loop-wraps. Therefore, in total, each individual transformed application binary has been executed 9 times as three packages of three loop-wraps to ensure better accuracy of estimations and fairness among the generation of executions. This technique is used both in the training and inference phases. In order to implement a dimension reduction technique we applied PCA.

This analysis reveals that by using a 5-D vector of features (a single-dimensional vector of length 5) we can capture 98% of the variance available in the training set. The Principal Components (PCs) have been computed using the MICA features collected from application executions (it is required only once), normalized by standard deviation across all data sets.

The proposed methodology is prediction model independent, and we report the results using three different models described in Table 5. To this end, we used **(i)** a Linear Regression (LR) classifier using the M5 attribute selection method with default ridge parameter, **(ii)** a *Multilayer Perceptron* using the default configuration, and the **(iii)** K* algorithm using default settings. In this work, the WEKA machine learning tool [15] has been integrated in our framework. We trained different speedup predictors, each one by excluding from the training application set, one of the applications. This technique is called Leave-One-Out-Cross Validation (LOOCV) and ensures a fair evaluation of our trained models. Validation data is used on the application excluded from the training set for prediction purpose.

4.1 Analysis of the variability of the distributions

4.2 Analysis of the MiCOMP's training data dependency

4.3 Analysis of the MiCOMP's timing breakdown

An application characterization phase takes between 15 to 40 seconds depending on the type of the application. We noticed a small factor of slowdown when we perform the feature collection phase versus measuring the pure application's execution time. The overhead is negligible first, as it is required once and second, the

Table 1. Analysis of paucity of data when different categories are used

Scenario		Speedup w.r.t. MiCOMP's default
Training	**Testing**	
Automotive	Security	0.9443
Automotive	Telcom	0.9743
Automotive	Consumer	0.9432
Automotive	Network	0.9896
Automotive	Office	0.9896

Table 2. MiCOMP timing breakdown for offline training and online inference

Phase	Category	Time
Offline Training	(A) Offline Data-collection (32 App)	5 days
	(B) Model Construction (MLP)	120 sec
Online Prediction	(C) Susan-c Feature Collection	17.4 sec
	(D) Susan-c Compilation	4.5 sec
	(E) Susan-c Execution	9.7 sec
	(F) Prediction (MLP)	2 sec
	(G) Recommendation (ACS)	12 sec

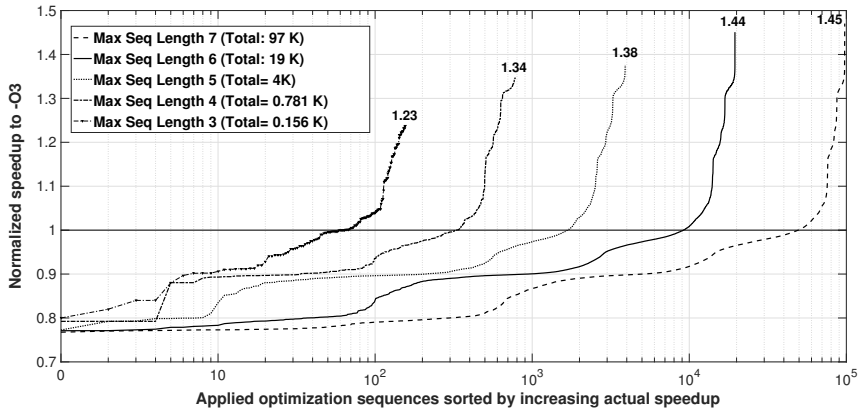


Fig. 5. Empirical analysis of having different compiler sequence lengths on 5 candidate applications: *telecom_adpcm_d, jpeg_d, bzipd, network_dijkstra, automotive_bitcount*. Note that X axis is in logarithmic scale.

speedup gained by using MiCOMP is far higher. In MiCOMP, cross validation is done in a few minutes for each application under analysis. Model construction is heavily correlated with the type of machine learning algorithm we use. We observed LR to be the fastest and MLP to be slowest for our data.

4.4 Analysis of Longer Sequence Length

As described in Section 3, MiCOMP requires having upper bound on the sequence length for using the encoding scheme. To this end, we evaluate MiCOMP by having different maximum values for the sequence length. A speedup prediction model requires a one time expensive training be done in order to construct an accurate model. We believe that the longer the sequence length, the better the chance of finding higher speedup values. We have tested our proposed sub-sequences with different maximum sequence lengths to empirically find the most effective length across all the training applications. This is done also with the goal of scalability and ultimately speeding up the training phase.

Table 3. Full Cbench Applications under analysis (CTuning CBench suite v1.1 [14])

No.	cBench list	Description
1	automotive_bitcount	Bit counter
2	automotive_qsort1	Quick sort
3	automotive_susan_c	Smallest Univalue Segment Assimilating Nucleus Corners
4	automotive_susan_e	Smallest Univalue Segment Assimilating Nucleus Edges
5	automotive_susan_s	Smallest Univalue Segment Assimilating Nucleus Smoothing
6	security_blowfish_d	Symmetric-key block cipher Decoder
7	security_blowfish_e	Symmetric-key block cipher Encoder
8	security_rijndael_d	AES algorithm Rijndael Decoder
9	security_rijndael_e	AES algorithm Rijndael Encoder
10	security_sha	NIST Secure Hash Algorithm
11	security_pgp_d	public key cryptography for the masses
12	security_pgp_e	public key cryptography for the masses
13	telecom_adpcm_c	Intel/dvi adpcm coder/decoder Coder
14	telecom_adpcm_d	Intel/dvi adpcm coder/decoder Decoder
15	telecom_gsm	gsm encoder/decoder
16	telecom_CRC32	32 BIT ANSI X3.66 CRC checksum files
17	consumer_jpeg_c	JPEG kernel
18	consumer_jpeg_d	JPEG kernel
19	consumer_lame	MP3 encoding engine
20	consumer_mad	MPEG audio decoder
21	consumer_tiff2bw	convert a color TIFF image to grey scale
22	consumer_tiff2rgba	convert a TIFF image to RGBA color space
23	consumer_tiffdither	convert a TIFF image to dither noisespace
24	consumer_tiffmedian	convert a color TIFF image to create a TIFF palette file
25	network_dijkstra	Dijkstra's algorithm
26	network_patricia	Patricia Trie data structure
27	office_stringsearch1	Boyer-Moore-Horspool pattern match
28	office_ghostscript	Aladdin Ghostscript
29	office_ispell	An interactive spelling corrector
30	office_rsynth	Klatt synthesizer
31	bzip2d	BurrowsâĂŞWheeler compression algorithm
32	bzip2e	BurrowsâĂŞWheeler compression algorithm

Table 4. Candidate clusters of compiler optimizations into sub-sequences (all derived from `LLVM -O3`)

sub-seq	Compiler Passes
A	-alignment-from-assumptions -argpromotion -barrier -bdce -block-freq -branch-prob -constmerge -deadargelim -demanded-bits -dse float2int -forceattrs -functionattrs -globaldce -globalopt -globals-aa -gvn -indvars -inferattrs -inline -ipsccp -jump-threading -lcssa -loop-accesses -loop-deletion -loop-idiom -loop-unroll -loop-unswitch -loop-vectorize -mldst-motion -prune-eh -reassociate -rpo-functionattrs -sccp -simplyfycfg -sroa -strip-dead-prototypes
B	-licm -mem2reg
C	-instcombine -loop-rotate -loop-simplyfy
D	-memcpyopt
E	-adce -loop-unswitch -slp-vectorize -tailcallelim

Figure 5 gives the *Harmonic* mean (as suggested by [16] [3]) values of the actual speedups using five selected applications each having different upper bound sequence lengths. We randomly selected an application from each of CBench categories (automotive, compression, telecom, consumer, office and network) since it was impractical to do this analysis with all applications. Having the upper bounds set to 3, 4, 5, 6 and 7 respectively, gives search spaces of 156, 781, 3909, 19k and 97k distinct permutations of sub-sequences with repetitions enabled (refer to Equation 1 for the optimization space). The five speedup lines show the trend of reaching a higher speedup value by iteratively exploring larger fraction of the optimization space. The maximum speedup found against `-O3` using sequence lengths of 3, 4, 5, 6, and 7, respectively, are 1.23, 1.34, 1.38, 1.44 and 1.45. These results suggest to set the maximum length to 6 as this ensures achieving good speedups while avoiding a potential exploration of 100K sequences per each application in the training set.

[3]We provide harmonic mean rather than geometric mean as we are dealing with averaging speedups. Note that harmonic-mean is always less than or equal to geometric mean.

Table 5. List of the predictive models used in our experiments. Note that the proposed methodology is independent from any specific machine-learning algorithm (classifier) and it can be paired with any algorithm desired.

Predictive Model	Description
MultilayerPerceptron (MLP)	A feedforward artificial neural network model that maps sets of input data onto a set of appropriate outputs. A MLP consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one
LinearRegression (LR)	An approach for modeling the relationship between a scalar dependent variable y and one or more explanatory variables (or independent variables) denoted X. In linear regression, the relationships are modeled using linear predictor functions whose unknown model parameters are estimated from the data.
KStar	It is an instance-based classifier, that is the class of a test instance is based upon the class of those training instances similar to it, as determined by some similarity function. It differs from other instance-based learners in that it uses an entropy-based distance function.

Table 6. Average error rate for the proposed encoding function versus an arbitrary encoding

M.L	MiCOMP Encoding		Arbitrary Encoding		Improvement Factor	
	MAE	AE	MAE	AE	MAE	AE
MLP	0.06778	0.05439	0.10826	0.11838	1.59×	2.16×
LR	0.07515	0.07795	0.12879	0.13974	1.71×	1.79×
KStar	0.05129	0.05078	0.09188	0.10866	1.77×	2.13×

4.5 MiCOMP's Prediction Accuracy

Unlike sequence prediction models [1, 6, 36] in speedup prediction approaches, prediction quality is measured by means of prediction error. This metric demonstrates how close the prediction values were to the actual speedups given the same sequence. We use the following different error measurement techniques.

Mean Absolute Error. In statistics, the Mean Absolute Error (MAE) [18] is a quantity used to measure how close predictions are to the eventual outcomes. The mean absolute error is given by:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|f_i - y_i| \tag{3}$$

where we define e_i as $|f_i - y_i|$ given f_i as the prediction values and y_i the actual values. Consequently, the value e_i is inverse proportional to the accuracy of the prediction.

Approximation Error. Complementary to MAE, Approximation Error (AE) [40] is a common error measurement whereas in some data there is some discrepancy between an exact value and the approximation. An approximation error can occur because (i) certain measurements of the data are not precise (which we consider it can be the case for any computer scientific measurement) and (ii) approximated values are used instead of the real values (the iterative prediction way keeps using the predicted values). It is calculated as:

$$\delta = \frac{|\epsilon|}{|v|} = \frac{|v - v_{approx}|}{|v|} \tag{4}$$

where the absolute error is the magnitude of the difference between the exact value and the approximation. These definitions can be extended to the case when v and $v_{approximate}$ are n-dimensional vectors, then by replacing the absolute error with an n-norm error.

4.5.1 Prediction Accuracy. We provide the prediction's error rate in Table 6. We observe that the arbitrary encoding leads to higher error rates in the prediction values. . Table 6 shows that the KStar model does slightly better in terms of accuracy compared with other models, it achieves around 5% error rate on average. In general, having a smaller error rate does not always guarantee higher performance gain but rather showcases the accuracy of the prediction model to capture the correlation between different compiler sub-sequences and the speedup values. [4]

[4]We are aware of the many other encoding possibilities that are more efficient (currently having $N \times M$ length). However, we believe that extending the current encoding scheme to a more sophisticated version is out of scope of the work. Moreover, the proposed clustering technique can effectively reduce the number of N, thus the encoding scheme is scalable for higher orders.

Table 7. Best compiler optimization sub-sequences found using an exhaustive iterative compilation and their related speedups

Application	Best sub-sequence	Speedup w.r.t. -O3	Kruskal-wallis *p*-value
telecom_adpcm_c	ECDDCC	1.35	0.9999
security_sha	ACCACE	1.06	0.9934
security_blowfish_e.csv	BCCEEA	1.13	0.9909
automotive_susan_e.csv	AABACA	1.15	0.9981
consumer_tiffdither	DCEDCD	1.20	0.9999
security_rijndael_e	CAEEC	1.10	0.9999
consumer_tiff2bw	CCDCD	1.30	0.9999
bzip2e	CBADCA	1.30	0.9944
automotive_susan_s	ECCCDE	1.22	0.9999
office_stringsearch1	ABCBAC	1.07	0.9999
telecom_adpcm_d	DCAACA	1.13	0.9939
consumer_jpeg_c	DDC	1.41	0.9932
network_patricia	CECBAA	1.18	0.9954
automotive_susan_c	BDBCCB	1.32	0.9999
consumer_tiff2rgba	DEDDC	1.32	0.9999
automotive_qsort1	CBAAAC	1.04	0.9969
security_blowfish_d	DACECA	1.09	0.9949
network_dijkstra	EECBBE	1.51	0.9991
security_rijndael_d	ECEACD	1.09	0.9992
bzip2d	CBDACA	1.29	0.9994
automotive_bitcount	BEACCA	1.19	0.9999
consumer_jpeg_d	CCED	1.18	0.9999
consumer_tiffmedian	BCBACB	1.15	0.9929
telecom_CRC32	DCAACA	1.26	0.9999
telecom_pgp_d	DCAACA	1.21	0.9999
telecom_pgp_e	DCA	1.22	0.9949
office_ispell	ABEBAE	1.09	0.9999
office_ghostscript	ABCBAC	1.08	0.9999
office_rsynth	ABCBA	1.10	0.9969
consumer_mad	DDCA	1.17	0.9999
consumer_lame	DDCAB	1.15	0.9999
Harmonic mean		**1.31**	**0.9976**

4.5.2 Iterative Compilation Max Speedups. *Iterative compilation* is known to be able to achieve good performance results when compiling applications [9]. However, the approach is expensive and should be combined with more intelligent search algorithms [1, 6]. Table 7 reports the maximum speedups found by an iterative compilation approach using our proposed clustering while exploring the full optimization space exhaustively. This experiment empirically confirms that the proposed clustering is useful on the phase-ordering space since we show that we can achieve on average a 31% speedup versus `-O3`. Figure 4 illustrates the trend when using MiCOMP sub-sequences with no baseline compared with having a baseline (e.g.: `-O1`, `-O2` or `-O3`). The best optimization sequence for each of applications under-analysis and its speedup value are reported in the second and the third columns of Table 7. Readers can refer to Table 4 to find the exact set of compiler optimizations clustered in each sub-sequence.

4.6 Performance Gain of The MiCOMP Technique Against The Ranking Approach

Our approach can improve the exploration to find the best optimization sequences in an optimization search space and to find the best speedups using a fewer number of predictions. Table 8 reports the comparison between the best speedup found by our approach and a state-of-the-art N-shot approach [34, 36]. The results, averaged using a Harmonic mean across all applications, show that using the same number of predictions from both models, our exploration technique can outperform the ranking approach on every number of predicted optimization sequences used (1, 5, 10, 15 and 20). This shows that our proposed methodology can effectively predicts the best compiler sequences to use and converges faster to better solutions in the space.

5 COMPARATIVE RESULTS

In this section we evaluate the results of our model against three different techniques: **(i)** Standard optimization levels, **(ii)** Random Iterative Compilation (RIC) and, **(iii)** state-of-the-art prediction Models. We use our MiCOMP exploration policy and compare the performance of predictions to a previously published ranking approach [34, 35]. For each application under analysis, we tested the speedup gained using 1, 5 and 10

Table 8. Prediction improvement of MiCOMP based on Adjusted Cosine Similarities against the Ranking (N-shot approach)

Exploration Techniques	Top-1	Top-5	Top-10	Top-15	Top-20
MiCOMP	1.01	1.06	1.09	1.10	1.12
Ranking	0.93	1.02	1.06	1.07	1.08

Table 9. MLP's speedup table against LLVM's -O3. Reported numbers are A (B%): **(A)** speedup and **(B)** percentage speedup w.r.t. the optimal speedup value of exhaustive exploration. Values are reported for 1 prediction, 5 predictions and 10 predictions.

Application	**1 prediction**	**5 predictions**	**10 predictions**
automotive_bitcount	1.04 (95.38%)	1.07 (98.12%)	1.08 (98.92%)
automotive_qsort1	1.01 (95.32%)	1.03 (96.93%)	1.03 (97.55%)
automotive_susan_c	1.04 (96.61%)	1.06 (98.53%)	1.06 (99.07%)
automotive_susan_e	1.04 (96.47%)	1.03 (98.41%)	1.04 (99.00%)
automotive_susan_s	0.99 (96.26%)	1.01 (98.42%)	1.02 (98.98%)
bzip2d	0.93 (92.77%)	0.96 (94.02%)	1.00 (94.37%)
bzip2e	1.09 (83.77%)	1.10 (86.02%)	1.12 (90.37%)
consumer_jpeg_c	1.01 (85.18%)	1.07 (90.35%)	1.10 (94.51%)
consumer_jpeg_d	1.09 (84.70%)	1.14 (88.97%)	1.17 (97.85%)
consumer_tiff2bw	0.96 (75.54%)	0.99 (80.59%)	1.02 (82.46%)
consumer_tiff2rgba	0.91 (80.61%)	0.95 (86.19%)	1.07 (88.08%)
consumer_tiffdither	1.02 (80.14%)	1.09 (85.86%)	1.11 (87.68%)
consumer_tiffmedian	0.94 (79.21%)	1.02 (85.72%)	1.06 (89.31%)
consumer_mad	1.02 (82.14%)	1.09 (85.86%)	1.11 (87.68%)
consumer_lame	0.99 (89.21%)	1.02 (90.72%)	1.06 (92.31%)
network_dijkstra	1.13 (60.00%)	1.29 (68.46%)	1.38 (73.00%)
network_patricia	0.91 (74.99%)	0.93 (80.79%)	0.97 (93.91%)
office_ispell	0.98 (84.99%)	1.01 (90.79%)	1.03 (93.91%)
office_ghostscript	0.99 (79.99%)	1.03 (82.79%)	1.03 (90.91%)
office_rsynth	1.01 (84.99%)	1.02 (90.79%)	1.03 (93.91%)
office_stringsearch1	0.98 (64.99%)	1.02 (70.79%)	1.01 (73.91%)
security_sha	0.93 (64.99%)	1.01 (70.79%)	1.03 (73.91%)
security_blowfish_e	0.97 (64.99%)	1.03 (70.79%)	1.03 (73.91%)
security_blowfish_d	0.97 (64.99%)	0.99 (70.79%)	1.02 (73.91%)
security_rijndael_e	0.99 (64.99%)	1.02 (70.79%)	1.01 (73.91%)
security_rijndael_d	1.00 (64.99%)	1.01 (70.79%)	1.04 (73.91%)
telecom_adpcm_c	0.96 (64.99%)	1.01 (70.79%)	1.02 (73.91%)
telecom_adpcm_d	0.98 (64.99%)	1.02 (70.79%)	1.01 (73.91%)
telecom_gsm_d	0.93 (64.99%)	1.03 (70.79%)	1.04 (73.91%)
telecom_CRC32	1.01 (85.18%)	1.07 (90.35%)	1.10 (94.51%)
telecom_pgp_d	1.04 (96.61%)	1.06 (98.53%)	1.06 (99.07%)
telecom_pgp_e	1.02 (80.14%)	1.09 (85.86%)	1.11 (87.68%)
Harmonic mean	**1.03 (84.74%)**	**1.05 (87.51%)**	**1.09 (91.52%)**

predictions and provide the Harmonic mean values. For example, one can see that for the network_dijkstra application we can gain a higher speedup values using MiCOMP and, on average even better than -O3 from just the first prediction. Moreover, we can achieve a 4% performance improvement over -O3 when we use 5 predicted optimization sequences from our model. Over all our benchmarks, using our model we can achieve 1%, 4%, and 9% speedups over -O3 using 1, 5, and 10 predicted optimization sequences, respectively. Consequently, our technique allows MiCOMP to outperform -O3 by high margins.

5.1 Comparison with Standard Optimization Levels

Standard optimization levels have been introduced to achieve good performance on average. However, they come short of the customized auto-tuning frameworks per architecture/application/dataset. As we showed in Table 9, MiCOMP can surpass the performance of -O3 with a few predictions on application bases. Here we provide Table 10 which reports more fine-grained speedup over all standard optimization levels. This demonstrates how fast (first number in the tuple) and in what percentage of the explored space (the second number), the framework is reaching a sequence which can outperform the specific standard optimization level. Each column is reporting two values: (i) in how many predictions and (ii) in what percentage of the whole configuration space the propped methodology can outperform OX levels.

Table 10. Average Speedup w.r.t LLVM -O3. Numbers are A (B%): **(A)** How fast (in terms of number of predictions) in average the proposed methodology outperforms LLVM standard Optimizations. **(B)** The percentage of the optimization space explored to satisfy the goal.

Predictive Modeling	-O1	-O2	-O3
MultilayerPerceptron	1 (0.01%)	1 (0.01%)	2 (0.016%)
LinearRegression	1 (0.01%)	1 (0.01%)	3 (0.02%)
KStar	1 (0.01%)	1 (0.01%)	2 (0.016%)

5.2 Comparison with State-of-the-art Iterative Compilation Models

In this section, we compare MiCOMP with two state-of-the-art intermediate-sequence prediction approaches proposed in [5, 24].

5.2.1 Intermediate Speedup Comparison Case. (A). Kulkarni et al., [24] used *Neuro-Evolution for Augmenting Topologies* (NEAT) to predict the best compiler optimization to apply given the state of source-code being optimized by the dynamic JIT Jikes RVM compiler. . Contrary to the technique we propose in this paper where we obtain features of the code only once before it is optimized. Kulkarni et al. used NEAT, a machine-learning framework based on genetic evolution, to generate many neural-networks where each network was evaluated on the task of using static source code features to predict the next compiler optimization to apply. NEAT can make optimization predictions to any given maximum-length to predict the most beneficial sequence of optimizations for the target application being compiled. In NEAT training time was reported around 10 days while the current approach requires a few hours to construct the model. Another advantage of the current work is the fact that it supports multiple predictions from the prediction-space while the NEAT approach can produce one-shot results based on the stop condition for each application and neural network configuration. We reproduced the work by Kulkarni et al. [24] by using 100 chromosomes and 500 generations on 12 core Xeon(R) CPU E5-1650 v2 @ 3.50GHz with 12GB running on Ubuntu and we report the result in Table 11. We ran NEAT in parallel with average running time of 1.75 hours per model (the longest took 4 hours). . The training and prediction is done with leave-one-out cross-validation in order to produce uniform results. .

5.2.2 Intermediate Speedup Comparison Case. (B). Ashouri et al. [5] demonstrated a predictive methodology to predict an intermediate speedup OF an optimization from the configuration space given the current state of the application.

However, unlike [24], they employed dynamic features of the application under study. As mentioned in Section 5.2.1, a major downside in an intermediate speedup approach is that application feature should be collected on every state by means feature extraction and this makes the system impractical on large-scale data, specially, when dynamic features are collected on every state. In addition to an efficient feature collection process and predicting the complete optimization sequence to apply to the unoptimized code at once, MiCOMP brings two extensions to the aforementioned work. Second, comparison baseline in [5] was LLVM's default optimization, while in this work we provide a comparison against LLVM's `-O3` (we show MiCOMP can outperform an aggressive optimization setting in LLVM, that is, `-O3`, in only a few predictions.). Figure 6 demonstrates the comparison. For this comparison, we used the same training data of up to the length of 4 (as it was declared in [5]) for both models to be uniform on both comparisons. We observe that, except the first two predictions, the proposed approach outperforms the intermediate speedup methodology reported in this work and on average MiCOMP brings 11% speedup gain.

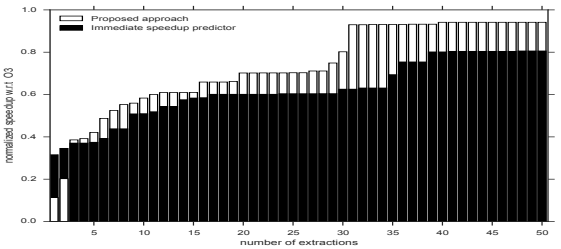


Fig. 6. Performance of MiCOMP w.r.t the performance of intermediate speedup predictor approach [5]

Table 11. Performance comparison of the single prediction by MiCOMP against the intermediate speedup approach reported in previous work [24]. All values are normalized by -O3.

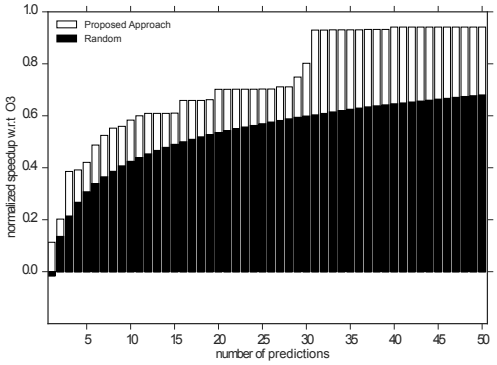
Application	NEAT		MiCOMP
	Best NN Size	1 prediction	1 prediction
automotive_qsort1	1105	1.0336	1.0385
automotive_bitcount	1536	1.0923	1.0898
automotive_susan_c	607	1.0012	1.0491
automotive_susan_e	613	1.0211	1.0481
automotive_susan_s	1295	1.0135	1.0195
bzip2d	1159	1.0514	0.9898
bzip2e	1259	1.0012	0.9798
consumer_jpeg_c	1327	1.0205	1.1882
consumer_jpeg_e	596	1.0712	1.0981
consumer_tiff2bw	1038	0.9522	0.9491
consumer_tiff2rgba	1147	0.9905	0.9295
consumer_tiffdither	612	1.0222	1.0288
consumer_tiffmedian	1356	0.9097	0.9497
consumer_lame	1612	1.0221	1.0288
consumer_mad	1256	0.9097	0.9497
network_dijkstra	1343	1.0353	1.1382
network_patricia	622	0.7971	0.8585
office_ispell	1056	0.9197	0.9897
office_ghostscript	1356	0.9097	1.0997
office_rsynth	1306	0.9797	1.0497
office_stringsearch1	858	0.9897	1.0397
telecom_adpcm_c	958	0.9754	1.0192
telecom_adpcm_d	948	0.9897	1.0232
telecom_gsm_d	924	0.9997	1.0397
telecom_CRC32	886	0.9423	1.0012
telecom_pgp_d	843	0.9697	1.0234
telecom_pgp_e	1002	0.9891	1.0254
security_sha	1536	1.0193	1.0178
security_blowfish_e	1221	1.0023	1.0298
security_blowfish_d	1534	1.0923	1.0898
security_rijndael_e	1132	1.0113	1.0395
security_rijndael_d	1033	1.0123	1.0598
Harmonic Mean		**0.9632**	**1.0295**

5.3 Comparison with Random Iterative Optimization

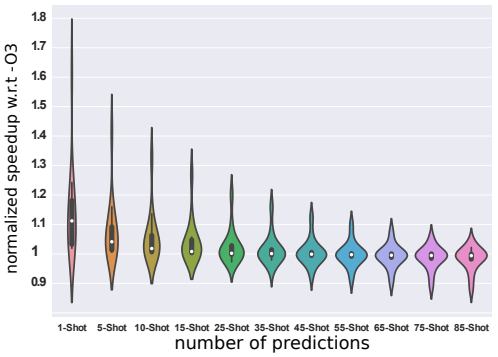
As we illustrated in Section 4.5.2, iterative compilation can improve application performance over standard compiler optimization sequences [1, 9]. Additionally, several published works have shown that drawing compiler optimization sequences at random can often be as good as using other more complicated search algorithms, such as genetic algorithms or simulated annealing [1, 10, 12]. In this section, we compare the effectiveness of MiCOMP to a Random Iterative Compilation (RIC) method that samples our clustered subsequences from a uniform distribution. We randomized the distribution of predictions 10000 times to make sure the obtained model is totally uniform. The purpose of this comparison is to show how effective the MiComp predictive modeling works against random iterative compilation. Our results are presented in Figure 7. To present our results, we define *Normalized Performance Improvement* (NPI) as the ratio of the performance improvement achieved over the potential performance improvement:

$$NPI = \frac{E_{ref} - E}{E_{ref} - E_{best}} \tag{5}$$

where E is the execution time achieved by the methodology under consideration, E_{ref} is the execution time achieved with a reference compilation methodology and E_{best} is the best execution time that can be obtained through an exhaustive exploration of all possible compiler optimization sequences in the optimization space we are exploring. As the execution time E of the iterative compilation methodology under analysis gets closer to the reference execution time E_{ref}, the value of NPI gets closer to 0, where 0 indicates no improvement was obtained. As E approaches the best execution time, E_{best}, the value of NPI approaches 1. An NPI value of 1 indicates that the optimal performance available was achieved. The goal of the evaluation in this section is to show how effective MiCOMP is at exploring the optimization sequence space compared to RIC. . The X axis pertains to the number of predicted optimization sequences used and the Y axis shows their corresponding speedup values. We used NPI (scaled within $[-\infty, 1]$) and the speedups are all normalized by



(a) Let both MiCOMP and random explore the opt. space



(b) Let only random explore the opt. space

Fig. 7. MiCOMP performance comparison versus random iteration compilation

`-O3` performance. Thus, $Y = 0$ is the speedup line corresponding to `-O3`. We observe that the performance of MiCOMP outperforms Random Iterative Compilation with fairly a clear margin on each number of predicted optimization sequences used. Table 7 gives the the absolute speedup values.

Figure 7b displays another result where we compare a fixed number of predicted optimization sequences for MiCOMP, that is 5 predicted sequences, versus different number of predicted sequences from RIC. That is we observe the prediction quality of MiCOMP compared to different numbers of predicted optimization sequences drawn from a random distribution. Figure 7b depicts this scenario using a violin plot where the Y axis pertains to the speedup with respect to the RIC and the X axis corresponds to the different predicted optimization sequences obtained from RIC. Statistically, we observe that the quality of the 5-prediction of MiCOMP is as good as using 35 prediction optimization sequences from RIC thus we observe that the predicted optimization sequences derived by MiCOMP can give up to 7× exploration speedup versus the RIC method.

6 CONCLUSION

We proposed and presented a clustering technique for all the compiler optimizations in LLVM's `-O3` and clustered them in five different optimization sub-sequences to speedup the training and exploration phase. This method helps us outperform LLVM's `-O3` optimization sequence. Moreover, MiCOMP has a simple encoding function that encodes an optimization sequence into a bit string, allowing us to apply standard machine learning techniques that require fixed length feature vectors. We incorporated analogies between the analyzed problem and the context of Recommender Systems, and integrate similarity measures to boost exploration efficiency. We show that MiCOMP can outperform LLVM's standard optimization levels with just a few predicted of optimizations sequences and achieves top 80% of the available speedup by traversing less than 5% of the optimization sequence space. .

REFERENCES

[1] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. 2006. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 295–305.

[2] Alan Agresti, I Liu, and others. 1999. Modeling a categorical variable allowing arbitrarily many category choices. *Biometrics* 55, 3 (1999), 936–943.

[3] Amir Hossein Ashouri. 2012. *Design space exploration methodology for compiler parameters in VLIW processors.* Master's thesis. Politecnico di Milano, Italy. http://hdl.handle.net/10589/72083.

[4] Amir Hossein Ashouri. 2016. *Compiler Autotuning Using Machine Learning Techniques.* Ph.D. Dissertation. Politecnico di Milano, Italy. http://hdl.handle.net/10589/129561.

[5] Amir Hossein Ashouri, Andrea Bignoli, Gianluca Palermo, and Cristina Silvano. 2016. Predictive Modeling Methodology for Compiler Phase-Ordering. In *Proceedings of 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 5th Workshop on Design Tools and Architectures for Multicore Embedded*

Computing Platforms. ACM. DOI: http://dx.doi.org/10.1145/2872421.2872424

[6] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. 2016. COBAYN: Compiler Autotuning Framework Using Bayesian Networks. *ACM Trans. Archit. Code Optim.* 13, 2, Article 21 (June 2016), 25 pages. DOI: http://dx.doi.org/10.1145/2928270

[7] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, and Cristina Silvano. 2014. A Bayesian network approach for compiler auto-tuning for embedded processors. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2014 IEEE 12th Symposium on*. IEEE, 90–97. DOI: http://dx.doi.org/10.1109/ESTIMedia.2014.6962349

[8] Amir Hossein Ashouri, Vittorio Zaccaria, Sotirios Xydis, Gianluca Palermo, and Cristina Silvano. 2013. A framework for Compiler Level statistical analysis over customized VLIW architecture. In *Very Large Scale Integration (VLSI-SoC), 2013 IFIP/IEEE 21st International Conference on*. IEEE, 124–129. DOI: http://dx.doi.org/10.1109/VLSI-SoC.2013.6673262

[9] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. 1998. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*.

[10] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. 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, USA, 185–197. DOI: http://dx.doi.org/10.1109/CGO.2007.32

[11] John Cavazos and J Eliot B Moss. 2004. Inducing heuristics to decide whether to schedule. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 183–194.

[12] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. 2012. Deconstructing iterative optimization. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 3 (2012), 21.

[13] Keith D Cooper, Philip J Schielke, and Devika Subramanian. 1999. Optimizing for reduced code space using genetic algorithms. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 1–9.

[14] Grigori Fursin. 2010. Collective benchmark (cbench), a collection of open-source programs with multiple datasets assembled by the community to enable realistic benchmarking and research on program and architecture optimization. (2010).

[15] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.

[16] Torsten Hoefler and Roberto Belli. 2015. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 73.

[17] Kenneth Hoste and Lieven Eeckhout. 2007. Microarchitecture-independent workload characterization. *IEEE Micro* 27, 3 (2007), 63–72.

[18] Rob J Hyndman and Anne B Koehler. 2006. Another look at measures of forecast accuracy. *International journal of forecasting* 22, 4 (2006), 679–688.

[19] Michael R Jantz and Prasad A Kulkarni. 2013. Exploiting phase inter-dependencies for faster iterative compiler optimization phase order searches. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*. IEEE, 1–10.

[20] Kevin J Johnson and Robert E Synovec. 2002. Pattern recognition of jet fuels: comprehensive GC× GC with ANOVA-based feature selection and principal component analysis. *Chemometrics and Intelligent Laboratory Systems* 60, 1 (2002), 225–237.

[21] Richard Arnold Johnson, Dean W Wichern, and others. 1992. *Applied multivariate statistical analysis*. Vol. 4. Prentice hall Englewood Cliffs, NJ.

[22] Toru Kisuki, Peter M. W. Knijnenburg, and Michael F. P. O'Boyle. 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), Philadelphia, Pennsylvania, USA, October 15-19, 2000*. 237–248. DOI: http://dx.doi.org/10.1109/PACT.2000.888348

[23] Prasad A Kulkarni, David B Whalley, Gary S Tyson, and Jack W Davidson. 2009. Practical exhaustive optimization phase order exploration and evaluation. *ACM Transactions on Architecture and Code Optimization (TACO)* 6, 1 (2009), 1.

[24] Sameer Kulkarni and John Cavazos. 2012. Mitigating the compiler optimization phase-ordering problem using machine learning. *ACM SIGPLAN Notices* 47, 10 (2012), 147–162.

[25] David B Loveman. 1977. Program improvement by source-to-source transformation. *Journal of the ACM (JACM)* 24, 1 (1977), 121–145.

[26] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. DOI: http://dx.doi.org/10.1145/1065010.1065034

[27] Luiz GA Martins, Ricardo Nobre, Joao MP Cardoso, Alexandre CB Delbem, and Eduardo Marques. 2016. Clustering-Based Selection for the Exploration of Compiler Optimization Sequences. *ACM Transactions on Architecture and Code*

Optimization (TACO) 13, 1 (2016), 8.
[28] Antoine Monsifrot, François Bodin, and Rene Quiniou. 2002. A machine learning approach to automatic production of compiler heuristics. In *Artificial Intelligence: Methodology, Systems, and Applications*. Springer, 41–50.
[29] J Eliot B Moss, Paul E Utgoff, John Cavazos, Doina Precup, Darko Stefanovic, Carla Brodley, and David Scheeff. 1997. Learning to schedule straight-line code. In *NIPS*, Vol. 97. 929–935.
[30] Ricardo Nobre, Luiz GA Martins, and João MP Cardoso. 2015. Use of Previously Acquired Positioning of Optimizations for Phase Ordering Exploration. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. ACM, 58–67.
[31] Ricardo Nobre, Luiz GA Martins, and João MP Cardoso. 2016a. A graph-based iterative compiler pass selection and phase ordering approach. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*. ACM, 21–30.
[32] Ricardo Nobre, Luıs Reis, and Joao MP Cardoso. 2016b. Compiler Phase Ordering as an Orthogonal Approach for Reducing Energy Consumption. In *Proceedings of the 19th Workshop on Compilers for Parallel Computing (CPC2016), Valladolid, Spain, July 6–8, 2016*.
[33] Holger Orup. 1999. On-the-fly one-hot encoding of leading zero count. (Oct. 26 1999). US Patent 5,974,432.
[34] EunJung Park, John Cavazos, and Marco A Alvarez. 2012. Using graph-based program characterization for predictive modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 196–206.
[35] Eunjung Park, John Cavazos, Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and P Sadayappan. 2013. Predictive modeling in a polyhedral optimization space. *International Journal of Parallel Programming* 41, 5 (2013), 704–750.
[36] Eunjung Park, Sameer Kulkarni, and John Cavazos. 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*. ACM, 65–74.
[37] Suresh Purini and Lakshya Jain. 2013. Finding good optimization sequences covering program space. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 56.
[38] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*. ACM, 285–295.
[39] Satu Elisa Schaeffer. 2007. Graph clustering. *Computer Science Review* 1, 1 (2007), 27–64.
[40] Steve Smale and Ding-Xuan Zhou. 2003. Estimating the approximation error in learning theory. *Analysis and Applications* 1, 01 (2003), 17–41.
[41] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. 2003. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 77–90. DOI: http://dx.doi.org/10.1145/781131.781141
[42] Steven R. Vegdahl. 1982. Phase coupling and constant generation in an optimizing microcode compiler. *ACM SIGMICRO Newsletter* 13, 4 (1982), 125–133.
[43] Wei Zhang, Deli Zhao, and Xiaogang Wang. 2013. Agglomerative clustering via maximum incremental path integral. *Pattern Recognition* 46, 11 (2013), 3056–3065.