A Survey on Compiler Autotuning using Machine Learning

AMIR H. ASHOURI, University of Toronto, Canada
WILLIAM KILLIAN, Millersville University of Pennsylvania, USA
JOHN CAVAZOS, University of Delaware, USA
GIANLUCA PALERMO, Politecnico di Milano, Italy
CRISTINA SILVANO, Politecnico di Milano, Italy

Since the mid-1990s, researchers have been trying to use machine-learning based approaches to solve a number of different compiler optimization problems. These techniques primarily enhance the quality of the obtained results and, more importantly, make it feasible to tackle two main compiler optimization problems: optimization selection (choosing which optimizations to apply) and phase-ordering (choosing the order of applying optimizations). The compiler optimization space continues to grow due to the advancement of applications, increasing number of compiler optimizations, and new target architectures. Generic optimization passes in compilers cannot fully leverage newly introduced optimizations and, therefore, cannot keep up with the pace of increasing options. This survey summarizes and classifies the recent advances in using machine learning for the compiler optimization field, particularly on the two major problems of (1) selecting the best optimizations, and (2) the phase-ordering of optimizations. The survey highlights the approaches taken so far, the obtained results, the fine-grain classification among different approaches and finally, the influential papers of the field.

CCS Concepts: • Computing methodologies → Machine learning; • Software and its engineering → Compilers;

Additional Key Words and Phrases: Autotuning, Phase ordering, Optimizations, Compilers, Machine Learning


1 INTRODUCTION

Moore’s Law [216] states that transistor density doubles every two years. However, the rate at which compilers improve is on the order of a few percentage points each year. Compilers are necessary tools bridging the gap between written software and target hardware. There are many unsolved research challenges in the domain of compilers [110]. Entering the post Moore’s Law era [79], compilers struggle to keep up with the increasing development pace of ever-expanding hardware landscape (e.g., CPUs, GPUs, FPGAs) and software programming paradigms and models (e.g., OpenMP, MPI, OpenCL, and OpenACC). Additionally, the growing complexity of modern compilers and increasing concerns over security are among the most serious issues that the compilation research community faces.

Compilers have been used for the past 50 years [4, 110] for generating machine-dependent executable binary from high-level programming languages. Compiler developers typically design optimization passes to transform each code segment of a program to produce an optimized version of

This work is partially supported by the EU Commission H2020-FET-HPC program under the grant ANTAREX-671623. Author’s addresses: AH. Ashouri, ECE Department, University of Toronto, Canada; w. Killian, Millersville University of Pennsylvania, US; J. Cavazos, Department of Computer and Information Science, University of Delaware, USA; G. Palermo and C. Silvano, DEIB, Politecnico di Milano, Italy;

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.
© 2018 ACM. 0360-0300/2018/1-ART1 $15.00
DOI: 10.1145/nnnnnn.nnnnnnn

an application. The optimizations can be applied at different stages of the compilation process since compilers have three main layers: (1) front-end (2) intermediate-representation (IR) and (3) backend. At the same time, optimizing source code by hand is a tedious task. Compiler optimizations provide automatic methods for transforming code. To this end, optimizing the intermediate phase plays an important role in the performance metrics. Enabling compiler optimization parameters (e.g., loop unrolling, register allocation, etc.) could substantially benefit several performance metrics. Depending on the objectives, these metrics could be execution time, code size, or power consumption. A holistic exploration approach to trade-off these metrics also represents a challenging problem [190].

Autotuning [33] addresses automatic code-generation and optimization by using different scenarios and architectures. It constructs techniques for automatic optimization of different parameters to maximize or minimize the satisfiability of an objective function. Historically, several optimizations were done in the backend where scheduling, resource-allocation and code-generation are done [53, 91]. The constraints and resources form a linear system (ILP) which needs to be solved. Recently, researchers have shown increased effort in introducing front-end and IR-optimizations. Two observations support this claim: (1) the complexity of a backend compiler requires exclusive knowledge strictly by the compiler designers, and (2) lower overheads with external compiler modification compared with back-end modifications. The IR-optimization process normally involves fine-tuning compiler optimization parameters by a multi-objective optimization formulation which can be harder to explore. Nonetheless, each approach has its benefits and drawbacks and are subject to analysis under their scope.

A major challenge in choosing the right set of compiler optimizations is the fact that these code optimizations are programming language, application, and architecture dependent. Additionally, the word optimization is a misnomer — there is no guarantee the transformed code will perform better than the original version. In fact, aggressive optimizations can even degrade the performance of the code to which they are applied [243]. Understanding the behavior of the optimizations, the perceived effects on the source-code, and the interaction of the optimizations with each other are complex modeling problems. This understanding is particularly difficult because compiler developers must consider hundreds of different optimizations that can be applied during the various compilation phases. The phase-ordering problem is realized when considering the order which these hundreds of optimizations are used.

There are several open problems associated with the compiler optimization field that have not been adequately tackled [265]. These problems include finding what optimizations to use, which set of parameters to choose from (e.g., loop tiling size, etc.), and in which order to apply them to yield the best performance improvement. The first two questions create the problem of selecting the best compiler optimizations and the taking into account the third question is forming the phase-ordering problem of compiler optimizations.

The problem of phase-ordering has been an open problem in the field of compiler research for many decades [18, 144, 146, 246, 253]. The inability of researchers to fully address the phase-ordering problem has led to advances in the simpler problem of selecting the right set of optimizations, but even the latter has yet to be solved [20, 40, 57].

The process of selecting the right optimizations for snippets of code, or kernels, is typically done manually, and the sequence of optimizations is constructed with little insight into the interaction between the preceding compiler optimizations in the sequence. The task of constructing heuristics to select the right sequence of compiler optimizations is infeasible given the increasing number of compiler optimizations being integrated into compiler frameworks. For example, GCC has more than 200 compiler passes, referred to as compiler options 1, and LLVM-clang and LLVM-opt each have more than 150 transformation passes 2. Additionally, these optimizations are targeted at different phases of the compilation process. Some of these passes only analyze data access while others may look at loop nests. Most compiler optimization flags are disabled by default, and compiler developers rely on software developers to know which optimizations should be

1https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html
2http://llvm.org/docs/Passes.html
beneficial for their application. Compiler developers provide standard “named” optimization levels, e.g., -O1, -O2, -Os, etc. to introduce a fixed-sequence of compiler optimizations that on average achieve better performance on a set of benchmark applications chosen by the compiler developers. However, using predefined optimizations usually is not good enough to bring the best achievable application-specific performance. One of the key approaches recently used in the literature to find the best optimizations to apply given an application is inducing prediction models using different classes of machine learning [8]. The central focus of this survey is to highlight approaches which leverage machine learning to find the best optimizations to apply.

**Contribution.** In this survey, we provide an extensive overview of techniques and approaches done by the authors tackling the aforementioned problems. We highlight over 150 recent papers proposed for compiler autotuning when Machine Learning (ML) was used. The classification in different subfields achieved by the use of several representative features shown in Figure 1. To the best of our knowledge, the first application of machine learning for the compiler autotuning problem was done by [66, 138, 177]. However, there were other original works which tackled the problem of compiler autotuning without the use of machine learning techniques [40, 61, 162, 188, 203, 246, 254, 260]. These other works left unsolved problems in the field and acted as the driving force of using machine learning. We consider the past 25 years of research related to autotuning compilers as it covers the entire time span of the literature 3. Additionally, this article can be leveraged as a connecting point for two existing surveys [32, 219] on the compiler optimization field.

**Organization.** We organize this survey as follows: We start by providing an overview of the different methods of data acquisition; followed by presenting preprocessing techniques. We then discuss different machine learning models and the types of objectives a model is constructed. Finally, we elaborate on the different target platforms these techniques have been applied. Additionally, to facilitate this organizational flow depicted in Figure 1, we provide Figure 2 for which we show a sample autotuning framework leveraging compilers and machine learning. We discuss more on the autotuning nomenclatures in Section 2.4.

Section 2 discusses the motivations and challenges involved in the compiler optimization research; this is followed by an analysis of the optimization space for the two major optimization problems in Section 2.3. We review the existing characterization techniques and the classification of those in Section 3. Furthermore, we discuss the machine learning models used in Section 4 and provide a full classification of different prediction techniques used in the recent research in Section 5. Section 6 examines the various space exploration techniques on how optimization configurations are traversed within the optimization space. In Section 7, we discuss the different target architectures and compiler frameworks involved in the tuning process. We include a brief review of the Polyhedral compilation framework along with other widely used compiler frameworks in Section 7. Section 8 includes a discussion on the influential papers of the field classified by their obtained (1) performance, (2) novelty, influence they had on the succeeding work, and (3) number of citations. Finally, we conclude the article with a brief discussion on past and future trends of compiler autotuning methodologies using machine learning.

**Scope of the survey.** We organize the research performed in this survey in different categories to highlight their similarities and differences. However, it is important to note that the presented study has many potential classifications in the domain of machine learning and compilers research. We organize the survey in a way that all research papers corresponding to a specific type of classification are cited under each classification. Furthermore, under every classification, we selectively picked the more notable works and we provide more elaboration on their contribution. As an example, Cavazos et al. [48] proposed to use performance counters to characterize an application. The vector of features then can be used to construct prediction models. Since this work was using this novel way of characterization, it is elaborated more on the Section 3.2.1. Due to our classification policy, this work is also cited in other classification tables (prediction type, target platform). We hope

---

3We look forward to keep updating the current survey on a quarterly basis at its arXiv branch [19] and we encourage researchers to inform us about their new related work.
this survey will be useful for a wide range of readers including computer architects, compiler developers, researchers, and technical professionals.

2 COMPILER OPTIMIZATIONS

2.1 A Note on Terminology and Metrics

Since research works mentioned in this survey originated from varying authors, terminologies may be locally and contextually defined. This results in term definitions which may not strictly be defined to apply to all cited publications. We clarify terms used in this survey here and relate them to the publications discussed. The field of compiler optimization has been referred to as compiler autotuning, compilation, code optimization, collective optimization, and program optimization. To maintain clarity, we do not use all these terms but instead use optimizing compilers, or compiler autotuning. Moreover, under each classified subsection, we will point out the other nomenclatures that have been used widely and our reference subsequently.
Table 1. A Classification Based on the type of the Problem

<table>
<thead>
<tr>
<th>Classes</th>
<th>References</th>
</tr>
</thead>
</table>

2.2 Compiler Optimization Challenges and Benefits

The majority of potential speedup no longer arrives at the increase of processor core clock frequencies. Automatic methods of exploiting parallelism and reducing dependencies are needed. As such, compiler optimizations [188] allow a user to affect the generated code without changing the original high-level source code. When these optimizations are applied may result in a program running better on a target architecture. Since a user is not able to manually tune a large code base, automatic methods must be introduced. Furthermore, manual tuning is not portable – transformations applied to code running on one architecture is not guaranteed to yield the same performance increase on another architecture.

2.3 Compiler Optimization Problems

The problem of interdependency among phases of compiler optimizations is not unique to the compiler optimization field. Phase inter-dependencies have been noted in traditional optimizing compilers between flow analysis and constant folding as well as between code generation and register allocation [154, 246].

In optimization theory, “a feasible set, search space, or solution space is the set of all possible points (sets of values of the choice variables) of an optimization problem that satisfy the problem’s constraints, potentially including inequalities, equalities, and integer constraints” [231]. The optimization process normally starts by finding the initial set, and the candidates are usually pruned and down-sampled (depending on the algorithm/scenario). Compiler optimization problems can be split into one of two subareas based on whether we (I) enable/disable a set of optimizations (optimization selection problem), or, (ii) change the ordering of those optimizations (phase-ordering problem). Here, we briefly discuss the different optimization space of the two. Table 1 classifies the existing literature based on the type of the compiler optimization problem. More recent work has addressed the selection problem since the phase-ordering problem is a more difficult research problem to solve.

2.3.1 The Problem of Selecting the Best Compiler Optimizations.

Several compiler optimization passes form an optimization sequence. When we ignore the ordering of these optimizations and focus on whether to apply the optimizations, we define the problem scope to be the selection of the best compiler optimizations. Previous researchers have shown that the interdependencies and the interaction between enabling or disabling optimizations in a sequence can dramatically alter the performance of a running code even by ignoring the order of phases [3, 20, 40].

Optimizations Space. Let \( o \) be a Boolean vector having elements \( o_i \) as different compiler optimizations. An optimization element \( o_i \) can be defined as either \( o_i = 1 \) (enabled), or \( o_i = 0 \) (disabled). An optimization sequence can be specified by a vector \( o \). This vector is Boolean and has \( n \) dimension:

\[
|\Omega_{Selection}| = \{0, 1\}^n \tag{1}
\]

For application \( a_i \) being optimized, \( n \) shows the number of optimizations under analysis [14]. The selection problem’s optimization space has an exponential space as its upper-bound. For example, with \( n = 10 \), there is a total state space of \( 2^n \) (1024) optimization options to select among. The different optimizations scale by the total number of target applications \( a_s \), yielding a search space of \( A = a_0...a_N \) where \( A \) is the set of our applications under analysis. One can define an extended
version of the optimization space by switching a binary choice (on or off) to a many-choice variant per each compiler optimization. Equation 1 shows the case where we have more than just a binary choice. Certain compiler optimizations such as loop unrolling and tiling offer multiple constant factors of tuning, i.e., 4, 8, 16, m. Also, some optimizations can take more than one parameter. To simplify the presentation of the equation, we consider m as their total number of optimization choices a compiler optimization have. We restrict this proposed presentation to discretized values. Continuous ranges could be approximated by predefining a discrete number of options within the continuous range of values; however, we did not find continuous options with any compilers used in this research domain. Consequently, we have the previous equation as:

$$|\Omega_{Selection\_Extended}| = \{0, 1, \ldots, m\}^n \quad (2)$$

### 2.3.2 The Phase-ordering Problem

A problem common to multi-phase optimizing compilers is that there is no ideal ordering of phases. An optimization pass, A, transforms the program in ways that hinder the effect of some optimizations that otherwise could have been performed by the following pass, B. If the order of the two phases is switched, phase B performs optimizations that may deprive phase A of opportunities. The dual of this situation is one in which the two phases open up new opportunities for each other [154]. Compiler designers must take into account the order in which each optimization phase is placed and performed. “A pair of optimization phases (comprises of many passes) may be interdependent in the sense that each phase could benefit from transformation produced by the other” [154, 246].

**Optimizations Space.** A phase-ordering optimization sequence is in the factorial space due to having permutations:

$$|\Omega_{Phases}| = n! \quad (3)$$

where n shows the number of optimizations under analysis [14, 17]. However, the mentioned bound is a simplified phase-ordering problem having a fixed length optimization sequence length and no repetitive application of optimizations. Allowing optimizations to be repeatedly applied and a variable length sequence of optimizations will expand the problem space to:

$$|\Omega_{Phases\_Repetition\_variableLength}| = \sum_{i=0}^{l} n^i \quad (4)$$

where n is the number of optimizations under study and l is the maximum desired length for the optimization sequence. Even having reasonable values for n and m, the formed optimization search space is enormous. For example, assuming n and m are both equal to 10, leads to an optimization search space of more than 11 billion different optimization sequences to identify given each piece of code being optimized. The problem of finding the right phases does not have a deterministic upper-bound given an unbounded optimization length [17, 154].

### 2.4 Autotuning Techniques Overview

Autotuning refers to a methodology where there is some model (can be a search heuristic, an algorithm, etc.) that infers one or more objectives with minimal or no interaction from a user [257]. Figure 2 shows a sample autotuning framework leveraging compilers and machine learning. This section will go over each of the components for this example framework. In the domain of compilers, autotuning usually refers to defining an optimization strategy by means of a design of experiment (DoE) [211] where a tuning parameter space is defined. This parameter space creates many versions of a given input program. When many versions of a given input program are explored, this is known as iterative exploration. Iterative exploration can either be done exhaustively by examining all versions of a given input or sampling a subset of the space by using a search exploration heuristic. When a search exploration heuristic is used, the entire search space may not be examined. This causes a trade-off between the number of instances used to construct a model (accuracy) and the computational complexity required to construct a model (time).

An autotuning framework must also have a desired target or outcome which is usually defined in terms of performance. Extracted information from an application and measured objectives are
Fig. 2. A Sample Autotuning Framework Using Machine Learning (1) Top: The data flow through the various components of the training phase, where a model is induced based on the characteristics of applications under training set, and, (2) Bottom: The data flow through the various components of the test phase, where the already trained model is used to predict an outcome for a given test application with applications under the test set.

fed into a machine learning algorithm from which a predictive model is emitted. The extracted application information could be extremely large or may be of varying sizes. Some form of dimension reduction is usually applied to the extracted features to reduce the amount of information. Dimension reduction is also necessary to form consistent features across varying applications. Principle component analysis is one of the more popular dimension reduction methods, as indicated by the surveyed research in this domain [16, 21, 22].

Finally, once the model is constructed, the test set can be fed through feature extraction and dimension reduction to produce a feature vector which can be fed into the predictive model. A predicted outcome is yielded from this model and can be compared to the known outcome for evaluations and prediction error measurement.

3 APPLICATION CHARACTERIZATION TECHNIQUES

For computer architects and compiler designers, understanding the characteristics of applications running on current and future computer systems is of utmost importance during design. Applications of machine learning to compiler optimizations require a representation of the code being optimized. Thus, a tool is required to scan the code being optimized to collect its representative features. To obtain a more accurate model, compiler researchers have been trying to better understand the behavior of running applications and build a vector of features that best represents pair functionality. In general, (1) the derived feature vector must be representative enough of its application, and, (2) different applications must not have the same feature vector. Thus, constructing a large, inefficient feature vector slows down – or even halts – the ML process and can reduce the precision. In this survey, we present different program characterization techniques used in the referenced literature on compiler autotuning: (i) Static analysis of the features, (ii) Dynamic feature collection, and, (iii) Hybrid feature collection, which uses a combination of the previous two or other extraction methods. Table 2 refers to such classification.

3.1 Static Analysis

Static Analysis, or static features collection, tries to collect features that are non-functional to a code being run on a given architecture. Static analysis involves parsing source code at the front-end, intermediate representation (IR), the backend, or any combination of the three. Collecting static features doesn’t require the code to be executed and is considered to be one of the strongest support cases for its use. We briefly classify source code features leveraged in prior research.

3.1.1 Source Code Features. Source code (SRC) features are abstractions of some selected properties of an input application or the current compiler intermediate state when other optimizations
Table 2. A Classification Based on Application Characterization Techniques

<table>
<thead>
<tr>
<th>Classification</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Static</strong></td>
<td></td>
</tr>
<tr>
<td>Graph-based (IR) Features</td>
<td>[49, 50, 64, 69, 76, 90, 94, 138, 151, 155, 158, 164, 167, 180, 183, 193, 197, 199, 227, 229, 232, 242, 253]</td>
</tr>
<tr>
<td><strong>Dynamic</strong></td>
<td></td>
</tr>
<tr>
<td>Architecture Independent</td>
<td>[14, 17, 18, 20, 27, 58, 76, 196]</td>
</tr>
<tr>
<td>Hybrid features</td>
<td>[20, 27, 75, 151, 155, 167, 193, 197, 199, 213, 242]</td>
</tr>
</tbody>
</table>

have already been applied. These features range from simple information such as the name of the current function to the values of compiler parameters to the pass ordering in the current run of the compiler. There are numerous source-code feature extractors used in the literature. Fursin et. al. proposed Milepost GCC [99, 102], as a plugin to GCC compiler to extract source-level features.

3.1.2 Graph-based Features. Graph-based representation makes data and control dependencies explicit for each operation in a program. “Data dependence graphs have provided an explicit representation of the definition-use relationships implicitly present in a source-code” [84, 187]. A control flow graph (CFG) [5] has often been the representation of the control flow relationships of an application. The program dependence graph explicitly represents both the essential control relationships and the essential data relationships (as present in the data dependence graph). This information is available without the unnecessary sequencing present in the control flow graph [84].

There are numerous tools to extract a control flow graph of a kernel, a function, or an application. MinIR [174], LLVM’s Opt, IDA pro [78] are such examples of these tools available.

Koseki et. al. [138] used CFG and dependency graphs to understand good unrolling factors to apply to loops. According to the formula for determining the efficiency of loop execution $P$ increases monotonically until it saturates. Therefore, their algorithm never chooses directions that lead to local maximum points. Moreover, they showed the method can find the point for which unrolling is performed the fewest times, as it chooses the nearest saturated point to determine the number of times and the directions in which loop unrolling is performed.

Park et. al. [195] introduced a novel compiler autotuning framework which used graph-based features of an application from its intermediate representation (IR). The authors used MinIR on the SSA form of the application to collect control flow graph of the basic blocks including the directed edges to which they have predecessor and successors. In order to construct the feature vectors needed in the predictive modeling, the authors used shortest path graph kernels [44]. The method compared the similarity of two application’s shortest path at each basic block. Once the feature vectors were built, a machine learning model was created to predict the speedup of a given sequence. Finally, the authors experimentally evaluated their proposed approach using polyhedral optimization and PolyBench suite against using source-code features and dynamic features with multiple compiler backends.

Nobrea et. al. [183] proposed an iterative compilation approach using graph-based features of the code being optimized to mitigate selecting and ordering of compiler optimization passes in LLVM. The authors proposed heuristics to reduce the search space and converge faster.

3.2 Dynamic Characterization

Dynamic characterization involves extracting the performance counters (PC) that are used to provide information as to how well an application executes. The data can help determine system bottlenecks and provides hints on the fine-tuning of an application’s performance. Applications

4http://ctuning.org/wiki/index.php/CTools:MilepostGCC:StaticFeatures:MILEPOST_V2.1
can also use performance counter data to determine how many system resources to utilize. For example, an application that is data cache intensive can be tuned by exploiting more cache locality. Here, we briefly describe and classify the different types of collecting performance counters. Refer to Table 2 for full classification on different application characterization techniques.

3.2.1 Architecture Dependent Characterization. Recent processors provide a special group of registers that make it possible to collect several measurable events corresponding to their performance. These events can be measured with minimal disruption to a running application and can describe several characteristics such as cache behaviors (hits and misses) and memory footprints [165]. For example, in the work of Cavazos et. al [48], there were 4 registers on the AMD Athlon that could collect performance counter events; however, 60 different events could be collected. "By multiplexing the use of special registers, it is possible to collect anywhere between 4 and 60 types of events per run" [48]. A downside of using performance counters to characterize applications is its limited reuse for other platforms. The metrics collected are solely intended for the platform the data was collected on, reducing the platform portability.

There are tools publicly available to collect such metrics as suggested by [81, 167, 198], such as PAPI [179]. PAPI is able to specify a standard application programming interface (API) on different architectures for collecting "portable" hardware-specific performance counters.

Monitoring these events helps the tuning process by constructing a correlation between the type of object code and the performance of that code on the target architecture. Triantafyllis et al. [243] used PFMon [123], another notable performance monitoring tool when executing programs for the Intel Itanium platform.

Cavazos et. al. [48] proposed an offline machine learning based model which can be used to predict a good set of compiler optimization sequences. Their method uses performance counters to construct an application’s feature vector and use them to predict good compiler optimization sequences. The authors showed that using this method, they could outperform existing static techniques because they were able to capture memory and cache behaviors of applications.

3.2.2 Architecture Independent Characterization. The information collected from a dynamic characterization, referred to as feature vector, is a compact summary of an application’s dynamic behavior at run-time. This information summarizes essential aspects of an application’s running performance, i.e. number of cache misses or the utilization of floating point unit [48]. However, the information can be inaccurate or misleading if the application is run on different target architectures. Variances may exist between different targets such as cache size, execution ports, and scheduling algorithms. Architecture dependent counters, while accurate, are unable to be used in a cross-platform manner. Because of this limitation, researchers proposed a different way of collecting dynamic behaviors which can be ported to other platform if they have the same instruction set architecture (ISA), i.e. X86_64. This way of collecting features are known as instrumentation and are done using the dynamic program analysis tools.

Intel Pin [163] is a noteworthy framework that enables an instrumented collection. Pin is a dynamic binary instrumentation tool that analyzes a binary as it executes on a given architecture. A major advantage of Pin is that it requires no recompilation of source code and supports instrumenting applications that can generate code dynamically. Pintools are the tools developed using Pin framework which can perform several analyses over applications defined in user space. Microarchitecture-independent workload Characterization of Applications (MICA) [116] is an example of a Pintool which is capable of collecting a number of program features at run-time. These collected features are independent from the microarchitecture and consider components such as the branch predictor, memory footprint, and other components that exist across various architectures instead of the specific program counters previously mentioned.

3.3 Hybrid Characterization
The hybrid characterization technique consists of the combination of the previously known technique in a way that adds more information on the application under analysis. In some cases [20, 197],
hybrid feature selection can more accurately capture application behaviors since different levels of feature extraction are considered.

Park et. al. [197] used HERCULES, a pattern-driven tool to characterize an application [128, 129]. This characterization technique is suitable for identifying interesting patterns within an application. They used HERCULES to construct prediction models that effectively select good sets of compiler optimization sequences given an application.

### 3.4 Dimension Reduction Techniques

In the studied literature, a dimension-reduction process is important for three main reasons: (i) elimination of noise that may perturb further analysis, (ii) significant reduction in the size and training time of a model, and (iii) improved understanding of the feature space. The techniques used are mainly Principal Component Analysis (PCA) [125], and Exploratory Factor Analysis (EFA) [108].

1. Elimination of Noise: Collected static and dynamic data may not only contain relevant features for learning. Feeding all extracted features may make it much more difficult for some machine learning algorithms to fit a model to a training set. For example, support vectors must construct a linearly separable space. The construction may become difficult or impossible with a large number of features. 2. Reduction of Size and Training Time: Many machine learning algorithms are constructed through the analysis of feature vectors. Support vectors’ and neural networks’ computational complexity are directly correlated with the size of the feature vectors. It is advantageous to reduce the size of the feature vectors to improve the training time and reduce the size of the generated model. 3. Improved Understanding of a Feature Space: If a large number of features are used to characterize a problem, it can be difficult to understand the subset of features that are most important to the learning model. Through dimension reduction, a large set of features can be reduced to a much smaller set of features, which can significantly improve the speed of learning and can help analysts understand which features, or a combination of features, are important. In terms of visualization and human analysis, the reduced dimension count makes it much easier to realize and identify trends and patterns related to different feature vectors.

Reducing the feature space used as input for a machine learning algorithm may yield better learning results; however, the type of feature reduction used can be an important factor itself [3, 20]. For instance, PCA was used in the original work proposed by Ashouri et al. [14], but in an extension the authors revised the model by exploiting EFA and observed clear benefits [20]. We elaborate more on these methods in latter sections of the survey. Table 3 classifies the use of each technique in the studied literature.

<table>
<thead>
<tr>
<th>Classification</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Principal Component Analysis (PCA)</td>
<td>[3, 14, 17, 18, 20, 27, 28, 58, 76, 137, 195, 196, 199, 240]</td>
</tr>
<tr>
<td>Factor Analysis (FA)</td>
<td>[20]</td>
</tr>
</tbody>
</table>
PCA, where the components are given by a combination of the observed features, EFA has factors represent a hidden process behind the feature generation. In both cases, there is no way to indicate by name the output columns, since they are not directly observable.

3.4.1 Principal Component Analysis. With PCA, the goal is to identify a summary of \( \gamma \). A second vector, \( \rho \), of the same length of \( \gamma \) is organized by some extracted variable change. \( \rho \) is calculated with a linear combination of the elements in \( \gamma \). The combination of \( \gamma \)'s elements yielding \( \rho \) is decided upon the analysis of the matrix \( P \). All elements in \( \rho \) are orthogonal (uncorrelated) and are sorted by their variance. Because of these two properties, the first elements of \( \rho \) (also named principal components) carry most of the information of \( \gamma \). The reduction can be obtained by generating a vector \( \alpha \) which keeps only the \( k \) most significant principal components in \( \rho \). Note that principal components in \( \rho \) are not meant to have a meaning; they are only used to summarize the vector \( \gamma \) as a signature.

3.4.2 Factor Analysis. Factor analysis explains the correlation between the different variables in \( \gamma \). Correlated variables in \( \gamma \) are likely to depend on the same hidden variable in \( \alpha \). The relationship between \( \alpha \) and the observed variables is obtained by exploiting the maximum likely method based on the data in matrix \( P \). When adopting PCA, each variable in \( \alpha \) tends to be composed of all variables in \( \gamma \). Because of this characteristic, it is difficult to tell what a specific component in PCA space represents. When adopting EFA, the components \( \alpha \) tend to depend on a smaller set of elements in \( \gamma \) which correlate with each other. \( \alpha \) is a compressed representation of \( \gamma \), where elements in \( \gamma \) that are correlated (i.e. that carry the same information) are compressed into a reduced number of elements in \( \alpha \).

4 MACHINE LEARNING MODELS

Machine learning investigates the study and construction of techniques and algorithms that are able to learn certain attributes from data and make predictions on them [175]. Many types and subfields of machine learning exist, so we chose to classify them based on three broad categories: (i) Supervised learning, (ii) Unsupervised learning, and, (iii) Other Methods (including reinforcement learning, graph-based technique, and statistical methods). The classification is depicted in Table 4. In each subsection, we provide an overview of the method and provide a summary of the notable related work.

When adapting an application of machine learning for compilers, the general optimization problem is to construct a function that "takes as input a tuple \((F, T)\), where \( F \) is the feature vector of the collected characteristics of an application being optimized" [196]; and \( T \) is one of the several possible compiler optimization sequences applied to this application. Its output may be a predicted speedup value of \( T \), or depending on the scenario, the predicted optimization sequence of \( T \), when applied to the original code [196]. There are other prediction types which we discuss and classify in details in Section 5.

4.1 Supervised learning

Supervised learning is the subclass of machine learning that deals with learning a function from labeled data in the training set [74, 175]. The learner receives a set of labeled examples as training data and makes predictions for all unseen points. This is the most common scenario associated with classification, regression, and ranking problems.

4.1.1 Bayesian Networks. Bayesian Networks (BN) [93, 201] are powerful classifiers to infer the probability distribution variables, which can be binary as well, that characterize a certain attribute such as the optimality of compiler optimization sequences. "A Bayesian Network is a direct acyclic graph whose nodes are variables and whose edges are the inter-dependencies between them" [14, 20]. Computed probabilities of those observed variables categorized under nodes can be further used as input to the model as evidence.

The probability distributions of compiler optimizations depend on the program features and the relationship between the different optimizations that are applied. Ashouri et. al. [14, 20] proposed
Table 4. A Classification Based on Machine Learning Models

<table>
<thead>
<tr>
<th>Classification</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervised Learning</td>
<td></td>
</tr>
<tr>
<td>Bayesian Net</td>
<td>[14, 20]</td>
</tr>
<tr>
<td>Linear Models / SVMs</td>
<td>[17, 18, 20, 68, 196, 198, 213, 229]</td>
</tr>
<tr>
<td>Decision Trees / Random Forests</td>
<td>[39, 75, 90, 97, 99, 102, 147, 160, 164, 176, 197, 245]</td>
</tr>
<tr>
<td>Graph Kernels</td>
<td>[133, 168, 169, 195]</td>
</tr>
<tr>
<td>Unsupervised</td>
<td></td>
</tr>
<tr>
<td>Clustering</td>
<td>[18, 28, 168, 169, 208, 240]</td>
</tr>
<tr>
<td>Evolutionary Algorithms (GAs, NEAT, NN)</td>
<td>[3, 6, 47, 51, 63, 64, 66, 67, 70, 76, 80, 106, 106, 117, 118, 137, 142–144, 146, 146, 147, 147, 148, 151, 155, 160, 169, 181, 183, 195, 199, 205, 208, 227, 227, 228, 230, 236, 250]</td>
</tr>
<tr>
<td>Reinforcement Learning</td>
<td>[63, 170, 171]</td>
</tr>
</tbody>
</table>

a Bayesian Network approach to infer the best compiler optimizations suitable for an embedded processor by means of Bayesian networks. The approach composed of using static, dynamic, or hybrid features of an application as evidence to their trained network. Rather using all sequence speedup pairs for the training of BN, the authors proposed to base their training data on only those instances of compiler sequences having the best 15% speedup w.r.t the GCC’s standard optimization level -O3. This way, BN is trained with only good sequences of optimizations to be used for the inference given a new application. The evaluations were carried out with the Cbench [95] and Polybench [109, 207] benchmark suites and the authors showed that employing BN with an iterative compilation methodology outperform GCC’s -O2 and -O3 by around 50%.

4.1.2 Linear Models and SVMs. Linear models are one of the most popular supervised learning methods to be widely used by researchers in tackling many machine learning applications. “Linear regression, nearest neighbor, and linear threshold algorithms are generally very stable” [74]. Linear models are algorithms whose output classifier does not yield dramatic fluctuation against minor changes in the training set.

Moreover, “SVMs are a type of supervised machine learning technique, can be used for both regression and classification scenarios” [198]. In SVMs, there are algorithms that can adapt linear methods to a non-linear class of problems using a technique called kernel trick [220]. Aside from constructing hyperplane or a set of hyperplanes in an high-dimensional space, SVMs can find the best hyperplane (so-called maximum margin clustering [132, 196]).

Sanchez et al. [213] proposed to SVMs to learn models to focus on autotuning the JIT compiler of IBM Testarossa and the build compilation plan. They used scalar features to construct feature vectors and employed the learning scheme of SVMs to experimentally test the quality of their model using a single-iteration and 10-iteration scenarios on SPECjvm98 benchmark suite.

4.1.3 Decision Trees and Random Forests. A binary decision tree is a tree representation of a partition of the feature space. Decision trees can be defined using more complex node questions resulting in partitions based on more complex decision surfaces [175]. Random forests, or random decision forests, are an ensemble learning method that use multiple learning methods to provide a better prediction for regression and classification purposes. Random forests start by constructing many decision trees at training time and output the class that fits the mode of classes (in the case of classification) or means of classes (in the case of regression). “Random decision forests correct decision trees’ habit of overfitting to their training set” [74, 92].

Fraser et al. [90] proposed to use machine learning to perform code compression. It used the IR structure of codes to automatically infer a decision tree that separates intermediate representation
code into a stream that compresses more efficiently. They evaluated their code compression approach with GCC and used opcodes which can also help predict elements of the operand stream.

Monsifrot [176] addressed the automatic generation of optimization heuristics for a target processor through machine learning. They used decision trees to learn the behavior of the loop unrolling optimization on the code being studied to decide whether to unroll on UltraSPARC and IA-64 machines.

Fursin and Cohen [97] built an iterative and adaptive compiler framework on the SPEC applications using a modified GCC. They constructed a transparent framework which reuses all the compiler program analysis routines from a program transformation database to avoid duplications in external optimization tools.

4.1.4 Graph Kernels. Graph kernels construct a low-dimensional data representation by a cost function that preserves properties of the data. “The use of kernel functions is very attractive because the input data does not always need to be expressed as feature vectors” [138, 183, 195]. Graph kernels are emerging as a means of exploiting many different machine learning applications on a wide range of applications from semi-supervised learning [54] to clustering and classification [46].

Park et. al. [195] proposed the use of graph kernels to characterize an application. The authors used the control flow graph of programs. Instead of flattening the control flow graph information into a fixed-length characterization vector, the authors created a similarity matrix to feed into SVM algorithm using the shortest path graph kernel [44]. The output of the shortest path graph kernel is a similarity score between two graphs. They evaluated the approach by using a model for both non-iterative and iterative fashion on two multicore systems. GCC and Intel compiler were the two back-end compilers once the code passed through a polyhedral source-to-source compiler. Finally, they compared the proposed model against using static features derived by MilepostGCC, performance counters, and 5-Reactions.

4.1.5 Other Supervised Methods. For conciseness proposes, we decided to classify other supervised learning methods under this subsection. These include Neural Networks, ANOVA, K-nearest neighbor, Gaussian process learning [175], etc..

Moss et al. [177] showed a machine learning process to construct heuristics for instruction scheduling, more specifically, scheduling a straight-line code. They used static and IR features of the basic block with the SPEC benchmark to experimentally evaluate their approach by using Geometric mean as fitness function and fold-cross-validation.

Cavazos and Moss [49] used the JIT Java compiler and SPECjvm98 benchmarks with a rule set induction learning model to decide whether to schedule instructions. They exploited supervised learning to induce heuristics to predict which blocks should schedule to maximize the benefit where the induced function acts as a binary classification. The authors experimentally showed they could obtain at least 90% of scheduling improvement for every block while spending at most 25% of the needed effort.

Pinker et al. [202] proposed an automatic iterative procedure to selectively enable or disable compiler options. This procedure is based on orthogonal arrays that are used for statistical analysis of profile information to calculate the main effect of the options. This approach can be used with any arbitrary compiler which allows a collection of options to be set by the user. The authors showed that the proposed approach outperforms GCC’s -O3 on some SPEC benchmarks.

Haneda et al. [111] introduced a statistical model to build a methodology which reduces the size of the optimization search space, thus allowing a smaller set of solutions for exploring strategies. They showed that the technique found a single compiler sequence across a set of SPECint95 benchmarks which could perform better on average against the GCC’s standard optimization set.

Tournavitis et al. [242] proposed a technique using profile-driven parallelism detection in which they could improve the usability of source code features. The proposed approach enabled the authors to identify and locate more parallelism on an application with user feedback required at the final stage. Moreover, the authors exploit a machine-learning based prediction mechanism that replaces the target-specific mapping heuristic. This new mapping scheme made better mapping decisions
while being more scalable and practical to use on other architectures. Finally, they experimentally evaluated their approach with the NAS and SPEC-OMP benchmark suite.

Namolaru et al. [180] proposed a general method for systematically generating numerical features from an application. The authors implemented the approach on GCC. This method does not place any restriction on how to logically and algebraically aggregate semantical properties into numerical features. Therefore, it statistically covers all relevant information that can be collected from an application. They used static features of MilePost GCC and MiBench to evaluate their approach.

4.2 Unsupervised learning

Unsupervised learning is the machine learning task of inferring a function to describe hidden structure from unlabeled data. Since the examples given to the learner are unlabeled, there is no error or reward signal to evaluate a potential solution [112, 175]. Unsupervised learning is closely tightened with the problem of density estimation in statistics [223]; however, unsupervised learning also encompasses many other techniques that seek to summarize and explain key features of the data such as the use of evolutionary algorithms. In unsupervised learning targets don’t exist, but an environment that permits model evaluation does exist. An environment can score a model with some value, i.e. which is the value passed to a model’s objective function.

4.2.1 Clustering Methods.

One of unsupervised learning’s key subclasses is clustering. Clustering helps to downsample the chunk of unrelated compiler optimization passes into meaningful clusters that correspond to each other, i.e. targets loop-nests or scalar values, or they should follow each other in the same sequence. The other importance of clustering and downsampling is to reduce the compiler optimization space, which can be tens of thousands orders of magnitude smaller (mentioned in Section 2.3).

Thomson et al. [240] proposed a methodology to decrease the training time of a machine learning based autotuning. They proposed to use a clustering technique, namely Gustafson Kessel algorithm [31], after applying the dimension reduction process. They evaluated the clustering approach on the EEMBCv2 benchmark suite and showed a factor of seven on reducing the training time with the proposed approach.

Ashouri et al. [15, 28] developed a hardware/software co-design toolchain to explore compiler design space jointly with microarchitectural properties of a VLIW processor. The authors have used clustering to derive to four good hardware architectures followed by mitigating the selection of promising compiler optimization with statistical techniques such as Kruskal-Wallis test and Pareto-optimal filtering. (This method involved with statistical methods as well. Refer to Section 4.1.5)

Martins et al. [168, 169] tackled the problem of phase-ordering by a method 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 optimization space to measure the exploration reduction time in the search space against state-of-the-art techniques such as Genetic algorithm.

4.2.2 Evolutionary Algorithms.

Evolutionary algorithms are inspired by biological evolution such as the process of natural selection and mutation. Candidate solutions in the optimization space play the role of individuals in a population. A common practice to identify the quality of solutions is using a fitness function such as an execution speedup. Evolution of the population takes place after the repeated application of the fitness function [175]. Here we briefly mention some of the more notable techniques used in the literature.

Genetic Algorithm (GA) is a meta-heuristic algorithm under this class which can be paired with any other machine learning technique or be used independently. A notable fast GA heuristic is NSGA-II (Non-dominated Sorting Genetic Algorithm II) [72], which is a popular method for many multi-objective optimization problems and have had numerous applications mostly in the computer
architecture domain \[166, 222\]. NSGA-II is shown to alleviate the computational complexity of classic GA algorithms.

Another interesting evolutionary model is Neuro Evolution of Augmenting Topologies (NEAT) \[226\]. They proved to be a powerful model for learning complex problems since they are able to change the network topology and parameter weight to find the best-balanced fitness function. NEAT specifically has been used in many notable recent research work as well \[63, 146, 147\]. This section summarized a few notable research work that used evolutionary algorithms.

Cooper et al. \[66, 67\] addressed the code size issue of the generated binaries by using genetic algorithms. The results of this approach were compared to an iterative algorithm generating fixed optimization sequence and also at random frequency. Given the comparison, the authors concluded that by using their GAs they could develop new fixed optimization sequences that generally work well on reducing the code-size of the binary.

Knijnenburg et al.\[137\] proposed an iterative compilation approach to tackle the selection size of the tiling and the unrolling factors in an architecture independent manner. They evaluated their approach using a number of state-of-the-art iterative compilation techniques, e.g., simulated annealing, and GAs. The authors compared their results with static techniques. The targeted compiler was native Fortran77 or g77 compiler with full optimization enabled. The benchmarks used were Matrix-Matrix Multiplication (M×M), Matrix-Vector Multiplication (M×V) and Forward Discrete Cosine Transform.

Stephenson et al. \[228\] introduced Meta Optimization, a methodology that automatically can find compiler heuristics for an application under analysis. Meta optimization exploits genetic programming to automatically traverse the optimization space of a given heuristic. The authors targeted the IMPACT compiler with SPEC and Mediabench \[152\] applications to evaluate their approach.

Cavazos and O’Boyle \[51\] developed a GA based approach to automatically tune inlining heuristic on a dynamic compiler. Their approach used a program’s performance to guide the search. Genetic algorithms have been used candidate and the geometric mean of the performance of the SPECjvm98 benchmark was used as their fitness function.

Agakov et al. \[3\] introduced models to speed up on the exploration of the optimization sequences for a most promising region. Their methodology exploits a Markov chain oracle and an independent identically distributed (IID) probability distribution oracle. Each of these oracles used generated models using offline learning to bias certain optimizations over others and replace the uniform probability distribution they applied earlier for the random iterative compilation (RIC) reference methodology. The authors reported significant speedup by coupling these machine-learning models with the nearest neighbor classifier. When predicting the probability distribution of the best optimizations given an unseen application, the classifier first selects the training application having the smallest Euclidean distance in the feature vector space (derived by PCA). Then it learns the probability distribution of the best compiler optimizations for this neighboring application either by means of the Markov chain or IID model. The probability distribution learned is then used as the predicted optimal distribution for the new application. The Markov chain oracle outperformed the IID oracle, followed by the RIC methodology using a uniform probability distribution.

Leather et al. \[151\] used a grammatical evolution-based approach with a genetic algorithm to describe the feature space. The authors showed that the selection of the right features for a compiler can lead to substantial performance improvement to the quality of machine learning heuristic. They described the space formed by the features with a grammar and then they generated many features from it. These generated features are later used in the predictive modeling to search the optimization space using GCC and Mediabench and experimentally validate the proposed approach on a Pentium machine.

Kulkarni et al. developed two approaches in order to tackle both optimization selection \[147\] and phase-ordering \[146\]. The approach for selecting the good compiler passes is done using NEAT and static features to tune the Java Hotspot compiler with SPEC Java benchmarks (using two benchmarks for training and two for testing). The authors used NEAT to train decision trees for
the caller and the callee whether to inline. When addressing the phase-ordering problem, they proposed an intermediate speedup prediction method that used static features of the current state of the application being studied to query the model and induce the current best optimization to use. This way, iteratively a compiler sequence is formed on an application-based manner.

Purini et al. [208] defined a machine learning based approach to downsample the compiler optimization sequences in LLVM’s -O2 and applied machine learning to learn a model. The authors introduced a clustering algorithm to clustering sequences based on the Sequence Similarity matrix by calculating the Euclidean distance between the two sequence vectors. In the experimental evaluation, they have mentioned the most frequent optimization passes with their fitness function (execution speedup) as well.

4.3 Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning which can not be classified as supervised or unsupervised. It is inspired by behaviorist psychology and uses the notion of rewards or penalties so that a software agent interacts with an environment and maximizes his cumulative reward. The interesting difference in RL is that the training and testing phases are intertwined [175]. RL uses Markov decision process (MDP) [119] to adapt and interact with its environments. In this section, we have provided the works done with RL in the field.

McGovern et al. [170] presented two methods of building instruction scheduler using rollouts, an improved Monte Carlo search [238], and reinforcement learning. The authors showed that the combined reinforcement learning and rollout approach could outperform the commercial Compaq scheduler on evaluated benchmarks from SPEC95 suite.

Coons et al. [63] used NEAT as a reinforcement learning tool for finding good instruction placements for an EDGE architecture. The authors showed that their approach could outperform state-of-the-art methods using simulated annealing in order to find the best placement.

5 PREDICTION TYPES

In this section, we provide a full classification of the output of the aforementioned prediction models. The major classes of prediction type include the target function to predict (i) the right set of compiler optimizations to be used, (ii) the speedup of a compiler optimization sequence given an application, (iii) the right set of features to be used in order to characterize the application, (iv) the intermediate speedup (tournament) predictor, and (v) the reduction of the search space through down-sampling of optimizations. Table 5 shows the classification of the related literature.

5.1 Clustering and Downsampling

Clustering, or cluster analysis, is the task of investigating the similarities between a set of variables in such a way that similar variables can be placed into the same group called a cluster. Clustering is most commonly used in the field of data mining and a common technique for statistical data
analysis. Cluster analysis is used in many fields including unsupervised machine learning, pattern recognition, and compiler autotuning [175]. In order to approach reasonable solutions to the compiler optimization problems, the optimization search space needs reduced. Researchers try to find ways to decrease the enormous size of the optimization space by orders of magnitudes. This technique, known as downsampling, has been used in many recent works. We mentioned all these papers in the table 5.

Purini et al. [208] defined a machine learning based approach to downsample the compiler optimization sequences in LLVM’s -O2 and then applied machine learning to learn a model. They preferred not to use features from the application under analysis. Instead, their downsampling technique used various genetic algorithms or a uniform random search. Moreover, they have introduced a clustering algorithm to cluster the sequences based on Sequence Similarity matrix by calculating the Euclidean distance between the two sequence vectors. In their experimental evaluation, they have mentioned the most frequent optimization passes with their fitness function (execution speedup).

Ashouri et al. [18] presented a full-sequence speedup predictor for the phase-ordering problem that rapidly converges to optimal points and outperforms both standard optimization levels and the state-of-the-art ranking approach [195, 196]. The authors applied a clustering technique on all optimization passes available in LLVM’s -O3 and trained their model using the resulted subsequences. The infeasibly large phase-ordering space is reduced to a fairly big yet explorable using iterative compilation and a machine learning method. The authors showed their approach could outperform LLVM’s highest optimization level and other existing speedup predictors with just a few predictions.

5.2 Speedup Prediction

Speedup predictive modeling is the process of constructing, testing, and validating a model to predict an unobserved timing outcome. The model is constructed based on the characterization of a state. The state being characterized is the code being optimized and the predicted outcome corresponds to the speedup metric calculated by normalizing the execution time of the current optimization sequence by the execution time of the baseline optimization sequence. “The general form of a speedup predictor is to construct a function that takes as input a tuple \((F, T)\), where \(F\) is the collected feature vector of the of an application being optimized” [196]. \(T\) can be one of the many possible compiler optimization sequences in the design space. The model’s output is the predicted value of a speedup that the sequence \(T\) is to achieve when applied to the application’s source-code in its original state. This prediction type is one of the more widely used among the researchers and here we mention some of its notable usages.

Dubach et al. [76, 77] proposed a speedup predictor based on the source code features of the optimized applications. The authors used static features from SUIF compiler infrastructure [257] for VLIW and compared the result with non-feature-based alternative predictors such as mean predictors, sequence encoding-based predictors, and reaction based predictors.

Park et al. [195–199] proposed several predictive modeling methodologies to tackle the problem of selecting the right set of compiler optimizations. Authors used static program features instead of hardware-dependent features on a polyhedral space [196]. In [195], the authors used Control Flow Graphs (CFGs) with graph kernel learning to construct a machine learning model. First, they constructed CFGs by using the LLVM compiler and convert the CFGs to Shortest Path Graphs (SPGs) by using the Floyd-Warshall algorithm. Then, they apply the shortest graph kernel method [44] to compare each one of the possible pairs of the SPGs and calculate a similarity score of two graphs. They calculated similarity scores for all pairs are saved into a matrix and directly fed into the selected machine-learning algorithm, specifically SVMs in their work. In [197], they used user-defined patterns as program features. They use a pattern-driven system named HERCULES [128] to derive arbitrary patterns coming from users. They focused on defining patterns related to loops: the number of loops having memory accesses, having loop-carried dependencies, or certain types of
data dependencies. These works use static program features mainly focusing on loop and instruction mixes.

5.3 Compiler Sequence Prediction

A compiler sequence predictor is a type of prediction model which outputs the best set of compiler passes or sequences to apply on a given application. Application characterization is fed to this model, and the model induces a prediction of a set of compiler passes \( \theta \in \mathcal{O} \) to apply to the given application. The objective could be configurable – it could maximize its performance or fulfill other objectives such as code size or energy consumption. [14, 20, 198]. This specific prediction type has attracted the most interest among research and we have seen the bulk of work addressing this. Here, we discuss a few notable contributions.

Cooper et al. [66, 67] proposed an adaptive compiler that used a biased random search to find an application specific compilation sequence that satisfies user objective functions. The adaptive compiler finds its optimization candidate within its pool of available optimization passes, objective function, and the target machine under analysis.

Cavazos et al. [48] investigated the problem of selecting the right set of compiler optimizations by means of dynamic characteristics of an application under analysis. The authors used logistic regression for model construction and showed that using their approach could outperform existing techniques used by static code features and rapidly reaching the achievable speedup.

Ashouri et al. [14, 20] proposed a Bayesian Network (BN) approach which was fed by either of static, dynamic or a hybrid characterization vector of an application. The BN model could subsequently induce a probabilistic model. The authors combined their model with iterative compilation to derive good compiler sequences and experimentally showed they could outperform the state-of-the-art models.

5.4 Tournament and Intermediate Prediction

A tournament predictor, proposed by [198], predicts a binary classification between two input optimization sequences. This type of predictor can rank compiler sequences accordingly and select the best among them. On the other hand, intermediate speedup predictor (also referred to as Reaction-based modeling [47]) [17, 146] is a prediction model that tends to iteratively predict the speedup of the current state of an application being optimized. Applications’ characteristics in each state along with the compiler sequence \( T \) serves as input, and the model predicts the speedup \( T \) should achieve. Since intermediate speedup predictors behave more or less the same as a tournament predictor, as both work on individual optimizations at a time to be applied on the current state, we decided to group these two under one classification. An intermediate sequence approach needs multiple execution profiles of the application being optimized; therefore, it tends to be slower in comparison to other methods. Nevertheless, it has been shown an effective method specifically to tackle the phase-ordering problems [17, 146].

Park et al. [198] have proposed tournament predictors in order classify whether an optimization should be applied as an immediate optimization. Park et al. evaluated three prediction models using program counters (PC) derived by PAPI [179]: sequence, speedup, and tournament on several benchmarks (PolyBench, NAS, etc.) using the Open64 compiler. They showed on many occasions tournament predictors can outperform other techniques.

Kulkarni and Cavazos [146] developed an intermediate speedup predictor using NEATs that selects the best order of compiler optimization passes iteratively based on its current state. They evaluated their approach using JAVA jikes dynamic compilers and observed on average 5-10% speedup when adjusting the different ordering of the phases.

Ashouri et al. [17] demonstrated a predictive methodology in order to predict the intermediate speedup obtained by an optimization from the configuration space given the current state of the application. The authors used dynamic features of an application for their prediction model unlike the work of Kulkarni and Cavazos [146]. They also used speedup values between the execution
times of the program before and after the optimization process as their fitness function and defined heuristics to traverse the optimization space.

5.5 Feature Prediction

Constructing a model from empirical data must have minimal user intervention. The accuracy of predictions is heavily related to the type of features and characterization collected from an application under optimization process. Therefore, feature prediction models are advantageous to use for learning the most promising features affecting performance. During this process, choosing the right set of features to characterize an application is crucial. Here we discuss a few notable research works on this matter.

Vaswani et al. [245] used empirical data to construct a model that is sensitive to microarchitecture parameters and compiler optimization sequences. The proposed model learns the relationship automatically from data and constructs automatic heuristics. The authors evaluated their approach using SPEC-CPU2000 and GCC compiler and on average attained performance improvement over 10% higher than the highest standard optimization levels available with GCC.

Li et al. [155] used a machine learning based optimization focused on feature processing. The authors adapted this method based on the application under analysis and apart from predefined source code features. They developed an approach to automatically generate several features of an application to be fed to the model construction using a template. Furthermore, they observed fluctuation on the values of the features when using different compiler optimizations and alleviated this by designing a feature extractor to iterative extract the needed features at run-time and fed them to the optimization model dynamically.

Ding et al. [75] proposed an autotuning framework capable of incorporating different input in a two-level approach to overcome the challenge of input sensitivity. They leveraged the Petabricks language and compiler [10] and used an input-aware learning technique to differentiate between inputs. The work clustered the space and chose its centroid for autotuning (i) to identify a set of configurations for each class of inputs and (ii) to identify a production classifier which allowed them to efficiently identify the best landmark optimization to use for a new input.

6 OPTIMIZATION SPACE EXPLORATION TECHNIQUES

As previously mentioned in Section 2.3, traversing the large compiler optimization space made by the different combinations of optimizations requires a proper exploration strategy. Iterative compilation and genetic algorithms are among the most mentioned strategies in the literature. In a broader perspective, the strategies are derived by the desired type of optimization space exploration. Design space exploration (DSE) is the activity of exploring design alternatives before implementation. Historically, it has been used at system-level design, but the methodologies can be adapted to use at compiler-level exploration as well [28]. DSE helps to define and follow exploration policies that undertake generation of candidates within certain or all optimization space and has been proven useful for many optimization tasks [140]. In general, different applications could impose different energy and performance requirements. The main goal of this phase is to efficiently traverse and configure the exploration parameters [189, 191]. In this section, we classify the different exploration strategies used by researchers in literature to overcome this challenge. Table 6 represents our fine-grain classification of the different related works based the exploration type.

6.1 Adaptive Compilation

Adaptive optimization [251], also known as profile-guided optimization, is a technique where the optimization space is explored based on the outcome of fitness functions, e.g., execution time to profile the executable and dynamically modifies/recompiles certain segments of an application under optimization. The profiling provides enough features so the compiler can decide on what portion of the code to be recompiled. An adaptive compiler is placed between a just-in-time (JIT) compiler and an interpreter of instructions.
Table 6. A Classification Based on Space Exploration Methods

<table>
<thead>
<tr>
<th>Classification</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-iterative</td>
<td>[90, 165, 205, 206, 209, 227, 246, 253, 254]</td>
</tr>
</tbody>
</table>

As suggested by [64], an adaptive compiler can benefit from a compilation-execution feedback loop to select the best optimization that satisfies the objectives of a scenario. The following works go over the state of the art and practice with profile-guided optimization.

Cooper et al. [64, 67] developed an adaptive compiler, ACME, and a user-interface to control the process of recompilation and exploration of different compiler optimizations. The authors introduced virtual execution – a technique to mitigate the concurrent execution of code being optimized. By running an application once, this technique helps to keep a record of information needed to estimate the run-time performance of different compiler optimization sequences without the need to re-run the application again. This technique was later referred to as speedup prediction [17, 146, 198].

Fursin and Cohen [97] built an iterative and adaptive compiler framework targeting the SPEC benchmark suite using a modified version of GCC. The authors also developed a transparent framework which reused all the compiler program analysis routines from a program transformation database to avoid duplicates in external optimization tools.

### 6.2 Iterative Compilation

In computer science, an iterative method is a sequence of approximated procedures that are applied to further improve the quality of the solution of a problem. Iterative compilation is the most commonly used exploration technique for the compiler optimization field. Many recent works found this technique interesting and successful either (i) alone [40, 138], (ii) combined with machine learning techniques [3, 20, 48], or (iii) combined with other search and meta-heuristics techniques such as random exploration [40] and DSE techniques [168, 182].

Bodin et al. [40] investigated an early path towards analysis of the applicability of iterative search techniques in program optimization. The authors showed that iterative compilation is feasible to practice on embedded applications where the primary cost is paid back by fabrication and distribution at scale. An embedded application usually considers with fewer parameters, so the cost of iterative compilation is worthwhile on small parameter counts. They used profile feedback in the form of execution time and to downsample the space on restricted optimization passes. In this work, the authors investigated the unrolling, tiling, and padding parameters; however, in the latter works other researchers expanded the experimental parameters to scalar, loop-nest, and other optimizations [15, 28, 81, 103, 134–136, 198, 205].

Triantafyllis et al. [243] proposed a generic optimizer that used practical iterative compilation framework called Optimization-Space Exploration (OSE). Their approach involved using compiler writer’s knowledge to prune and exclude several configuration parameters in the design space. The OSE had function granularity and could apply different optimizations on the same code segments. They used the iterative notion of selecting the next compiler optimization to be used based on the current state of the application being optimized. A sub-classification of this approach was later named as intermediate speedup prediction [17, 146].

Killian et al. [131] incorporated iterative compilation to drive to automatic vectorization. The authors evaluated their approach using an SVM and predict promising compiler optimization given an application under analysis.
6.3 Non-iterative Compilation

Unlike iterative compilation, a non-iterative approach tries to present a global optimization approach for a class of compiler optimization problems. Fewer recent works are observed tackling compiler optimization problems using this method [154]. As we noted in Section 1, the driving force towards using approximation methods such as iterative compilation and machine learning was the inability of researchers to tackle the phase-ordering problem by straightforward non-iterative approaches. Thus, this branch of compiler autotuning has suffered from further investigation. The polyhedral compilation community has gained attention in many interesting directions and is an orthogonal approach to optimizing compilers. We address polyhedral compilation in Section 7.

Vegdahl et al. [246] have used constant-unfolding to produce code sequences that can be more compacted on a horizontal target architecture. A constant-unfolding axiom replaces a constant by a constant expression of equal value. The goal is to make use of constants which are hard-wired into the micro machine, replacing difficult-to-generate constants with expressions involving only hard-wired constants.

Whitfield et al. [254] proposed a framework that investigates the use of Gospel specifications [252] for improving the performance of the code being optimized. The authors implemented a tool called Genesis which was able to transform codes based on the Gospel specifications. They demonstrated the benefits of the proposed framework through binary (enabling or disabling) exploration of compiler optimizations.

7 TARGET DOMAIN

Finding the best set of compiler optimizations to apply on a given application is heavily correlated with the type of compiler, target processor architecture, and target platform to be tuned. Research on compiler autotuning has tried to avoid generic optimizations due to this very reason. By employing machine learning models, a framework should be adaptable based on a given application or target platform. An optimized code segment for a given compiler or a target platform may not yield the same optimality on a different compiler or platform. Each compiler and target platform combination have different ways of generating the binaries and executing the code segments. Moreover, optimization techniques might be useful for a class of applications, e.g., security, scientific, etc., but they might not be for other classes. To this end, we classify the literature based on the two aforementioned subclasses. These are shown in Tables 7, and 8.

7.1 Target Platform

Today, essentially all programming for desktop and HPC application is done using high-level languages (as is most programming for embedded applications). This observation implies that since most instructions executed are the output of a compiler, an instruction set architecture is essentially a compiler target [200]. For example, consider special loop instructions found in an application. Assume that instead of decrementing by one, the compiler wanted to increment by four, or instead of branching on not equal zero, the compiler wanted to branch if the index was less than or equal to the limit. As a result, the loop instruction may be a mismatch having different target instruction sets ISAs or architectures. Choosing the right set of optimizations given an architecture is a necessary task. A classification based on the type of target platform is shown in Table 7.

7.1.1 Embedded Domain. In terms of embedded computing, embedded encompasses nearly all computing that is not considered general purpose (GP) and High Performance Computing (HPC). Embedded processors include a large number of interesting chips: those found in cars, mobile phones, pagers, handheld consoles, appliances, and other consumer electronics [88]. Some of the more notable embedded architectures are Very Long Instruction Word (VLIW) [85, 86] and the big.LITTLE heterogeneous architecture from ARM [124]. There are many recent low-cost implementation boards with different system-on-a-chip (SoC) specifications such as Raspberry Pi [244] and Texas Instrument’s Pandaboard [120]. Readers can refer to the already available surveys in the field of embedded computing and FPGAs [12, 62, 113].
One of the key differences of leveraging compiler optimization techniques for the embedded domain is the trade-off between the application’s code-size, performance, and power consumption. Code-size optimization specially in VLIW architecture has been extensively investigated [82, 161]. However, due to recent advancements in the embedded SoC code-size is no longer the main issue. Thus, the focus has shifted towards the Pareto-frontiers of performance, power, and energy metrics [13, 189, 191]. Compiler optimization techniques can be exploited for this task as well [28, 85, 181].

Dubach et al. [76] presented a new technique to automatically induce a speedup prediction based on the characteristics of the source code of an application under optimization. The authors used static-features from the SUIF infrastructure [257] compiler toolchain for VLIW, and compared the result with non-feature-based alternative predictors such as mean predictor, sequence encoding-based, and reaction based predictors.

Namolaru et al. [180] proposed a general method for systematically generating numerical features from an application. The authors implemented their approach on top of GCC. This method does not place any restriction on how to logically and algebraically aggregate semantical properties into numerical features; therefore, it statistically covers all relevant information that can be collected from an application. They used static features of MilePost GCC and MiBench to evaluate their approach on an ARC 725D embedded processor.

7.1.2 Desktop and Workstations. Despite the fact that the majority of the research has been done for desktops and workstations, the focus has recently shifted towards the both ends of the architectural spectrum, namely embedded and high performance computing (HPC). In this section, we have classified those works by their experimental setup where the target platform was not either of the two ends, thus we classify them as desktops and workstation category. This classification is shown in Table 7.

Thomson et al. [240] proposed a clustering technique to decrease the offline training that normally supervised machine learning techniques have. The authors achieved this goal by focusing only on those applications which best characterize the optimization space. They leveraged the Gustafson-Kessel algorithm after applying the dimension reduction process and evaluated their clustering approach with the EEMBCv2 benchmark suite and an Intel Core 2 Duo E6750 machine. They experimentally showed that employing technique could drastically reduce the training time.

Park et al. [198] proposed the term tournament predictor in order classify whether an optimization was better off to choose or not as an intermediate optimization. The authors evaluated all three prediction models namely, sequence prediction, speedup, and tournament prediction using program counters (PC) derived from PAPI on Polybench, NAS, etc. and other benchmarks using Open64 compiler. Their target machines were two different Intel Core 2 Quad processors and an Intel Xeon E5335. The authors observed that on average those good compiler sequences could work well across other experimentally tested machines as well.

7.1.3 HPC Domain. There are fundamental differences between a cluster and supercomputer. For instance, mainframes and clusters run multiple programs concurrently and support many concurrent users versus supercomputers which focus on processing power to execute a few programs or instructions as quickly as possible and to accelerating performance to push boundaries of what
hardware and software can accomplish [45, 235]. However, for conciseness purposes in this survey, we have placed the recent works having used mainframes and clusters together with those having supercomputers as their experimental setup.

Tiwari et al. [241] proposed a scalable autotuning framework that incorporated Active Harmony’s parallel search backend [234] on the CHiLL compiler framework [56]. The proposed methodology enabled the authors to explore the search space in parallel and rapidly find better transformed versions of a kernel that bring higher performance gain.

Ding et al. [75] presented an autotuning framework capable of leveraging different input in a two-level approach. The framework is built upon the Petabricks language and its compiler [10]. It uses input-aware learning technique to differentiate between inputs, clusters the space, and chooses its centroid for autotuning. The two-level approach consists of identifying a set of configurations for each class of inputs and producing a classifier to efficiently identify the best optimization to use for a new input.

Fang et al. [81] proposed IODC; an iterative optimization approach which could be deployed on data centers. They used intrinsic characteristics of data centers to overcome the well-known hurdle of running an iterative compilation method, since the technique normally requires a large number of runs per each scenario. IODC approach this challenge by spawning a large number of iterative compilation jobs at once to many workers and collect their performance back to a master node so that an optimal compilation policy can be rapidly found. Moreover, they evaluate their approach using a MapReduce and a compute intensive approach by, e.g., âÅ§0x flags. They used the clang compiler targeting a multicore ARM processor in an ODROID board and a dual x86 desktop representative of a node in a supercomputing center.

### 7.2 Target Compiler

In instruction-level parallel (ILP) and super-scalar architectures [248], parallelism mainly comes from the compiler heuristics. This insight is key to reach peak performance, energy efficiency, and lower power consumption. Therefore, it is important to guide the compiler in order to look for the trade-off that satisfies specific objectives on a platform. “The typical investment for a compiler back-end before maturity is measured in man-decades, and it is common to find compiler platforms with man-century investments” [85, 87]. Table 8 classifies the recent literature based on the type of the compiler framework used.

#### 7.2.1 GCC

GNU compiler collection (GCC) is the GNU compiler and toolchain project which supports various high-level languages, e.g., C, C++, etc.. “The Free Software Foundation (FSF) distributes GCC under the GNU General Public License (GNU GPL). GCC has played an important

| Compilers          | References                                                                 
|--------------------|-----------------------------------------------------------------------------
| LLVM               | [15, 17, 18, 20, 27, 28, 39, 69, 107, 168, 169, 181–184, 196, 199, 208]          |
| Intel-ICC          | [42, 58, 89, 130, 131, 158, 195, 197, 199, 205, 217, 232, 261]                   |
| JIT Compiler       | [49–52, 118, 121, 146, 147, 199, 209, 213, 217, 227]                           |
| Java Compiler      | [49–52, 118, 121, 146, 209]                                                  |
| Polyhedral Model   | [41, 42, 196, 199, 204–206, 241, 249, 261, 262]                               |
role in the growth of free software, as both a tool and an example” [224, 225]. This project has been ported to various processor architectures, including most embedded systems (ARM, AMCC, and Freescale), and is able to be used on many target platforms. Due to the wide support and open-source nature of GCC, it has been the center focus of researchers on compiler autotuning methodologies. It is worth noting that GCC out-of-the-box does not support tuning with the phases of its internal compiler passes as its pass manager overrides predefined ordering. However, modifying the pass manager enables tackling such compiler optimizations problem. The GCC optimizer supports different predefined levels of fixed optimization levels such as standard levels -Ofast, -O1, -O2 and -O3.

7.2.2 LLVM. The LLVM Project is a collection of modular and reusable compiler and toolchain technologies used to develop compiler front ends and back ends. Latner and Vikram [150] described LLVM as “a compiler framework designed to support transparent, lifelong program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs.” LLVM brings many interesting features: (i) It has a common intermediate representation of code called LLVM-IR in static single assignment (SSA) form. (ii) Its C language frontend system called clang offers many pragma-based extensions languages. (iii) Its backend has been ported to various architectures such as x86-64, ARM, FPGA, and even GPUs. Recently, LLVM’s community has become a vibrant research community towards porting and building new features into the different LLVM sub-modules e.g., opt (optimizations), clang (C language family front-end), and llc (code generator). There are also many research papers associated with using and building LLVM 6.

7.2.3 Intel-ICC. Intel’s propriety compiler (ICC) 7 provides general optimizations, e.g., -O1, -O2, -O3, and processor’s specific optimizations depending on the target platform. Moreover, Interprocedural Optimization (IPO) is an automatic, multi-step process that allows the compiler to analyze your code to determine where you can benefit from specific optimizations. With IPO options, you may see additional optimizations for Intel microprocessors than for non-Intel microprocessors.

Franke et al. [89] proposed an approach called source-level transformation which used feedback-driven search in order to find good optimization sequences by their probabilistic score. Using this fitness function, the authors focused on a local search on more promising areas and evaluated their approach to UTDSP benchmark [153] using an embedded processor. They observed performance gain of 1.22x when using ICC versus GCC.

7.2.4 Just-in-time Compiler. Just-in-time compilation (JIT), also known as dynamic translation, is a widely known technique that has been used for many decades. “Broadly, JIT compilation includes any translation performed dynamically after a program has started execution” [30]. High-level benefits of using a JIT compiler can be summarized as: (i) Compiled programs run faster, especially if they are compiled into a form that is directly executable on the underlying hardware. (ii) Interpreted programs tend to be more portable. (iii) Interpreted programs can access run-time information. There are many implementations of JIT compilers targeting different programming languages. Majic, a Matlab JIT compiler [7], OpenJIT [185] a Java JIT compiler, and IBM’s JIT compiler targets the Java virtual machine [233].

Sanchez et al. [213] used SVMs to learn models to focus on autotuning the JIT compiler of IBM Testarossa and the build compilation plan. They experimentally evaluated the learned and observed that the models outperforms out-of-the-box Testarossa on average for start-up performance, but underperforms Testarossa for throughput performance. They also generalized the learning process from learning on SPECjvm98 to DaCapo benchmark.

7.2.5 Java Compiler. A Java compiler is specifically designed to compile Java high-level programming code to emit platform-independent classes of Java bytecode. Subsequently, the Java

---

5https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
6http://llvm.org/pubs/
7https://software.intel.com/en-us/intel-compilers
virtual machine (JVM) can load these files and (i) emit a JIT, or, (ii) interpret the bytecode into target platform machine code. The optimizations can possibly be done in the process, technically classifying them under HIT compilation as well [2, 126]. Some of the notable research works [49, 118] including a work tackling the phase-ordering problem have been done using Java JIT compiler [146]. We have already mentioned these in the Section 4.2.2.

7.2.6 Polyhedral Model. Polyhedral compilation community has recently gained attraction due to its wide appliance on the high performance computing projects. "Polyhedral compilation encompasses the compilation techniques that rely on the representation of programs, especially those involving nested loops and arrays, thanks to parametric polyhedra or Presburger relations" [83, 255]. Wide range of compute-intensive and research applications spend the majority of their execution time in loop-nests and this suitable for targeting high-level compiler optimizations [247]. Polyhedral compilation can address many of these challenges by efficiently transforming their loop-nest [35, 42, 43, 159, 205]. We refrain from focusing more on this interesting topic as it is outside the scope of this survey.

7.2.7 Other Compilers. In this survey, we focused on the classification of the more widely known compiler framework in autotuning field. However, there are numerous other well known compilers which worth mentioning including bot not limited to Cosy [9] and SUIF [257]. We classified all work related to use of other compiler in Table 8 under Others subfield. Here we review a couple of them.

Stanford University Intermediate Format (SUIF) compiler is a free and open-source framework implemented to support collaborative research in parallelizing and applying high-level optimizations for compilers. It supports both Fortran and C as input languages and it can be built on top of the application. “The kernel defines the intermediate representation, provides functions to access and manipulate the intermediate representation, and structures the interface between compiler passes” [257]. It has been used in many research works of recent literature [55, 59, 76].

Fortran compilers include many different implementations and different ports such as Open64 [73], GNU Fortran [221], XL Fortran [141], Salford Fortran 77 compiler [149], etc. and are widely used in literature [6, 137, 214, 215].

8 MOST INFLUENTIAL PAPERS
Evaluating a research work is no easy task and often involves human error. However, in this survey, we assess and present influential work using their scientific breakthrough, novelty, and a competitive citation metric. We discretized the process by presenting the influential papers by their corresponding topic and elaborate more on their proposed approach. Some had effects on their succeeding work, and this was taken into consideration as well.

8.1 Breakthroughs by topic
The following paragraphs highlight novel research in the areas of initial introduction of learning methods with compiler optimizations, genetic algorithms, phase ordering, iterative compilation, dynamic and hybrid features, creating optimization groups with Bayesian learners, and clustering of optimizations to tackle the phase-ordering problem.

Introducing Learning Methods. Leverett et al. and Vegdahl et al. [154, 246] were the first to perform non-iterative optimization without leverage machine learning. [253] extended this work by proposing intelligent ordering of a subset of compiler optimizations. [254] continued their phase-ordering work with a formal language, Gospel, which could be used to automatically generate transformations. The first usage of machine learning techniques arrived with [138] and their work with predicting the optimal unroll size for nested loops. [177] was the first to use machine learning techniques to construct flexible instruction schedules, paving the way for continued efforts leveraging ML techniques.

http://polyhedral.info/
Genetic Algorithms. Cooper et al. [66] expanded machine learning efforts with optimization selection using genetic algorithms with iterative compilation. [67] expanded their prior work by switching to adaptive compilation and one of the earliest works creating an adaptive compilation framework. Predictive modeling was first introduced by [243] where they applied iterative compilation of the SPEC benchmarks. [137] proposed iterative compilation to select tile and unroll factors using genetic algorithms, random sampling, and simulated annealing. They were able to show that their method worked on many different architectures.

Phase Ordering. Kulkarni et al. [142] was one of the first to propose solving the phase-ordering problem using machine learning by combining iterative compilation and meta-heuristics. [146] tackled the phase-ordering problem within the JIKES Java virtual machine. They leveraged static features fed into a neural network generated with NEAT to construct good optimization sequence orders.

Iterative compilation. Bodin et al. [40] proposed an intelligent iterative compilation method which explored less than 2% of the total space in a non-linear search space. [3] used Markov chains to focus iterative optimization using static features. Using a relatively small exhaustive search space for learning (14^5) and a large test space for testing (80^20), they were able to achieve up to 40% speedup.

Dynamic and Hybrid Features. The first use of dynamic features for learning was introduced by Cavazos et al. [48]; they showed that using dynamic features for learning outperformed the use of static features. The advancement of multivariate (static, dynamic, hybrid) feature selection and learning algorithms paved the way for tournament predictors introduced by [198].

Hybrid Characterization and Bayesian Learners. Massive dataset analysis on over 1000 benchmarks was performed by [57, 58]. They proposed optimization sequence groups (beyond traditional compilers’ -O3 baseline) that are, on average, beneficial to use on the applications in their dataset. Most recently, Ashouri et al. [14, 20] used the output of the passes suggested by [57] to construct a Bayesian network to identify the best compiler flags for a given application using static, dynamic, and hybrid features. The Bayesian network generated optimization sequences resulted in application performance outperforming existing models.

Optimization Clustering and Full-sequence Predictors. Ashouri et al. [18] introduced MiCOMP framework to cluster all the LLVM’s -O3 optimization passes into optimization sub sequences and introduce the first full-sequence speedup predictor for the phase-ordering problem. The authors leveraged a recommender systems [210] approach to defined exploration policy using dynamic information stored in the pair-wised optimizations across the training applications to outperform the state-of-the-art ranking approach. They show MiCOMP can achieve over 90% of the available speedup and outperform -O3 using just a few predictions.

8.2 Breakthroughs by Performance

This section provides a brief quantitative comparison of the proposed approaches concerning their reported results. Evaluating papers using their performance is a hard task and often involves comparison errors [115], i.e., using absolute vs. relative speedup values, averaging using different techniques, etc.. To this end, we look into papers which explicitly provided comparison of their proposed approach against their baseline or the state-of-the-art methods.

8.2.1 Iterative Compilation. Random Iterative compilation (RIC) is known to achieve good results when compiling long running applications [40]. However, the approach is expensive and should be combined with an intelligent search algorithms, such as machine learning techniques [3, 18, 20, 40]. Early work in iterative compilation methods [1, 40] involved the exploration of non-linear transformation spaces and found the fastest execution time in a fixed number of evaluations targeted to embedded domain. In embedded domains, the cost of performing iterative compilation is amortized over a large number of devices. Researchers used profile information in the form of
execution time, searched a large but restricted subset of transformations to find good results, and often achieve large speedups by exploring only a small fraction of the optimization space. Other work outperformed these approaches by introducing more efficient search algorithms on the space in addition to considering a more extensive optimization space with different unroll factors and tile sizes [137]. Mpeis et al. [178] implemented an iterative compilation approach targeted to mobile devices to further optimize the Java bytecode of day-to-day running application when the device was not being used. The authors observed on average a 57% performance improvement using a benchmark with minimal slowdowns. Other major work tried to incorporate iterative compilation methodology with machine learning techniques and we will discuss this category next.

8.2.2 Machine Learning Techniques.

The Selection Problem. Agakov et al. [3] introduced a machine learning approach to focus on iterative compilation. Their approach using static features and Markov oracle led to 22% and 27% performance improvement on a Texas instrument and an AMD architecture. Later, Cavazos et al. [48] outperformed the previous work by 7% using a dynamic feature characterization method. The authors believed that using static features of an application although is good enough for embedded multimedia applications but cannot perform well on a large scale general purpose applications. The use of predictive modeling approaches went on by other authors [95, 102, 117]. Park et al. [198] later introduced a novel approach using tournament predictors (see Section 5.4), by which they outperformed existing methods by around 6%. The authors went on to propose to use graph kernels [196] (see Section 5.4) as a means of improving existing prediction models. Ashouri et al. introduced COBAYN [20], which used hybrid features and Bayesian network learners. The authors showed that this approach can outperform existing methods [3, 199] by around 11%.

The Phase-ordering Problem. Cooper et al. [67] proposed a search mechanism for finding good ordering of phases using genetic algorithm. The approach relied upon an adaptive compilation flow that could gain up to 49% speedup on the given benchmark. Later, Kulkarni et al. [142] proposed yet another genetic algorithm which was able to drastically reduce the search time up to 65%. The first intermediate speedup predictor was later proposed by Kulkarni and Cavazos [146]. The authors used the characterization of the current state of the code and NEAT (refer to Section 4.2.2) and gained up to 20% speedup on applications using Java Jikes compiler. Ashouri et al. proposed MiCOMP [18], which uses subsequences derived from LLVM’s -O3 and a full-sequence speedup predictor. The authors showed that using their approach they could outperform existing approaches [17, 146] by 5% and 11%, respectively.

8.2.3 Multi objective Optimization. In this subsection, we briefly discuss the quantitative results of investigating the major optimization objectives drawn in the literature. These objectives are code size, area, power metrics, and performance metrics.

Code Size and Area. Early works were targeting code-size reduction as an optimization objective. Cooper et al. [66] introduced a Genetic algorithm (GA) method to search for reduced code size and found it had an advantage of 14.5% against a default fixed sequence. These works were followed by a series of others mostly tackling the code size reduction in VLIW embedded architecture [13, 28, 88, 259]. However, through the advancement of storage systems specifically in the embedded domain, this issue has become far less of a concern and often has been neglected in the recent literature.

Performance and Intensity. Roofline model [256] relates processor performance to off-chip memory traffic, thus provides a theoretical upper bound for operational intensity and attainable performance in modern architectures. Certain works have used the notion to model their optimization approaches and find a Pareto curve to satisfy both objectives [13, 15, 28, 72, 117, 191, 263]. Hoste and Eeckhout [117] tackled iterative compilation on an Intel Pentium machine and observed the feasibility of their approach together with a multi-objective search algorithm called multiple populations which finds Pareto curves within the different population of an optimization space. They observed
Table 9. A Classification of Top 15 Influential Papers by their Citation Count

<table>
<thead>
<tr>
<th>No.</th>
<th>Reference</th>
<th>Citation Count</th>
<th>ACPY</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Agakov et al. [3]</td>
<td>362</td>
<td>32</td>
<td>iterative compilation, static features, Markov chain oracle, PCA</td>
</tr>
<tr>
<td>2</td>
<td>Cooper et al. [66]</td>
<td>312</td>
<td>17</td>
<td>genetic algorithm, iterative compilation, reduced code-size</td>
</tr>
<tr>
<td>3</td>
<td>Stephenson et al. [2003] [228]</td>
<td>271</td>
<td>19</td>
<td>iterative compilation, genetic programming, Mediabench, metaheuristics</td>
</tr>
<tr>
<td>4</td>
<td>Triantafyllis et al. [243]</td>
<td>268</td>
<td>19</td>
<td>iterative compilation, SPEC, predictive modeling</td>
</tr>
<tr>
<td>5</td>
<td>Cooper et al. [67]</td>
<td>234</td>
<td>15</td>
<td>adaptive compilation, biased random search, compiler sequence predictor</td>
</tr>
<tr>
<td>6</td>
<td>Knijnenburg et al. [137]</td>
<td>229</td>
<td>16</td>
<td>iterative compilation, unrolling factor, architecture-independent manner</td>
</tr>
<tr>
<td>7</td>
<td>Tournavitis et al. [242]</td>
<td>210</td>
<td>26</td>
<td>static-analysis, profile-driven parallelism, NAS, SPEC</td>
</tr>
<tr>
<td>8</td>
<td>Cavazos et al. [48]</td>
<td>204</td>
<td>20</td>
<td>iterative compilation, dynamic characterization, PC, sequence predictor</td>
</tr>
<tr>
<td>9</td>
<td>Almagor et al. [6]</td>
<td>188</td>
<td>15</td>
<td>adaptive compilation, compiler sequence predictor, SPARC architecture</td>
</tr>
<tr>
<td>10</td>
<td>Tiwari et al. [241]</td>
<td>184</td>
<td>23</td>
<td>CHiLL framework, iterative compilation, compiler sequence predictor</td>
</tr>
<tr>
<td>11</td>
<td>Monsfro et al. [176]</td>
<td>166</td>
<td>11</td>
<td>decision trees, loop-unrolling, boosting, abstract loop representation</td>
</tr>
<tr>
<td>12</td>
<td>Stephenson et al. [2005] [229]</td>
<td>162</td>
<td>13</td>
<td>supervised learning, unrolling factor, multiclass classification</td>
</tr>
<tr>
<td>13</td>
<td>Pan et al. [194]</td>
<td>155</td>
<td>13</td>
<td>combined elimination, iterative compilation, SPEC</td>
</tr>
<tr>
<td>14</td>
<td>Bodin et al. [40]</td>
<td>141</td>
<td>7</td>
<td>iterative compilation, multi-objective exploration</td>
</tr>
<tr>
<td>15</td>
<td>Hoste and Eeckhout [117]</td>
<td>122</td>
<td>13</td>
<td>iterative compilation, metaheuristic, genetic algorithm</td>
</tr>
</tbody>
</table>

up to 37% speedup when they used their evaluation metric. Ashouri et al. [28] applied roofline model to a customized VLIW architecture and formed four clusters to satisfy a multi-objective proposed scenario. They observed reaching a speedup of up to 23% in execution time by using those compiler optimizations found in their method that contributed to the satisfaction of the objectives.

**Power.** Except for a few recent work, namely [181, 249], this objective has been mostly investigated at architectural and system-level design [13, 29, 156, 191]. For this reason, it is outside the scope of this survey.

### 8.3 Citation Metric

In this section we classify the top 15 most-cited papers among more than 150 papers we elaborated. We present their citation count and the average citation per year (ACPY) with a few keywords representing their methodology we already covered in this survey. Table 9 depicts the classification.

### 9 DISCUSSION & CONCLUSIONS

In the coming decades, research on compilation techniques and code optimization will play a key role in tackling and addressing various challenges of computer science and the high performance computing field. This includes auto-parallelization, security, exploiting multi and many-core processors, reliability, reproducibility, and energy efficiency. Using compiler optimizations to exploit large-scale parallelism available on architectures and power-aware hardware is an essential task. Additionally, machine learning is becoming more powerful by leveraging deep learning to find and construct heuristics. These complex learners allow automated systems to efficiently perform these task with minimal programmer effort.

Additionally, research on collaborating tuning methodologies have gained attention by the introduction of Collective Knowledge framework (CK) [99–101, 172]. CK is a cross-platform open research SDK developed in collaboration with academic and industrial partners to share artifacts as reusable and customizable components with a unified, portable, and customizable experimental work flows. Researchers can automate the tuning process across diverse hardware and environments.

---

Data has been extracted from Google Scholar on January 2018 and they subject to change.
Crowd-sourcing and reproducing experiments across platforms provided by other fellows would also be feasible. It enables sustainable and portable research software while accelerating AI research.

By the advent of Deep Learning, i.e., deep neural networks (DNN) [218], we are witnessing more and more applications of such techniques, i.e., speech recognition [114], image classification [139], etc. Deep learning algorithms extract high-level and complex abstractions from data through a hierarchical learning process and have shown effectiveness in many applications. DNNs normally require large data to train, thus adapting new benchmarks and large data sets will play an important role in achieving the potential benefits of using DNNs. Recently, we have seen benchmark synthesizers which are able to generate synthetic programs for use in machine learning-based performance autotuning. Genesis [60] and CLgen [71] are such examples that bring diversity and control to the generation of new benchmarks to the user.

In this survey, we have synthesized the research work on compiler autotuning using machine learning by showing the broad spectrum of the use of machine learning techniques and their key research ideas and applications. We surveyed research works at different levels of abstraction, application characterization techniques, algorithm and machine learning models, prediction types, space exploration, target domains, and influence. We discussed both major problems of compiler autotuning, namely the selection and the phase-ordering problem along with the benchmark suits proposed to evaluate them. We hope this article will be beneficial to computer architects, researchers, and application developers and inspires novel ideas and open promising research avenues.

REFERENCES


A Survey on Compiler Autotuning using Machine Learning


1:32 A. H. Ashouri et al.


A Survey on Compiler Autotuning using Machine Learning


Texas Instruments. 2012. OMAP4430 SoC dev. board, revision A.


A Survey on Compiler Autotuning using Machine Learning


Received November 2016; revised August 2017; revised February 2018; accepted March 2018