MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning

AMIR H. ASHOURI, University of Toronto, Canada
 ANDREA BIGNOLI, GIANLUCA PALERMO, and CRISTINA SILVANO,
 Politecnico di Milano, Italy
 SAMEER KULKARNI and JOHN CAVAZOS, University of Delaware, USA

Recent compilers offer a vast number of multilayered optimizations 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. Finding the best ordering represents a long standing problem in compilation research, named the *phase-ordering* problem. The traditional approach of constructing compiler heuristics to solve this problem simply cannot cope with the enormous complexity of choosing the right ordering of optimizations for every code segment in an application.

This article proposes an automatic optimization framework we call MiCOMP, which Mitigates the Compiler Phase-ordering problem. We perform phase ordering of the optimizations in LLVM's highest optimization level using optimization sub-sequences and machine learning. The idea is to cluster the optimization passes of LLVM's O3 setting into different clusters to predict the speedup of a complete sequence of all the optimization clusters instead of having to deal with the ordering of more than 60 different individual optimizations. The predictive model uses (1) dynamic features, (2) an encoded version of the compiler sequence, and (3) an exploration heuristic to tackle the problem.

Experimental results using the LLVM compiler framework and the Cbench suite show the effectiveness of the proposed clustering and encoding techniques to application-based reordering of passes, while using a number of predictive models. We perform statistical analysis on the results and compare against (1) random iterative compilation, (2) standard optimization levels, and (3) two recent prediction approaches. We show that MiCOMP's iterative compilation using its sub-sequences can reach an average performance speedup of 1.31 (up to 1.51). Additionally, we demonstrate that MiCOMP's prediction model outperforms the -O1, -O2, and -O3 optimization levels within using just a few predictions and reduces the prediction error rate down to only 5%. Overall, it achieves 90% of the available speedup by exploring less than 0.001% of the optimization space.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computing methodologies** → **Machine learning**; • **Information systems** → *Recommender systems; Clustering and classification*;

Additional Key Words and Phrases: Autotuning, phase-ordering, optimizations, supervised-learning

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

Authors' addresses: A. H. Ashouri, ECE, 10 King's College Rd, Toronto, ON M5S 3G8, Canada; email: aashouri@ece.utoronto.ca; A. Bignoli, G. Palermo, and C. Silvano, DEIB, Via G. Ponzio 34/5, Milan, Italy; emails: andrea.bignoli@mail.polimi.it, {gianluca.palermo, cristina.silvano}@polimi.it; S. Kulkarni and J. Cavazos, EECIS, Newark, DE 19716, USA; emails: {sameer, cavazos}@udel.edu.

Permission to make digital or hard copies of all or part 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 bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM 1544-3566/2017/09-ART29 \$15.00

<https://doi.org/10.1145/3124452>

ACM Reference format:

Amir H. 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 Trans. Archit. Code Optim.* 14, 3, Article 29 (September 2017), 28 pages. <https://doi.org/10.1145/3124452>

1 INTRODUCTION

Compiler developers design optimization passes to transform each code segment of a program to generate an optimized version of an application. Code segment can be a basic block, a function, a source file, or the whole program. The optimizations can be applied at different stages of the compilation process. Optimizing source code by hand is a tedious and long task; therefore, compiler optimizations are provided to automatically transform the code. However, code optimizations are dependent on programming language, application, and architecture. Additionally, the word optimization is a misnomer and there is no guarantee the transformed code will perform better than the original version. Aggressive optimizations might even degrade the performance of the code. Understanding the behavior of the optimizations, their actual effects on the source code, and the interaction of the optimizations to each other 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 generates the phase-ordering problem. The phase-ordering problem has been an open problem in the field of compiler research for many decades [32, 56]. The inability of researchers to solve the phase-ordering problem has led to significant advances in the simpler problem of selecting the right set of optimizations, but even this problem has yet to be solved [8, 12, 15].

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 on the interaction between the preceding compiler optimizations in the sequence. 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's `clang` and LLVM's `opt` 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`, to introduce a fixed sequence of compiler optimizations that, on average, can 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 article, we propose the MiCOMP framework 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 the code being optimized:

- (1) *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 [6, 30].
- (2) *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. Although a number of works have used complete sequence prediction to select the best compiler optimizations [8, 13, 44–46], there has been no recent work using this technique to tackle the phase-ordering problem.

The framework proposed in this article, MiCOMP, belongs to the second category. Our work represents the first attempt to use machine learning models for predicting the speedup of a complete optimization sequence 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, provides 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 off line; program features and different compiler configurations are fed as inputs. As outputs, a prediction model generates a speedup number without the need to actually run the code on the target architecture. 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 [49] 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 state-of-the-art sorting/ranking [13, 44–46]. In our experimental results, we show that our technique outperforms LLVM’s highest optimization level of -O3 within just a few predictions (up to three in the worst observed case). We also show quantitative comparisons with respect to state-of-the-art techniques. We select the full set of applications from the Ctuning Cbench benchmark [19] to assess and evaluate the benefits of the proposed approach and to prove its feasibility. The main contributions of the proposed approach are:

- We propose 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 model. We use existing algorithms to create a custom machine learning framework for mitigating the long-standing phase-ordering problem in compilers. Thus, MiCOMP represents an independent predictive-modeling framework.
- We clustered different compiler optimizations (all taken from LLVM’s O3) into five 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 optimization (as baseline) and exploring different orderings of the sub-sequences using the same optimizations available to -O3, we outperformed -O3.
- We adapted a simple mapping technique to encode an optimization sequence into a fixed-length bit string. The proposed technique transforms a variable-length representation to a fixed-length feature vector representation, enabling us to apply traditional machine learning algorithms, since they are mostly designed to cope with fixed-length feature vectors.
- We adapted a Recommender System (RS) approach to speed up the search for good sequences of optimizations with respect to a state-of-the-art ranking approach. We enhance the search not only by considering already predicted speedups, but also by increasing diversity with respect to previously analyzed sequences. Thus, the proposed methodology offers better recommendations since it explores a larger space than previous state-of-the-art approaches, while reducing the risk of falling into local minima.

The rest of the article is organized as follows. Section 2 presents the related work, while Section 3 introduces our proposed methodology and its components. In Section 4, we present the experimental results and we assess the results by means of several comparisons in Section 5. We conclude the article with some ideas for future work and conclusions.

2 RELATED WORK

Literature [7] 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: (1) Autotuning and iterative compilation approaches and (2) Applying machine learning to the problem of optimization selection.

Autotuning addresses automatic code-generation and optimization by using different scenarios and architectures. It deals with building techniques for automatic optimization of different parameters in order to maximize or minimize the satisfaction of an objective function. An autotuning strategy consists of coupling the approach with a random generation of code variants at each run. This technique can generally improve application performance in reference to static-hand-crafted compiler optimization sequences [2]. Early works investigated the applicability of *iterative compilation* on embedded applications to outperform hand-craft optimizations [1, 12]. The authors explored the non-linear transformation space and found the fastest execution time in a fixed number of evaluations. In this approach, they used profile feedback in the form of execution time that searched a large but restricted transformation space to find better results. They achieved a Top-30% speedup by exploring 0.25% of the optimization space. We will show that MiCOMP's prediction model speeds up the iterative search and achieves a Top-10% of the available speedup by exploring less than 0.001% of the optimization space. Given the complexity of the iterative compilation problem, 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 [2, 13, 15]. Other authors [4, 10] explored compiler Design Space Exploration (DSE) techniques jointly with architectural DSE for Very Long Instruction Word (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 [14, 18, 27, 35, 36, 53] were among the first notable works introducing the use of machine learning to solve compilation problems. Recent related work [5, 8, 9, 44, 45] 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)). Different objective functions have been used with machine learning on the problem: (1) 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. [13, 44, 45] (2) 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 [2, 5, 8, 47]. (3) A *tournament predictor* [46] 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 first optimization sequence will be more or less than the speedup given by applying the second optimization sequence.

The phase-ordering problem still remains a challenging research problem. However, there are a few notable works that attempted to solve the problem. Triantafyllis et al. [55] proposed a generic optimizer that used practical iterative compilation framework called Optimization-Space

Exploration (OSE). It uses the compiler writer's knowledge to select a small number of promising optimization alternatives for a given code segment, thereby mitigating compile time.

Almagor et al. [3] presented results, which characterize the space of optimization interactions. These results showed that the space was highly discontinuous, but also that the space featured many local maxima, all of them in a reasonable bound of the global maximum. This work suggested search and pruning heuristics to reduce the fraction of the space to search. The authors' findings holds in the context of MiCOMP. We experimentally observed that many local maxima are discontinuous, which was one of the reasons we employed our recommender system exploration policy so as to alleviate this issue. It is doing so by taking advantage of dynamic information at prediction time. Thus, by biasing the exploration of the optimization space, more promising areas could be covered leading to global maxima (highest speedup values) in fewer steps.

Cooper et al. [17] introduced Adaptive Compilation Made Efficient (ACME), which attempted to reduce the per-sequence cost of searching through the space of optimization interactions by employing a virtual execution technique. In virtual execution, profiling information is conservatively propagated through program optimizations, so that the quality of the transformed code could be estimated without re-running the application. Additionally, they experimented with several different search algorithms and a user-friendly interface. This work [17] has major differences with respect to our work. First, unlike most autotuning works, which use timing measurements, it uses instruction counts as performance measure. Although it does not vary from run to run, it limits the use of some compiler optimizations since the performance on modern architectures is heavily dependent on the behavior of the memory subsystem. Second, the final framework is restricted by using predefined pairwise orders. Thus, ACME can not be used directly to tackle a realistic phase-ordering problem. Finally, ACME falls short in transferring knowledge on a cross-application basis.

Tate et al. [54] presented a formal approach of structuring optimizers that obviates the need to worry about optimization ordering by enabling the use of a global optimization heuristic that selects among fully optimized programs and can be used to perform translation validation on a cross-optimization basis. The authors experimentally proved that their proposed engine can quickly reach its saturation mode even without employing their exploration heuristic. By the proposed method, they removed the burden of optimization ordering in a set of compiler optimizations they formalized. Our Machine Learning (ML)-based approach is more general and can be adapted to any compiler framework that supports phase-ordering. In addition, the work in Tate et al. [54] does not take into account different dataset inputs. As suggested by Chen et al. [15], optimal compiler optimizations can be correlated with the type of dataset an application uses.

Kulkarni and Cavazos [30] have applied Neuro-Evolution for Augmenting Topologies (NEAT) in the Java Jikes Research Virtual Machine (RVM) compiler to phase ordering using intermediate sequence prediction. They built prediction models that received as input, features of the current state of the transformed source-code. The authors defined certain stop-condition rules to complete the final predicted sequence at each iteration. They used source code features and the Java JikesRVM Just-in-time (JIT) compiler to experimentally evaluate their approach. In contrast, we tackle the problem by using predictive modeling and dynamic independent characterization of the applications. The proposed methodology enables us to predict the full sequence in one shot.

Matrins et al. [34] 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 algorithms. Our proposed approach, on the other hand, is quite 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 [29]. 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. [24] 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. [6] introduced an approach that uses predictive modeling to construct an intermediate sequence of optimizations for code being compiled. Other related work used iterative DSE and clustering-based approaches to down-sample and cluster the available optimizations targeting performance gain and power reduction [38, 40, 41].

Our approach, MiCOMP, is significantly different with respect to these works mentioned in literature. Our work mostly resembles the approach of Park et al. [45, 46]. However, our technique tackles the significantly harder phase-ordering problem. We introduce an encoding 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 [6, 30]. An intermediate sequence approach needs multiple rounds of profiling of the application being optimized (based on the characteristics of the code in its present state). Our approach needs just one round of application profiling in both offline-training and online-prediction phases. We used dynamic 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 machine learning approach to mitigate the phase-ordering problem by predicting the speedup of a *complete sequence* of optimizations. In Section 4, we improve the machine learning model through Recommender System techniques and we provide some experimental results to compare with 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 one 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 -O3 (highest available standard optimization level), to achieve the maximum speedup for all the applications being optimized. We form sub-sequences of -O3's optimization sequences by using a clustering technique and found that if we could reorder sub-sequences of optimizations that tended to perform well, we could significantly outperform the single static optimization sequence obtained by -O3.

In particular, the proposed approach customizes the optimization process based on the features derived from the application under analysis. To mitigate the phase-ordering problem, a model is built to correlate the effect of using different compiler sequences and the corresponding achievable speedup. The model is then used by MiCOMP to recommend good sequences of optimizations

Fig. 1. Proposed framework: (1) offline training phase which is done once; (2) online-prediction phase for optimizing new unseen applications.

that maximize an application's performance with very few predictions. Moreover, the phase-ordering problem is complicated by the possibility of having variable-length compiler sequences. State-of-the-art approaches for selecting the right set of optimizations used fixed-length feature vectors [8, 13, 45, 46] to induce a prediction model. To tackle this problem, we propose a method to encode the phase-ordering space into a conventional fixed-length feature vector space to apply traditional machine learning algorithms and build the previously mentioned 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. Our heuristic, derived from the recommender system field, is able to predict optimization sequences outperforming state-of-the-art *ranking* [45, 46] techniques.

Figure 1 illustrates the two main phases of MiCOMP: (1) *offline training* and (2) *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, or target architecture, 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 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., Principal Component Analysis (PCA) [26]). 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 is then used during the online phase to drive the selection of the best set of compiler optimization sequences.

The *online prediction* phase is used every time a new application needs to be 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. Since the goal of our method is to discover the best sequence of compiler optimizations using the fewest number of trials, we coupled the prediction model with a heuristic derived from the field of Recommender Systems (see Section 3.5). This technique is used to obtain a predicted set of promising optimization sequences for which the diversity of the solution is preserved, thus guaranteeing the coverage of a large part of the

optimization configuration space, consequently obtaining a set of optimization sequences that are robust with respect to model inaccuracies.

3.1 Application Characterization

In this work, we used a PIN-based [33] dynamic instrumentation framework to analyze and characterize the behavior of applications at the execution-time. In particular, our framework provides a high level Micro-architectural Independent Characterization of Applications (MICA) [22] 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. In our experimental setup, an application is compiled and profiled on an Intel Xeon machine and the machine learning model is fed by a high-level abstraction of the application characterization carried out with MICA, as suggested in Hoste and Eeckhout [22].

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). Furthermore, generating a predictive model is a process whose time complexity grows with the number of parameters in use and at the same time, including all features that might lead to higher noise. To significantly improve the speed of model construction, we applied a dimension reduction by using PCA [26] 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 [25]. 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 [22]. In particular, we set $k = 5$, which captures more than 98% of the overall variance across all training data. In PCA, output values are no longer directly representing a certain measured feature since they are a combination of the observed features [25], used to explain a certain phenomenon. Thus, the output of the PCA analysis is nothing but a five-dimensional vector of weights (or loadings). There is no way to indicate output components by name, since they are not directly observable. These five principal components are derived by the extracted 99 MICA framework features that have been used as a signature for characterizing the target applications.

3.2 Constructing Compiler Sub-Sequences

In this section, we briefly explain our idea behind clustering certain compiler optimizations as *sub-sequences* to scale down the phase-ordering problem, while maintaining the quality of the solution. A phase-ordering optimization sequence represented by the vector 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 with a fixed-length optimization sequence length and no repetitive application of optimizations. Allowing optimizations to be repeatedly applied and allowing a variable-length sequence of optimizations will expand the problem space to:

$$|\Omega_{phases_repetition}| = \sum_{i=0}^m n^i, \quad (1)$$

where n is the number of optimizations under study and m is the maximum desired length for the optimization sequence. Even for reasonable values of n and m , the entire search space is huge. For example, assuming n and m are both equal to 10, this leads to an optimization search space of more than 11 billion different optimization sequences to select from for each piece of code being optimized [6].¹

3.2.1 The Optimization Dependence Graph. Tackling 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, thus reducing the size of the search space to explore. This additionally introduces scalability on a larger number of compiler passes available in recent compiler frameworks. Certain analysis algorithms typically should be done before an optimization in order for the optimization to have any significant impact. For example, we might 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 LLVM's -O3 will contain optimizations that should follow other optimizations in order to obtain the best performance.

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 these 157 compiler passes, some of them are analysis passes (i.e., basicaa, memdep, etc.) that 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, and the like, which perform optimizations on the code [31]. In this article, we apply clustering on all the passes available at LLVM's -O3 to form our sub-sequences. Later, we will experimentally demonstrate that this technique, despite reducing the search space, can easily outperform LLVM's -O3, when it is paired with a predictive model to dynamically reorder the clusters into good optimization sequences.

Let $\sigma = \{o_1, \dots, o_N\}$ be the set of all transformation passes from LLVM optimization level -O3. We can represent the optimizations in optimization level -O3 with a directed graph of $G = (V, E)$, where V is the set of nodes representing the optimizations and E is the set of edges, where we add a directed edge between two optimization nodes if that pair of optimizations consecutively appears in the optimization sequence in -O3. Further, we can annotate an edge with the number of times that each pair of optimizations consecutively appears in the sequence. This resulted graph is called *optimization dependency graph*. The optimization dependency graph is then translated in a *Weighted Adjacency Matrix* used to cluster the set of optimizations into different sub-sequences. Figure 2 shows the constructed graph on LLVM -O3.

3.2.2 Graph and Sub-Sequence Clustering. Constructing the right set of optimization sub-sequences can have a great impact on the quality of the results. Thus, here we pay much consideration to define an easily reproducible process to construct our sub-sequences. We decided not to have a manual clustering approach ([6, 39]), since we found that it requires detailed knowledge on each optimization pass and also on the interaction among the passes. It would make it difficult to port across other compilers or versions of the same compiler. On the other hand, automated black box clustering such as random clustering was not considered since it does not exploit the domain knowledge available in our case. For these reasons, we decided to use *agglomerative clustering*.

¹The problem of phase-ordering does not have a deterministic upper-bound in the case of having unbounded length.

Fig. 3. An example of the proposed encoder when we encode an original version of $N = 5$ optimization subsequences (Top), while having an upper bound sequence length of $M = 6$ (Bottom). Each letter represents a sub-sequence.

There is a simple way of encoding the space $\mathcal{W}$ of all words of length at most M (thus, $k \leq M$) 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 $f : \mathcal{A} \rightarrow \{0, 1\}^N$, which encodes each letter α_i to the binary string $f(\alpha_i) = b_1 \cdots b_N$, where $b_j = 1$ if $j = i$, and $b_j = 0$ otherwise. Now we define the mapping function $F : \mathcal{W} \rightarrow \{0, 1\}^{N \times M}$ by encoding each word $\alpha_{i_1} \alpha_{i_2} \cdots \alpha_{i_k}$ to the binary string

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

where $0 = 0 \cdots 0$ is the zero string of length N . 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. Indeed, the image $F(\mathcal{W})$ can be simply characterized by two requirements: (1) each $f(\alpha_{i_t})$ has at most one non-zero binary digit; (2) if $f(\alpha_{i_t}) = 0$ and $f(\alpha_{i_v}) \neq 0$, then $t > v$. Summarizing, the proposed encoding is composed of the bijective function F that preserves the characteristics of the original words by solving the problem of categorical variables and mapping them to fixed-length binary strings. An example of the proposed encoding scheme is shown in Figure 3.

3.4 Predictive Modeling

Similar to any traditional supervised learning [28] workflow, we start by collecting data over a set of applications followed by a training process to construct a model. This model is subsequently used to induce a prediction for a compiler sequence's speedup without actually running that code. MiCOMP adopts a predictive model to automatically relate the effects of compiler optimization sequences over unseen applications. In particular, the input of the model receives two segments: (1) a vector of dynamic features collected through instrumentation of an application once passed through our dimension reduction technique (see Section 3.1), and (2) an encoded version (see Section 3.3) of a compiler sequence composed of our proposed clustered sub-sequences (see Section 3.2). The output represents the corresponding predicted speedup of that compiler sequence over LLVM's -O3 speedup. Here, we preferred not to restrict our proposed methodology to a specific predictive modeling, but rather aimed at defining a more general autotuning framework for recent compilers. To this end, MiCOMP is implemented as an independent predictive modeling autotuner that can be paired with any desired model. In the experimental results section, we show the outcome of adopting different predictive models.

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 (1) in the offline training phase to construct the model and (2) in the online prediction phase to 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. To avoid the overfitting problem, MiCOMP uses the leave-one-out cross-validation (LOOCV).

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 the characterization of a state from which to predict the outcome. In this article, the state being characterized is the code being optimized, and the predicted outcome corresponds to the speedup metric, which is 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. The output of the model is a prediction the speedup T should achieve when applied to the original code.

3.4.2 Application-Specific Prediction. Our machine learning constructed models can be used for unseen target applications to predict the speedup of a compiler sequence. The predicted speedup values correspond to the optimization sequence applied to the program. For a given input program, a feature vector containing dynamic instrumentation is first collected. Then, our prediction model is fed by the features of the program being compiled to predict the expected speedup if an optimization sequence T was applied to it. By predicting the performance of each possible optimization sequence that can be applied, the optimization sequences can be ranked according to their expected speedup.

A state-of-the-art ranking approach [44, 46] was used to rank optimization sequences in descending order and select the top N optimization sequences to evaluate their actual optimization quality. However, 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. The details of the heuristic are further discussed in the next section.

3.5 Recommender System Heuristic

Employing an efficient exploration strategy on the vector of predicted speedups is of great importance, especially when the space is rather large. An efficient exploration strategy can recommend better solutions faster, therefore, converging to an optimal solution in a fewer number of iterations. This is referred to as *recommendations* [48, 49] in the recommender systems field.

In the initial steps taken by Ashouri et al. [6] and Kulkarni and Cavazos [30], the authors defined iterative exploration heuristics, based on the current optimized state of the target application, to select the next best optimization to apply. By iteratively exploiting the prediction algorithm, this can lead to a better 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 (see Section 5.3). In this article, 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.

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. However, none of them make use of *dynamic* information to 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 proposed technique leverages the *similarity* between the unexplored and the explored optimization sequences. MiCOMP prioritizes the evaluation of promising solutions that are 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 guaranteeing a larger coverage of the optimization space. Moreover, this also reduces the risk of falling in to local minima determined by possible prediction model inaccuracies. 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 application p in the set of applications P . We define an iterative process to look for predicted similarities in i and j when they were applied to different applications in P .

Adjusted Cosine Similarity. In 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 differences in the rating scale are not taken into account. The Adjusted Cosine Similarity (ACS) offsets this drawback by subtracting the corresponding user-average from each co-rated pair and it is proved to have the lowest error-rate among the different similarity measurement techniques [49]. Adapting this technique, we get these co-rated pairs from the speedup values of a specific pair of compiler optimization sequences i and j across all training data. Using this method, we can compute the ACS between optimization sequence i and j as:

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

where $S_{p,i}$ is the speedup achieved by sequence i when applied to program p of all sets 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 the one already tested in the current iteration, skip it. (3) If the end of the list has been reached and there is still room for optimization sequences to test, exclude from the entire list the solutions already tested and go to (2) starting a new iteration.

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. This way, ACS biases the way exploration is done and covers promising points within the prediction space quicker than it would have achieved by the state-of-the-art pure ranking approach [45, 46], thus achieving better speedups with fewer exploration steps.

4 EXPERIMENTAL RESULTS

In this section, we evaluate our proposed methodology on an Intel Xeon architecture.² We adapted our architecture-independent instrumentation tool (see Section 3.1) to extract characteristics from a large set of benchmarks from the Ctuning CBench suite v1.1 [19], composed of 32 different applications ranging from automotive, security, office, and telecommunication.

We have used LLVM compilation framework v3.8 (clang for the frontend/backend and opt for applying the optimization passes in IR). Table 1 illustrates the list of different compiler optimizations that are clustered into five different *sub-sequences*. All optimization passes are derived from LLVM's -O3 by applying the clustering methodology described in Section 3.2. The optimizations are fixed within a sub-sequence, but sub-sequences are allowed to appear in any order in the full optimization sequence. It is worth mentioning that adding a new optimization, i.e., one outside the

²More precisely, the target architecture is a 12 core Xeon(R) CPU E5-1650 v2 @ 3.50GHz with 12GB memory running Linux Ubuntu OS.

Table 1. Candidate Clusters of Compiler Optimizations into Sub-sequences
(All Derived from LLVM -O3)

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

current -O3 passes, can potentially alter the optimizations' clusters and the training data. Thus, model construction has to be redone. *Active learning* [51] is a recent branch of machine learning which tries to alleviate the problem of continuous learning by interactively querying the information source to obtain the desired output. However, it is outside the scope of this article.

The application execution time has been estimated by using the Linux *Perf* tool. Cbench provides a way to control the number of times an application should be executed. This feature is called loop-wraps. The execution time is calculated 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 binary has been executed nine times as three sets of three loop-wraps to ensure better accuracy of estimation and fairness among the generation of executions. To validate this claim, we have calculated the standard deviation of the samples and the extracted results reveal that the standard deviation of the measured data is less than 1% across each applications' loop-wraps.

As described in Section 3.1, we applied a PCA to generate a 5-D vector of the features (a single-dimensional vector of length 5), which captures 98% of the variance available in the training set. The Principal Components (PCs) have been computed using the MICA features collected from application executions (we only need to run the application once), normalized by standard deviation across all data sets. MiCOMP uses dynamic feature collection and can handle different input datasets as well. A new input dataset changes the dynamic features of an application under analysis and its corresponding prediction results will be different. However, here, each application was used with only one input dataset. As a side note, MiCOMP uses the leave-one-out cross-validation method; therefore, an application under analysis plus any of its future input datasets will be excluded from the training set and this will ensure not predicting an already seen application.

The proposed methodology is independent from the prediction model, and we report the results by using three different models described in Table 2. WEKA machine learning tool [20] has been integrated in our framework to test the different speedup predictors. To this end, we used (1) a Linear Regression (LR) classifier by using the M5 attribute selection method with default ridge parameter, (2) a *Multilayer Perceptron*, and (3) K^* algorithm using default settings.

4.1 Analysis of Selecting the Compilation Baseline

MiCOMP does not use any of the default compilation optimization levels as a baseline to start, since it uses all compiler optimization passes within LLVM's -O3 for its clustering purpose. We

Table 2. List of the Predictive Models Used in the Experiments

Predictive Model	Description
Multilayer Perceptron (MLP)	An MLP [43] is a feed-forward artificial neural network model that maps sets of input data onto a set of appropriate outputs. It consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one. It consists of multiple layers of simple, two-state, sigmoid processing elements (nodes) or so-called neurons that interact using weighted connections. No interconnections exist within a layer but all neurons in a layer are fully connected to neurons in adjacent layers. Weights can measure the correlation degree between the activity levels of neurons that they connect.
LinearRegression (LR)	LR [37] is an approach for modeling the relationship between a scalar dependent variable y and one or more explanatory variables (or independent variables) denoted X . Linear models such as LR are algorithms whose output classifier does not yield dramatic fluctuation against minor changes in the training set. In linear regression, the relationships are modeled by using linear predictor functions whose unknown model parameters are estimated from the data.
KStar	K^* [16] are instance-based learning algorithms that predict the classification of new instances based on instances already classified in the memory. K^* uses entropy as a distance measure. It assumes that similar instances belong to a similar class.

Note that the Proposed Methodology is Independent from any Specific Machine Learning Algorithm and it can be Paired with any Algorithm Desired.

found that starting with a standard compilation sequence as a baseline (i.e., -O2 or -O3) ultimately reduced the speedups achieved by the framework. As we construct prediction models, we want to make sure that the speedup achievable by a sequence is maximized. We empirically justify this argument by running a set of experiments: one not starting from a standard optimization level as a baseline and another where we use a sequence of optimizations to apply on top of different -Ox baselines. Figure 4 illustrates the mean speedup of MiCOMP's proposed compiler optimization sequences with different compilation baselines. In all the four independent experiments, each Cbench application has been optimized with the same 5,000 generated compiler sequences derived by the 5 clustered sub-sequences of MiCOMP. The X axis shows the applied compiler sequences sorted by their actual speedup, and the y shows their corresponding speedup.

Results suggest that using the MiCOMP optimization sequences without an optimization level as a baseline can lead to substantial benefits compared with using any of -Ox optimization levels as a baseline. We call this a region of interest. MiCOMP targets the region of interest shown in Figure 4, where the highest values for achievable speedup are located. Note that using a baseline of -O1, -O2, or -O3 converges to a sub-optimal speedup. The best option was not to use a baseline sequence at all, but let MiCOMP sub-sequences drive its own iterative compilation. By doing so, MiCOMP sub-sequences can iteratively get re-ordered to achieve the highest achievable speedup by an exhaustive search. By using machine learning, we could add intelligence on the model and focus on the way the iterative compilation is generated. Additionally, employing our recommender system based policy biases the prediction model to reach that region of interest with the least number of predictions.

Fig. 4. Empirical analysis of different compilation baselines for all CBench applications (Harmonic mean). The region of interest represents where MiCOMP sub-sequences outperform other compiler sequences.

Summarizing, the insights of this experiment are threefold: (1) The clustering technique is beneficial; first, to gain better speedup values without the need to have an optimization baseline, and second, to reduce the number of optimization variables needed to achieve the best results from more than 60 to 5 so that our iterative compilation problem can be easier when we employ machine learning models. (2) 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. (3) Phase-ordering does matter in the field of compilers: given the same set of optimization flags available to -O3, MiCOMP can significantly outperform -O3 itself.

4.2 Analysis of MiCOMP's Training Data Dependency

MiCOMP's results are computed by means of LOOCV by using 32 applications of Cbench. It constructs 32 independent prediction models where 31 applications were used for training and the excluded application was used for testing. As suggested in Banko and Brill [11], the prediction accuracy of a supervised learning methodology can be affected by the paucity of data. To this end, we have setup an analysis where we train MiCOMP by using only one Cbench category (all applications within one category) and we test using a different category to see how the quality of prediction can be adversely affected. With this experiment, the number of applications presented in a training set is reduced by a factor of roughly six, as Cbench has six different categories. Table 3 reports the result of this experiment. We have randomly selected the *Automotive* category for our training set and the applications in each of the five remaining categories were used for testing. We used *harmonic* mean (as suggested by Hoefler and Belli [21])³ over the calculated speedup values of every application under a single category and we report them in the third column. As anticipated, a training set that is six times smaller has decreased the prediction accuracy of MiCOMP by around 2–6% depending on the category. Moreover, another set of independent experiments has been carried out where we train using five categories of Cbench while excluding the sixth solely for testing purposes. We show the results in Table 3.

³We 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 3. Analysis of Training Data Dependency When Different Application Categories Are Used

Scenario		Speedup w.r.t. MiCOMP's default
Training	Testing	
Automotive	Security	0.9443
Automotive	Telecom	0.9643
Automotive	Consumer	0.9432
Automotive	Network	0.9696
Automotive	Office	0.9596
All excluding automotive	Automotive	0.9934
All excluding security	Security	0.9838
All excluding telecom	Telecom	0.9712
All excluding network	Network	0.9901
All excluding office	Office	0.9984

Table 4. 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) Average Feature Collection	16.8 sec
	(D) Average Compilation	5.1 sec
	(E) Average Execution Time	7.3 sec
	(F) Average Prediction (MLP)	2.1 sec
	(G) Average Recommendation (ACS)	12.5 sec

4.3 Analysis of MiCOMP's Timing Breakdown

To demonstrate the timing overhead of the MiCOMP framework, we provide Table 4 that shows a fine-grain breakdown of each phase, both the training and the inference phase. Constructing MiCOMP's model is a one-time process (for each machine learning model mentioned in Table 2) and depends on the number of applications in the training set. Table 4 reports the numbers for each specific phase by considering the CBench with 32 applications and one dataset for each as training set, and the average time for a target application under optimization. To calculate the average time, we ran the framework 32 times, each time with a new target application taken from the entire set of CBench. During the (A) *offline training-phase*, the time needed for data collection is around 5 days as each application should be compiled and executed for 19k times plus one round of feature collection. The time needed to post-process the data and to (B) *construct an MLP model*, which was the most time-consuming model among our three used models, was around 120 seconds.

During the online phase (inference phase), the average time needed for (C) *collecting the application features* was 16.8 seconds while the (D) average time for *compilation* and (E) *execution-time* on the target platform were 5.1 and 7.3 seconds, respectively. Using the trained model for (F) *prediction* took less than 2.1 seconds, and the time for our (G) *recommendation* engine (explained in Section 3.5) took around 12.5 seconds on average. Those numbers show that the initial overhead in adopting MiCOMP on the user-side (composed of the application feature collection and the prediction phase) is very reasonable at only two compilation/execution pairs.

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

4.4 Analysis of Longer Sequence Length

As described in Section 3, MiCOMP requires an upper bound on the sequence length for using the encoding scheme. To this end, we evaluate MiCOMP with different maximum values for the sequence length. A speedup prediction model requires a one time expensive training to construct an accurate model. We believe that the longer the sequence length goes, the better the chance of finding higher speedup values. We 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.

Figure 5 gives the Harmonic mean values of the actual speedups by using five selected applications. Each one has different upper bound sequence lengths. We randomly selected an application from each of the Cbench categories (automotive, compression, telecom, consumer, office, and network) since it was impractical to do this analysis with all applications. Given the upper bounds set to 3, 4, 5, 6, and 7, respectively, it generates 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 fractions 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. For our optimization sub-sequences, the upper bound value found in these experiments (sequence length ≤ 6) represents the right tradeoff between a good optimization sequence and the exploration efficiency.

4.5 MiCOMP's Prediction Accuracy

Unlike in sequence prediction models [2, 8, 46], in speedup prediction approaches the quality of prediction is measured by means of the prediction error. This metric demonstrates how close the prediction values were to the actual speedups given the same sequence. We used different error measurement techniques: *Mean Absolute Error* and *Approximation Error* to provide the error.

Mean Absolute Error (MAE). In statistics, the MAE [23] is a quantity used to measure how close predictions are to the eventual outcomes. The MAE is given by:

Table 5. Average Error Rate for the Proposed Encoding Function Versus No Encoding

M.L	MiCOMP Encoding		No 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×

Table 6. Prediction (Recommendations) Improvement of MiCOMP Using ACS Against the Ranking 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

$$MAE = \frac{1}{n} \sum_{i=1}^n |f_i - y_i|, \quad (3)$$

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 inversely proportional to the accuracy of the prediction.

Approximation Error (AE). Complementary to MAE, AE [52] is a common error measurement whereas in some data, there is some discrepancy between the exact value and the approximate one. An approximation error can occur because (1) certain measurements of the data are not precise and (2) approximated values are used instead of the real values (the iterative prediction way keeps using the predicted values). It is calculated as:

$$\delta = \frac{|\epsilon|}{|v|} = \frac{|v - v_{approx}|}{|v|}, \quad (4)$$

where $|\epsilon|$ represents the difference between the exact and the approximated (predicted) values. The definition can be extended to the case when v and $v_{approximate}$ are n-dimensional vectors.

Employing our adapted encoding scheme reduces the standard deviation and increases the accuracy of the prediction on all tested models by a factor of 1.59 up to 2.16. Table 5 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, 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

4.6 Performance Improvement of MiCOMP Recommendations

Our approach improves the exploration to find the best optimization sequences in an optimization search space, thus it recommends the best solution in fewer steps. Table 6 reports the comparison between the best speedups found by our approach and a state-of-the-art ranking approach [44, 46].

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

The results, averaged using a Harmonic mean across all applications, show that using the same number of predictions from both models, MiCOMP outperforms the ranking approach on each number of predicted optimization sequences used (1, 5, 10, 15, and 20). Here, we refer to a Top-X% speedup as the best speedup found while using the prediction model X% times.

4.7 MiCOMP's Random Iterative Compilation and Prediction

Random Iterative compilation (RIC) is known to achieve good results when compiling applications on a long run [12]. However, the approach is expensive and should be combined with more intelligent search algorithms such as machine learning techniques [2, 8, 12].

MiCOMP's RIC. In Figure 5, we show that by using a RIC on MiCOMP's sub-sequences, we outperform LLVM's -O3. Hereafter, we call this specific RIC as MiCOMP's RIC. For example, note in Figure 5 that it takes MiCOMP's RIC of sequences up to six (19k iterations in total), roughly 10^4 iterations to surpass the performance of -O3. An exact number for each application is reported on the last column of Table 7. This result reveals that in a worst-case scenario, it needs MiCOMP's RIC to explore half of the optimization space to surpass the performance of LLVM's -O3. Another insight gained could be that the probability of guessing at least one sequence that is better than LLVM's -O3 in 10 guesses is $1 - (9720/19000)^{10}$, which is more than 99.9%. Additionally, here we report columns five and six of Table 7 as: (1) the maximum speedups found by using our proposed sub-sequences while exploring the full optimization space exhaustively (19k), and (2) the corresponding optimization sub-sequences. Since the optimization space is explored exhaustively, we cannot really stress on a single number of iterations required to arrive to the maximum speedup, but rather we let the RIC finish exploration of the full space and we report the optimal solution found. The reader can refer to Table 1 to find the exact set of compiler optimizations we clustered in each sub-sequence. This experiment empirically confirms that the proposed clustering is useful on the phase-ordering space since we achieve on average a 31% speedup versus -O3. This is important for our prediction model as it defines an upper-bound for the achievable speedup we could reach by using our prediction techniques.

MiCOMP's Prediction. By adding the prediction and recommendation techniques to our RIC, we can focus on finding better results within only a few number of predictions. Here, we call iterations "predictions" when we use MiCOMP's model to predict the speedup of a full-sequence. Columns two to four in Table 7 report the results of MiCOMP's prediction when 1, 5, and 10 predictions have been made. All values have been normalized to the performance of LLVM's -O3. For each application and number of predictions, we provide two values: (1) speedup, and (2) percentage of achievable speedup, which is the speedup we could have achieved when we exhaustively searched the optimization space using our MiCOMP's RIC. We can notice that for the `network_dijkstra` application, we can gain higher speedup values, and on average, even better than -O3 from just the first prediction. Moreover, we can achieve a 5% performance improvement over -O3 when we use five predicted optimization sequences from our model. Over all our benchmarks, using our model, we can achieve 1%, 5%, and 9% speedups over -O3 using 1, 5, and 10 predicted optimization sequences, respectively. The results show that MiCOMP's prediction model is able to gain 91% of the achievable speedup by exploring less than 0.001% of its optimization space, thus providing a factor of exploration efficiency over its pure RIC. We discuss this claim in more detail in the next section.

5 COMPARISON RESULTS

In this section, we focus on the evaluation of the MiCOMP's prediction results against three different techniques: (1) MiCOMP's RIC (in Section 5.1), (2) standard optimization levels (in Section 5.2), and (3) state-of-the-art prediction models (in Section 5.3).

Table 7. Reported Speedup Numbers on the Columns 2 to 4 are A(B%): (A) Speedup (normalized by -O3), and (B) Percentage Speedup w.r.t. Optimal Speedup Value of an Exhaustive Exploration of MiCOMP's RIC. Columns 5, 6, and 7 are the Best Optimization Sub-sequences found using an Exhaustive RIC with MiCOMP's Sub-sequences, Their Corresponding Speedups, and Number of Iterations it Took to Outperform LLVM's -O3 (Total:19k)

Applications	MiCOMP's Prediction			MiCOMP's RIC		
	1 prediction	5 predictions	10 predictions	Best sub-sequence found	Speedup w.r.t -O3	# Iterations to outperform -O3 (K)
automotive_bitcount	1.04 (95.38%)	1.07 (98.12%)	1.08 (98.92%)	BEACCA	1.19	8.53
automotive_qsort1	1.01 (95.32%)	1.03 (96.93%)	1.03 (97.55%)	CBA AAC	1.04	9.41
automotive_susan_c	1.04 (96.61%)	1.06 (98.53%)	1.06 (99.07%)	BDBCCB	1.32	8.1
automotive_susan_e	1.04 (96.47%)	1.03 (98.41%)	1.04 (99.00%)	AABACA	1.15	8.78
automotive_susan_s	0.99 (96.26%)	1.01 (98.42%)	1.02 (98.98%)	ECCDCE	1.22	8.36
bzip2d	0.93 (92.77%)	0.96 (94.02%)	1.00 (94.37%)	CBDACA	1.29	8.24
bzip2e	1.09 (83.77%)	1.10 (86.02%)	1.12 (90.37%)	CBADCA	1.3	8.25
consumer_jpeg_c	1.01 (85.18%)	1.07 (90.35%)	1.10 (94.51%)	DDC	1.41	8.64
consumer_jpeg_d	1.09 (84.70%)	1.14 (88.97%)	1.17 (97.85%)	CCED	1.18	9.74
consumer_ti_2bw	0.96 (75.54%)	0.99 (80.59%)	1.02 (82.46%)	DDCAB	1.15	10.17
consumer_ti_2rgba	0.91 (80.61%)	0.95 (86.19%)	1.07 (88.08%)	DDCA	1.17	10.23
consumer_ti_dither	1.02 (80.14%)	1.09 (85.86%)	1.11 (87.68%)	CCDCD	1.3	10.12
consumer_ti_median	0.94 (79.21%)	1.02 (85.72%)	1.06 (89.31%)	DEDDC	1.32	10.48
consumer_mad	1.02 (82.14%)	1.09 (85.86%)	1.11 (87.68%)	DCEDCD	1.2	10.34
consumer_lame	0.99 (89.21%)	1.02 (90.72%)	1.06 (92.31%)	BCBACB	1.15	10.51
network_dijkstra	1.13 (60.00%)	1.29 (68.46%)	1.38 (73.00%)	EECBBE	1.51	8.32
network_patricia	0.91 (74.99%)	0.93 (80.79%)	0.97 (93.91%)	CECBAA	1.18	8.55
office_ispell	0.98 (84.99%)	1.01 (90.79%)	1.03 (93.91%)	ABCBAC	1.08	11.22
office_ghostscript	0.99 (79.99%)	1.03 (82.79%)	1.03 (90.91%)	ABEBAE	1.10	10.74
office_rsynth	1.01 (84.99%)	1.02 (90.79%)	1.03 (93.91%)	ABCBA	1.12	10.55
office_stringsearch1	0.98 (64.99%)	1.02 (70.79%)	1.01 (73.91%)	ABCBAC	1.07	10.91
security_sha	0.93 (64.99%)	1.01 (70.79%)	1.03 (73.91%)	DACECA	1.10	12.1
security_blow_sh_e	0.97 (64.99%)	1.03 (70.79%)	1.03 (73.91%)	BCCEEA	1.13	12.31
security_blow_sh_d	0.97 (64.99%)	0.99 (70.79%)	1.02 (73.91%)	ECEACD	1.10	12.33
security_rijndael_e	0.99 (64.99%)	1.02 (70.79%)	1.01 (73.91%)	CAEEC	1.11	12.12
security_rijndael_d	1.00 (64.99%)	1.01 (70.79%)	1.04 (73.91%)	ACCACE	1.06	12.17
telecom_adpcm_c	0.96 (64.99%)	1.01 (70.79%)	1.02 (73.91%)	ECDDCC	1.35	9.23
telecom_adpcm_d	0.98 (64.99%)	1.02 (70.79%)	1.01 (73.91%)	DCAACA	1.13	10.11
telecom_gsm_d	0.93 (64.99%)	1.03 (70.79%)	1.04 (73.91%)	DCAAC	1.34	9.12
telecom_CRC32	1.01 (85.18%)	1.07 (90.35%)	1.10 (94.51%)	DCAACA	1.26	8.86
telecom_pgp_d	1.04 (96.61%)	1.06 (98.53%)	1.06 (99.07%)	DCAACA	1.21	8.84
telecom_pgp_e	1.02 (80.14%)	1.09 (85.86%)	1.11 (87.68%)	DCA	1.22	9.12
Harmonic mean	1.01 (84.74%)	1.05 (87.51%)	1.09 (91.52%)	-	1.31	9.72

5.1 Comparison with Random Iterative Optimization

In this section, we compare the effectiveness of MiCOMP's prediction model to its pure RIC method shown in Section 4.7. To provide a fair comparison, first we randomized the distribution of RIC's iterations for 10,000 times to make sure the obtained model is totally uniform. We define *Normalized Performance Improvement* (NPI) as the ratio of the performance improvement achieved over the potential performance improvement:

$$NPI = \frac{E_{ref} - E}{E_{ref} - E_{best}}, \quad (5)$$

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 explored optimization space. As the execution time E of RIC 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

(a) Normalized speedup by varying the number of predictions (b) Performance of MiCOMP's prediction fixed at Top-5 while MiCOMP's RIC keeps drawing guesses

Fig. 6. Performance comparison: MiCOMP's predictions versus MiCOMP's RIC.

Table 8. MiCOMP's Performance Against OX Levels. Numbers are A (B%): (A) Average Number of Predictions, (B) Percentage of the Optimization Space Explored to Outperform OX Levels

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%)

1. An NPI value of 1 indicates that the optimal performance available was achieved. Our results are presented in Figure 6. The goal of the evaluation is to show how effective MiCOMP's prediction is at exploring the optimization space compared to RIC. Figure 6(a) depicts the comparison for both MiCOMP's prediction and RIC having the same optimization sub-sequences. The X axis refers to the number of predicted optimization sequences and the Y axis shows their corresponding speedup values. We used NPI (scaled within $[-\infty, 1]$) and the speedups are all normalized by -O3 performance. Thus, $Y = 0$ is the speedup line corresponding to -O3. We observe that MiCOMP outperforms RIC with a clear margin for each number of predictions.

Figure 6(b) compares a fixed number of MiCOMP's predictions, that is five, versus different guesses drawn from MiCOMP's RIC. Here again, we randomized the distributions 10,000 times to make sure the obtained model is totally uniform. The dots in the violin plot represent the median of the MiCOMP's top-5 prediction. The Y axis pertains to the normalized speedup with respect to the MiCOMP's RIC and the X axis corresponds to the different guesses drawn by MiCOMP's RIC. Statistically, we observe that the quality of the MiCOMP's top-5 prediction is as good as 25 to 30 guesses drawn from MiCOMP's RIC, even by considering the overhead of MiCOMP's prediction and recommendation models (we presented the fine-grain breakdown in Table 4). Thus, it achieves a factor of 5–6 $\times$ optimization *exploration efficiency* over its RIC, when we find an equal speedup in the same optimization space.

5.2 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, and dataset. As shown in Table 7, MiCOMP outperforms -O3 with a few predictions on application bases. Table 8 reports more fine-grained speedups over all standard optimization levels. This

demonstrates using A predictions (first number in the tuple), and exploring B percentage of the space (the second number) the framework finds a sequence which outperforms the OX levels. Notice that numbers are integers and in many cases are rounded up.

5.3 Comparison with State-of-the-Art Intermediate-Sequence Prediction Models

In this section, we compare MiCOMP with two state-of-the-art intermediate-sequence prediction approaches proposed in Kulkarni and Cavazos [30] and Ashouri et al. [6], respectively.

5.3.1 Intermediate Speedup Comparison Case (A). Kulkarni and Cavazos [30] 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. They used hybrid features of the dynamic compilation environment to characterize each state of the application under optimization. Specifically, the authors used 26 different features as following: The first two entries are integer values defining the size of the code and data of the method. The next six are simple Boolean properties (represented using 0 or 1) of the method that represent certain binary properties of a method, e.g., is the method synchronized or is it a leaf method. The remaining features are simply the percentage of byte codes belonging to a particular category. Contrary to the technique we propose in this article where we obtain features of the code only once before it is optimized, Kulkarni and Cavazos [30] 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 of the mentioned authors by using 100 chromosomes and 500 generations on our target architecture and we report the result in Table 9. We ran NEAT in parallel with an average running time of 1.75 hours per model (the longest took 4 hours). NEAT requires a feature collection on many optimized states of an application. Thus, collecting the needed features for the applications in the training set is a time consuming task, especially for long length sequences. The training and prediction is done with leave-one-out cross-validation to generate uniform results. We used Harmonic-mean to average the speedup gains on both models and observed $1.0295\times$ speedup compared with -O3 and 5% performance improvement against the compared work. Additionally, in 26 out of 32 applications (81% of the cases) under analysis, we observed a performance gain.

5.3.2 Intermediate Speedup Comparison Case (B). Ashouri et al. [6] demonstrated a predictive methodology to predict an intermediate speedup of an optimization from the configuration space given the current state of the application. Similar to Kulkarni and Cavazos [30], the fitness function used for the intermediate speedup was the ratio between the execution times of the application before and after a compiler optimization was added to the current optimization sequence. For instance, in order to calculate the speedup of compiler optimization a when an application was in the optimized state of $\{b, c\}$, we need to calculate the ratio of the execution times of the application in the state $\{b, c\}$ followed by the state $\{b, c, a\}$. Thus, we iterate over all the possible states and construct the optimization vectors in order to induce a model for an application under analysis.

However, unlike Kulkarni and Cavazos [30], they employed dynamic features of the application under study. As mentioned in Section 5.3.1, a major downside in an intermediate speedup approach is that an application feature should be collected on every state by means feature

Table 9. Performance Comparison of the Single Prediction by MiCOMP Against the Intermediate Speedup Approach Reported in Kulkarni and Cavazos [30]. Values Normalized to -03

Application	NEAT		MiCOMP
	Best NN Size	1 prediction	1 prediction
automotive_qsort1	1,105	1.0336	1.0385
automotive_bitcount	1,536	1.0923	1.0898
automotive_susan_c	607	1.0012	1.0491
automotive_susan_e	613	1.0211	1.0481
automotive_susan_s	1,295	1.0135	1.0195
bzip2d	1,159	1.0514	0.9898
bzip2e	1,259	1.0012	0.9798
consumer_jpeg_c	1,327	1.0205	1.1882
consumer_jpeg_e	596	1.0712	1.0981
consumer_tiff2bw	1,038	0.9522	0.9491
consumer_tiff2rgba	1,147	0.9905	0.9295
consumer_tiffdither	612	1.0222	1.0288
consumer_tiffmedian	1,356	0.9097	0.9497
consumer_lame	1,612	1.0221	1.0288
consumer_mad	1,256	0.9097	0.9497
network_dijkstra	1,343	1.0353	1.1382
network_patricia	622	0.7971	0.8585
office_ispell	1,056	0.9197	0.9897
office_ghostscript	1,356	0.9097	1.0997
office_rsynth	1,306	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	1,002	0.9891	1.0254
security_sha	1,536	1.0193	1.0178
security_blowfish_e	1,221	1.0023	1.0298
security_blowfish_d	1,534	1.0923	1.0898
security_rijndael_e	1,132	1.0113	1.0395
security_rijndael_d	1,033	1.0123	1.0598
Harmonic Mean		0.9632	1.0295

extraction and this makes the system impractical on large-scale data, especially when dynamic features are collected on every state. This was apparent in the results in Ashouri et al. [6] since their predictive methodology was bounded to an upper-bound length of 4. 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. First, here we introduced a graph-based iterative clustering approach rather than manual clustering presented in Ashouri et al. [6]. In this way, we can reach higher speedup values and we can outperform LLVM's highest standard optimization levels. Second, comparison baseline in Ashouri

Fig. 7. Performance of MiCOMP w.r.t the performance of intermediate speedup predictor approach [6].

et al. [6] 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 7 demonstrates the comparison. For this comparison, we used the same training data of an upper-bound of length 4 (to match the results reported in [6]) 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.

6 CONCLUSIONS

This work presents MiCOMP, a framework to exploit predictive modeling to mitigate the complexity of the compiler phase-ordering problem. We 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 outperforms LLVM’s -O3 optimization sequence. Moreover, MiCOMP has a simple encoding function that encodes an optimization sequence into a bit string, enabling 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 integrated similarity measures to boost exploration efficiency. We show that MiCOMP outperforms LLVM’s standard optimization levels within the first few predictions and achieves 90% of the available speedup by traversing less than 0.001% of the optimization sequence space. This is rather crucial when the optimization space consists of a large number of different optimization sequences to choose from for an application. Future works will focus on adapting accurate deep-learning models on the problem.

ACKNOWLEDGMENTS

The authors would like to thank Saeed Zakeri at CUNY for his insights on the encoding scheme.

REFERENCES

- [1] Bas Aarts, Michel Barretea, François Bodin, Peter Brinkhaus, Zbigniew Chamski, Henri-Pierre Charles, Christine Eisenbeis, John Gurd, Jan Hoogerbrugge, Ping Hu, and others. 1997. OCEANS: Optimizing compilers for embedded applications. *Euro-Par’97 Parallel Processing (1997)*, 1351–1356.
- [2] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael F. P. O’Boyle, John Thomson, Marc Toussaint, and Christopher K. I. 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.

- [3] Lelac Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. 2004. Finding effective compilation sequences. *ACM SIGPLAN Notices* 39, 7 (2004), 231–239.
- [4] 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>.
- [5] Amir Hossein Ashouri. 2016. *Compiler Autotuning Using Machine Learning Techniques*. Ph.D. Dissertation. Politecnico di Milano, Italy. <http://hdl.handle.net/10589/129561>.
- [6] 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>
- [7] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2017. A survey on compiler autotuning using machine learning. *ACM Trans. Computing Surveys (CSUR)* (2017).
- [8] 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>
- [9] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, and Cristina Silvano. 2014. A bayesian network approach for compiler auto-tuning for embedded processors. In *Proceedings of the 2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*. IEEE, 90–97. DOI: <http://dx.doi.org/10.1109/ESTIMedia.2014.6962349>
- [10] 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 *Proceedings of the 2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 124–129. DOI: <http://dx.doi.org/10.1109/VLSI-SoC.2013.6673262>
- [11] Michele Banko and Eric Brill. 2001. Mitigating the paucity-of-data problem: Exploring the effect of training corpus size on classifier performance for natural language processing. In *Proceedings of the First International Conference on Human Language Technology Research*. Association for Computational Linguistics, 1–5.
- [12] 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*.
- [13] 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, D.C., 185–197. DOI: <http://dx.doi.org/10.1109/CGO.2007.32>
- [14] John Cavazos and J. Eliot B. Moss. 2004. Inducing heuristics to decide whether to schedule. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 183–194.
- [15] 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.
- [16] John G. Cleary, Leonard E. Trigg, and others. 1995. K*: An instance-based learner using an entropic distance measure. In *Proceedings of the 12th International Conference on Machine Learning*, Vol. 5. 108–114.
- [17] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. 2005. ACME: Adaptive compilation made efficient. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 69–77.
- [18] 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.
- [19] Grigori Fursin and Olivier Temam. 2010. Collective optimization: A practical collaborative approach. *ACM Transactions on Architecture and Code Optimization (TACO)* 7, 4 (2010), 20.
- [20] 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.
- [21] 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.
- [22] Kenneth Hoste and Lieven Eeckhout. 2007. Microarchitecture-independent workload characterization. *IEEE Micro* 27, 3 (2007), 63–72.
- [23] Rob J. Hyndman and Anne B. Koehler. 2006. Another look at measures of forecast accuracy. *International Journal of Forecasting* 22, 4 (2006), 679–688.

- [24] Michael R. Jantz and Prasad A. Kulkarni. 2013. Exploiting phase inter-dependencies for faster iterative compiler optimization phase order searches. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 1–10.
- [25] 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.
- [26] Richard Arnold Johnson, Dean W. Wichern, and others. 1992. *Applied Multivariate Statistical Analysis*. Vol. 4. Prentice Hall, Englewood Cliffs, NJ.
- [27] 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)*. 237–248. DOI : <http://dx.doi.org/10.1109/PACT.2000.888348>
- [28] Sotiris B. Kotsiantis, I. Zaharakis, and P. Pintelas. 2007. Supervised machine learning: A review of classification techniques. (2007).
- [29] P. A. Kulkarni, D. B. Whalley, G. 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).
- [30] Sameer Kulkarni and John Cavazos. 2012. Mitigating the compiler optimization phase-ordering problem using machine learning. *ACM SIGPLAN Notices* 47, 10 (2012), 147–162.
- [31] LLVM 2017. LLVM Compiler Passes. Retrieved from <http://www.llvm.org/docs/Passes.html>.
- [32] David B. Loveman. 1977. Program improvement by source-to-source transformation. *Journal of the ACM (JACM)* 24, 1 (1977), 121–145.
- [33] 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, 190–200. DOI : <http://dx.doi.org/10.1145/1065010.1065034>
- [34] Luiz G. A. Martins, Ricardo Nobre, Joao M. P. Cardoso, Alexandre C. B. 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.
- [35] 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.
- [36] 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.
- [37] John Neter, Michael H. Kutner, Christopher J. Nachtsheim, and William Wasserman. 1996. *Applied Linear Statistical Models*. Vol. 4. Irwin, Chicago.
- [38] Ricardo Nobre, Luiz G. A. Martins, and João M. P. 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.
- [39] Ricardo Nobre, Luiz G. A. Martins, and Joao M. P. 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.
- [40] Ricardo Nobre, Luiz G. A. Martins, and João M. P. Cardoso. 2016. 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.
- [41] Ricardo Nobre, Luis Reis, and Joao M. P. Cardoso. 2016. Compiler phase ordering as an orthogonal approach for reducing energy consumption. In *Proceedings of the 19th Workshop on Compilers for Parallel Computing (CPC’16)*.
- [42] Holger Orup. 1999. On-the-fly one-hot encoding of leading zero count. (Oct. 26, 1999). US Patent 5,974,432.
- [43] Sankar K. Pal and Sushmita Mitra. 1992. Multilayer perceptron, fuzzy sets, and classification. *IEEE Transactions on Neural Networks* 3, 5 (1992), 683–697.
- [44] EunJung Park, John Cavazos, and Marco A. Alvarez. 2012. Using graph-based program characterization for predictive modeling. In *Proceedings of the 10th International Symposium on Code Generation and Optimization*. ACM, 196–206.
- [45] 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.
- [46] 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.
- [47] 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.

- [48] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2011. *Introduction to Recommender Systems Handbook*. Springer.
- [49] 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.
- [50] Satu Elisa Schaeffer. 2007. Graph clustering. *Computer Science Review* 1, 1 (2007), 27–64.
- [51] Burr Settles. 2010. Active learning literature survey. *Computer Sciences Technical Report* 1648 (2010).
- [52] Steve Smale and Ding-Xuan Zhou. 2003. Estimating the approximation error in learning theory. *Analysis and Applications* 1, 01 (2003), 17–41.
- [53] 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, 77–90.
- [54] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: A new approach to optimization. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 264–276.
- [55] Spyridon Triantafyllis, et al. 2003. Compiler optimization-space exploration. In *Code Generation and Optimization (CGO'03)*. IEEE, 204–215.
- [56] Steven R. Vegdahl. 1982. Phase coupling and constant generation in an optimizing microcode compiler. *ACM SIGMICRO Newsletter* 13, 4 (1982), 125–133.
- [57] Wei Zhang, Deli Zhao, and Xiaogang Wang. 2013. Agglomerative clustering via maximum incremental path integral. *Pattern Recognition* 46, 11 (2013), 3056–3065.

Received September 2016; revised May 2017; accepted June 2017