A Survey on Compiler Autotuning using Machine Learning

Accepted in ACM Computing Surveys 2018 (Received Nov 2016, Revised Aug 2017)

https://doi.org/10.1145/3197978

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.

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

1 INTRODUCTION

Moore's Law [220] 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 [115]. Entering the *post Moore's Law* era [81], 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, 115] 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 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 [193].

Autotuning [35, 256] 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 [56, 93]. 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 [251]. 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 [274]. 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 [19, 148, 151, 254, 262]. 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 [21, 42, 60]. 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 beneficial for their application. Compiler developers provide standard "named" optimization levels, e.g. `-O1`, `-O2`, `-Os`, etc. to introduce a fixed-sequence

[1] https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html
[2] http://llvm.org/docs/Passes.html

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 proposed by the authors tackling the aforementioned problems. We highlight over 200 recent papers proposed for compiler autotuning when Machine Learning (ML) was used. The classification in different subfields is done by several representative features shown in Figure 1. To the best of our knowledge, the first application of machine learning adapted for autotuning compilers were proposed by [69, 143, 180]. However, there were other original works which discuss the problems and acted as the driving force of using machine learning [34, 42, 64, 164, 191, 207, 254, 263, 269]. Thus, 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 [33, 223] 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. [51] 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,

[3]We look forward to keep updating the current survey on a quarterly basis at its arXiv branch [20] and we encourage researchers to inform us about their new related works.

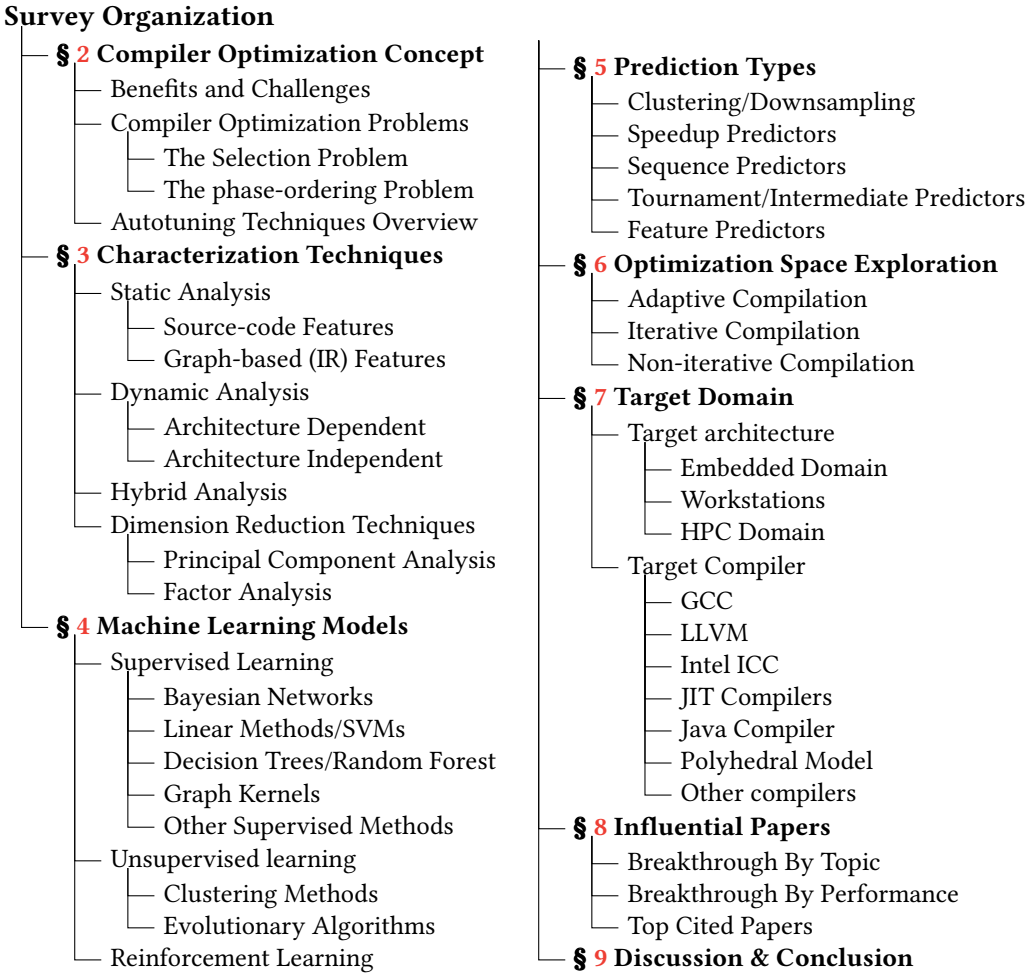

Fig. 1. Organization of the survey in different sections

this work is also cited in other classification tables (prediction type, target platform). We hope 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*

Table 1. A Classification Based on the type of the Problem

Classes	References
The Selection Problem	[3, 6, 11, 15–17, 21, 22, 24, 27–29, 39, 42, 50–55, 61, 62, 67, 69, 70, 72, 78, 79, 83, 91, 92, 96, 98–101, 105–108, 116, 122, 123, 132, 135, 136, 142, 143, 152, 155, 157, 160, 162, 162, 163, 166, 169, 170, 178, 183, 195–198, 200–202, 206, 208–210, 217–219, 221, 235–238, 240, 243, 244, 248, 250, 253, 259, 267, 271, 273]
The Phase-ordering Problem	[17–19, 25, 26, 28, 68, 111, 127, 146–149, 151, 153, 171, 184–187, 199, 212, 213, 251, 254, 262, 263]

autotuning. Moreover, under each classified subsection, we will point out the other nomenclatures that have been used widely and our reference subsequently.

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 [191] 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 [156, 254]. 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" [239]. 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, 21, 42].

Optimizations Space. Let $\boldsymbol{o}$ 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 $\boldsymbol{o}$. 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 [15]. 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_i, 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:

$$|\Omega_{Selection\_Extended}| = \{0, 1, ..., m\}^n \tag{2}$$

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 hinders 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 [156]. 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" [156, 254].

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

$$|\Omega_{Phases}| = n! \tag{3}$$

where n shows the number of optimizations under analysis [15, 18]. 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:

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

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 [18, 156].

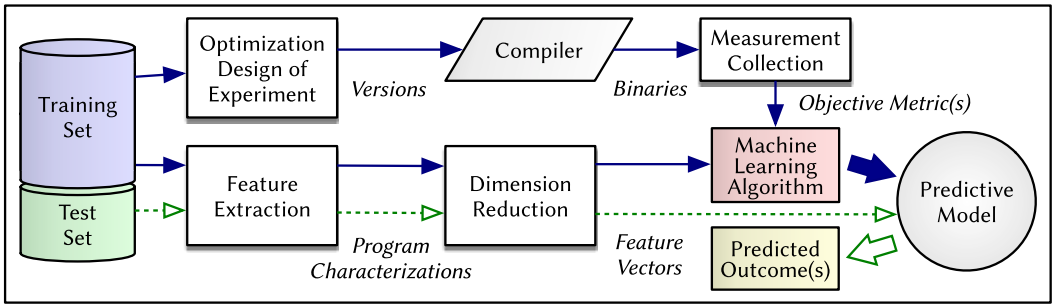

Fig. 2. A Sample Autotuning Framework Using Machine Learning (1) Top: The data flow through the various components of the training phase, where a model is induced based on the characteristics of applications under training set, and, (2) Bottom: The data flow through the various components of the test phase, where the already trained model is used to predict an outcome for a given test application with applications under the test set.

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 [266]. 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) [215] 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 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 [17, 22, 23].

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

Table 2. A Classification Based on Application Characterization Techniques

Classification		References
Static	Source-code Features	[3, 21, 28, 50, 52, 55, 66, 72, 78, 92, 96, 101, 105, 107, 108, 112, 115, 151, 152, 155, 157, 160, 162, 166, 169, 170, 178, 183, 196, 198, 200, 202, 213, 217, 235, 236, 250, 270, 271]
	Graph-based (IR) Features	[52, 53, 67, 72, 78, 92, 96, 143, 155, 157, 166, 169, 183, 186, 196, 200, 202, 235, 237, 240, 250, 262]
Dynamic	Architecture Dependent	[51, 61, 77, 79, 96, 98, 100, 136, 160, 169, 195, 196, 199–202, 217, 235, 237, 250, 253, 259]
	Architecture Independent	[15, 18, 19, 21, 28, 61, 78, 199]
Hybrid features		[21, 28, 77, 155, 157, 169, 196, 200, 202, 217, 250]

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 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 [101, 105], as a plugin to GCC compiler to extract source-level features [4].

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" [86, 190]. 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 [86]. There are numerous tools to extract a control flow graph of a kernel, a function, or an application. MinIR [176], LLVM's Opt , IDA pro [80] are such examples of these tools available.

[4]http://ctuning.org/wiki/index.php/CTools:MilepostGCC:StaticFeatures:MILEPOST_V2.1

Koseki et. al. [143] 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. [198] 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 [46]. 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.

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 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 [167]. For example, in the work of Cavazos et. al [51], 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" [51]. 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 [83, 169, 201], such as PAPI [182]. 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. [251] used PFMon [128], another notable performance monitoring tool when executing programs for the Intel Itanium platform.

Cavazos et. al. [51] 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.

Table 3. A Classification Based on Dimension Reduction Techniques

Classification	References
Principal Component Analysis (PCA)	[3, 15, 18, 19, 21, 28, 29, 61, 78, 142, 198, 199, 202, 248]
Factor Analysis (FA)	[21]

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 [51]. 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* [165] 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. Microarchitecture-independent workload Characterization of Applications (MICA) [121] 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 [21, 200], hybrid feature selection can more accurately capture application behaviors since different levels of feature extraction are considered.

Park et. al. [200] used HERCULES, a pattern-driven tool to characterize an application [133, 134]. 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) it eliminates the noise that may perturb further analyses, (ii) reduces 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) [130], and Exploratory Factor Analysis (EFA) [113]. Reducing the input feature space for a machine learning algorithm may yield better learning results; however, the type of feature reduction used can be an important factor itself [3, 21]. For instance, PCA was used in the original work proposed by Ashouri et al. [15], but later the authors revised the model by exploiting EFA and observed clear benefits [21]. We elaborate more on these methods in latter sections of the survey. Table 3 classifies the use of each technique in the studied literature.

Let $\boldsymbol{\gamma}$ be a characterization vector having all feature data of an application's execution. This vector represents l variables to account for either the static, dynamic or a hybrid analyses. Let us consider a set of feature vectors representing the application characterization profiles A consisting of m vectors $\boldsymbol{\gamma}$. The feature vectors can be organized in a matrix P with m rows and l columns. Each vector $\boldsymbol{\gamma}$ (a row in P) includes a large set of characteristics obtained through analysis. Many of these application characteristics (columns of matrix P) are correlated to each other in complex ways, while other characteristics may have no apparent correlation. "Both PCA and FA are statistical techniques aimed at identifying a way to represent $\boldsymbol{\gamma}$ with a shorter vector $\boldsymbol{\alpha}$ while minimizing the information loss. Nevertheless, they rely on different concepts for organizing this reduction" [21, 113, 247]. In both cases, output values are derived by applying the dimension reduction and no longer directly representing a certain feature. Contrary to PCA, where the components are given by a combination of the observed features, EFA has factors represent a hidden process behind the feature generation.

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

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

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 [177]. 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 to 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" [199]; 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 [199]. There are other prediction types which we discuss and classify in details in Section 5.

Table 4. A Classification Based on Machine Learning Models

Classification		References
Supervised Learning	Bayesian Net	[15, 21]
	Linear Models / SVMs	[18, 19, 21, 71, 199, 201, 217, 237]
	Decision Trees / Random Forests	[41, 77, 92, 99, 99, 101, 105, 152, 162, 166, 178, 200, 253]
	Graph Kernels	[138, 170, 171, 198]
	Others	[3, 11, 16, 19, 21, 21, 28, 28, 50–52, 52, 69, 70, 72, 78, 79, 83, 91, 92, 96, 96, 98, 100, 101, 105, 107, 108, 116, 132, 136, 151, 152, 155, 162, 163, 183, 183, 197, 199, 201, 202, 206, 217, 221, 235, 235, 236, 236, 240, 248, 250, 253, 253, 259, 259]
Unsupervised	Clustering	[19, 29, 170, 171, 212, 248]
	Evolutionary Algorithms (GAs,NEAT,NN)	[3, 6, 50, 54, 66, 67, 69, 70, 73, 78, 82, 110, 110, 122, 123, 142, 146–148, 151, 151, 152, 152, 153, 155, 157, 162, 171, 184, 186, 198, 202, 205, 209, 212, 235, 235, 236, 238, 244, 259]
Reinforcement Learning		[66, 172, 173]

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 [76, 177]. 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) [95, 204] 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" [15, 21]. 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. [15, 21] proposed a Bayesian Network approach to infer the best compiler optimizations suitable for an embedded processor. The approach uses static, dynamic, and hybrid features of an application as evidence to the trained networks. Rather using all training data, 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 and subsequently ables to populates the optimal solutions for an unseen application. The evaluations were carried out with Cbench [97] and Polybench [114, 211] suites and the authors showed that employing BN with an iterative compilation outperforms 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" [76]. 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" [201]. In SVMs, there are algorithms that can adapt linear methods to a non-linear class of problems using a technique called kernel

trick [224]. 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 [137, 199]).

Sanchez et al. [217] 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 [177]. 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" [76, 94].

Fraser et al. [92] 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 [178] 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.

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" [143, 186, 198]. Graph kernels are emerging as a means of exploiting many different machine learning applications on a wide range of applications from semi-supervised learning [57] to clustering and classification [48].

Park et. al. [198] 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 [46]. 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 [177], etc..

Moss et al. [180] 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 [52] 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.

Haneda et al. [116] 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. [250] 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.

Namolaru et al. [183] 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 [117, 177]. Unsupervised learning is closely tightened with the problem of density estimation in statistics [231]; 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. [248] 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 [32], 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. [16, 29] 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. [170, 171] tackled the problem of phase-ordering by a clustering-based selection method for grouping functions with similarities. 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. Finally, they applied the optimization passes already included in the optimization space to measure the exploration speedup versus the 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 [177]. 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) [75], which is a popular method for many multi-objective optimization problems and have had numerous applications mostly in the computer architecture domain [168, 229]. NSGA-II is shown to alleviate the computational complexity of classic GA algorithms.

Another interesting evolutionary model is Neuro Evolution of Augmenting Topologies (NEAT) [234]. 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 [66, 151, 152]. This section summarized a few notable research work that used evolutionary algorithms.

Cooper et al. [69, 70] 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.[142] proposed an iterative compilation approach to tackle the selection size of the tiling and the unrolling factors in an architecture independent manner. The authors evaluated their approach using a number of state-of-the-art iterative compilation techniques, e.g., simulated annealing and GAs, and a native Fortran77 or g77 compiler enabling optimizations for Matrix-Matrix Multiplication (M×M), Matrix-Vector Multiplication (M×V), and Forward Discrete Cosine Transform.

Agakov et al. [3] adapted a number of models to speed up the exploration of an iterative compilation space. The methodology exploits a Markov chain oracle and an independent identically distributed (IID) probability distribution oracle. The authors gained significant speedup by guiding their iterative compilation using these two models when tested on unseen applications. When predicting the best optimizations given an unseen application, they use a nearest neighbor classifier. First, it identifies 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. [155] 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 [152] and phase-ordering [151]. 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. [212] defined a machine learning based approach to downsample a set of compiler optimization sequences within LLVM's `-O2` and applied machine learning to train a model. The authors introduced a clustering algorithm to cluster sequences based on the 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. Later, Ashouri et al. [19] used such similarity matrix for LLVM's `-O3` passes to speed up the exploration, or recommendation, of predicted sequences.

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 [177]. RL uses Markov decision process (MDP) [124] to adapt and interact with its environments. In this section, we have provided the works done with RL in the field.

McGovern et al. [172] presented two methods of building instruction scheduler using rollouts, an improved Monte Carlo search [246], and a 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. [66] 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

The major classes of prediction type are: (i) to predict the right set of compiler optimizations to be used, (ii) to predict the speedup of a compiler optimization sequence without actually executing the sequence, (iii) to predict the right set of features to be used in order to characterize the application, (iv) the intermediate speedup (tournament) prediction, 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 a 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 [177]. In order to approach reasonable solutions to the compiler optimization problems, the optimization search space needs to be reduced. Researchers

Table 5. A Classification Based on Prediction Types

Classification	References
Clustering / Downsampling	[19, 29, 111, 122, 142, 170, 171, 185, 186, 212, 248]
Speedup Prediction	[18, 19, 21, 28, 50, 67, 77, 78, 96, 99, 100, 136, 152, 155, 199–202, 235, 236, 259]
Compiler Sequence Prediction	[3, 6, 14–16, 21, 28, 29, 38, 41, 42, 50–55, 61, 67, 69, 70, 79, 83, 91, 98, 99, 101, 105–108, 116, 122, 123, 132, 142, 146–149, 151, 160, 162, 163, 166, 169–171, 178, 183–185, 196, 198, 201, 202, 206, 208, 209, 212, 213, 217, 221, 235–237, 240, 248, 250, 253, 259]
Tournament / Intermediate	[18, 151, 199, 201, 202, 251]
Feature Prediction	[66, 73, 77, 157, 202, 253]

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 mention these papers in the table 5.

Ashouri et al. [19] 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 [198, 199]. 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" [199]. 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. [78, 79] proposed a speedup predictor based on the source code features of the optimized applications. The authors used static features from SUIF compiler infrastructure [266] 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. [198–202] proposed several predictive modeling methodologies to tackle the problem of selecting the right set of compiler optimizations. In [198], 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 [46]

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. Later in [199], instead of using hardware-dependent features, authors used static features from the source-code on a polyhedral space and predict the performance of each high-level complex optimization sequence with trained models. In a later work [200], they used user-defined patterns as program features, named HERCULES [133], 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 $\bar{o} \in O$ 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. [15, 21, 201]. 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.

Cavazos et al. [51] 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 [15, 21] 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 [201], 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 [50]) [18, 151] 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 [18, 151].

Park et al. [201] 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 [182]: 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 [151] 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. [18] 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 [151]. 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.

Vaswani et al. [253] used empirical data to construct a model that is sensitive to microarchitecture parameters and compiler optimization sequences. The proposed model automatically learns the relationship 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. [157] used a machine learning based optimization focused on feature processing. The authors adapted this method based on an 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 a model constructor 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 iteratively extract the required features at runtime to be fed to the optimization model.

Ding et al. [77] proposed an autotuning framework capable of incorporating different inputs 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 [29]. 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 [145] . In general, different applications could impose different energy and performance requirements. The main goal of this phase is to efficiently traverse and

Table 6. A Classification Based on Space Exploration Methods

Classification	References
Adaptive Compilation	[6, 10, 11, 41, 51, 55, 62, 67, 69, 70, 77, 79, 83, 91, 96, 98, 99, 101, 105, 107, 108, 116, 122, 123, 146, 151, 155, 162, 163, 166, 169, 183, 189, 196, 197, 206, 217, 219, 235–237, 240, 253, 260, 267]
Iterative Compilation	[1, 3, 6, 10, 11, 15, 16, 16, 18, 19, 21, 28, 29, 38, 39, 42, 50, 51, 53–55, 58, 61, 67, 69, 70, 77–79, 83, 91, 96, 98–101, 105–108, 116, 122, 123, 135, 136, 142, 143, 146–153, 155, 160, 162, 163, 163, 166, 169–171, 183–186, 189, 196–202, 206, 208–210, 217, 219, 221, 235–237, 243, 245, 248, 250, 253, 259, 267, 270, 271]
Non-iterative	[92, 167, 209, 210, 213, 235, 254, 262, 263]

configure the exploration parameters [192, 194]. 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 [260], 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. As suggested by [67], 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. [67, 70] 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 [18, 151, 201].

Fursin and Cohen [99] 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 [42, 143], (ii) combined with machine learning techniques [3, 21, 51], or (iii) combined with other search and meta-heuristics techniques such as random exploration [42] and DSE techniques [170, 185].

Bodin et al. [42] 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 unrolling, tiling, and padding parameters. Later, Other researchers, inspired by the iterative approach, explored other optimization parameters to scalar, loop-nest, and other optimizations [16, 29, 83, 106, 139–141, 201, 209].

Triantafyllis et al. [251] 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 an iterative method for 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 [18, 151].

6.3 Non-iterative Compilation

Unlike iterative compilation, a non-iterative approach tries to presents a global optimization approach for a class of compiler optimization problem. Fewer recent works are observed tackling compiler optimization problems using this method [156]. 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 using 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. [254] 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. [263] proposed a framework that investigates the use of Gospel specifications [261] 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.

Table 7. A Classification Based on Target Platform

Platform	References
Embedded Domain	[1, 3, 11, 15, 16, 21, 28, 29, 39, 41, 49, 50, 70, 78, 91, 92, 96, 101, 105, 135, 143, 148, 170, 171, 183–186, 195, 197, 202, 273]
Desktop	[3, 6, 10, 11, 18, 19, 28, 44, 51–55, 58, 61, 67, 69, 70, 79, 91, 96, 98–101, 105–108, 116, 122, 123, 135, 136, 142, 146, 151, 152, 155, 157, 162, 162, 163, 166, 169, 178, 183, 186, 187, 189, 196–202, 206, 208–210, 212, 213, 217, 219, 221, 230, 235–238, 240, 248, 250, 253, 259, 270, 271]
HPC Domain	[10, 11, 19, 36, 77, 83, 98, 101, 105, 109, 153, 159, 160, 175, 184, 198, 208, 209, 217, 221, 225–228, 240, 243, 249, 250, 270, 271]

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 [203]. 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 [90]. Some of the more notable embedded architectures are Very Long Instruction Word (VLIW) [87, 88] and the big.LITTLE heterogeneous architecture from ARM [129]. There are many recent low-cost implementation boards with different system-on-a-chip (SoC) specifications such as Raspberry Pi [252] and Texas Instrument's Pandaboard [125]. Readers can refer to the already available surveys in the field of embedded computing and FPGAs [12, 65, 118]. 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 [84, 163]. 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, 192, 194]. Compiler optimization techniques can be exploited for this task as well [29, 87, 184].

Namolaru et al. [183] 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. [248] 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.

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 [47, 243]. 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. [249] proposed a scalable autotuning framework that incorporated Active Harmony's parallel search backend [242] on the CHiLL compiler framework [59]. 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. [77] presented an autotuning framework capable of leveraging different input in 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. [83] 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., `-Ox` 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 parallelism (ILP) and super-scalar architectures [257], 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" [87, 89]. 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

Table 8. A Classification Based on Target Compiler

Compilers	References
GCC	[3, 11, 15, 21, 28, 39, 40, 50, 61, 79, 83, 91, 92, 99, 101, 105, 107, 108, 112, 116, 122, 132, 135, 153, 155, 157, 160, 166, 169, 183, 189, 195–198, 200, 202, 205, 206, 208–210, 221, 230, 235, 240, 248, 253, 259, 270, 271]
LLVM	[14, 16, 18, 19, 21, 28, 29, 41, 72, 111, 112, 170, 171, 184–187, 199, 202, 212]
Intel-ICC	[44, 61, 91, 112, 135, 136, 160, 198, 200, 202, 209, 221, 240, 270]
JIT Compiler	[52–55, 123, 126, 151, 152, 202, 213, 217, 221, 235]
Java Compiler	[52–55, 123, 126, 151, 213]
Polyhedral Model	[43, 44, 199, 202, 208–210, 249, 258, 270, 271]
Others	[3, 6, 10, 11, 16, 28, 36, 51, 58, 67, 69, 70, 78, 91, 96, 98–101, 105, 106, 142, 146, 151, 152, 162, 162, 163, 170, 171, 178, 201, 202, 213, 216–219, 235–238, 249, 250, 259]

Table 9. Default Compiler Passes Inside GCC's -O3

Compiler Passes
-fauto-inc-dec -fbranch-count-reg -fcombine-stack-adjustments -fcompare-elim -fcprop-registers -fdce -fdefer-pop -fdelayed-branch -fdse -fforward-propagate -fguess-branch-probability -fif-conversion2 -fif-conversion -finline-functions-called-once -fipa-pure-const -fipa-profile -fipa-reference -fmerge-constants -fmove-loop-invariants -freorder-blocks -fshrink-wrap -fsplit-wide-types -fssa-backprop -fssa-phiopt -ftree-bit-ccp -ftree-ccp -ftree-ch -ftree-coalesce-vars -ftree-copy-prop -ftree-dce -ftree-dominator-opts -ftree-dse -ftree-forwprop -ftree-fre -ftree-phiprop -ftree-sink -ftree-slsr -ftree-sra -ftree-pta -ftree-ter -funit-at-a-time -fthread-jumps -falign-functions -falign-jumps -falign-loops -falign-labels -fcaller-saves -fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fdelete-null-pointer-checks -fdevirtualize -fdevirtualize-speculatively -fexpensive-optimizations -fgcse -fgcse-lm -fhoist-adjacent-loads -finline-small-functions -findirect-inlining -fipa-cp -fipa-cp-alignment -fipa-bit-cp -fipa-sra -fipa-icf -fisolate-erroneous-paths-dereference -flra-remat -foptimize-sibling-calls -foptimize-strlen -fpartial-inlining -fpeephole2 -freorder-blocks-algorithm=stc -freorder-blocks-and-partition -freorder-functions -frerun-cse-after-loop -fsched-interblock -fsched-spec -fschedule-insns -fschedule-insns2 -fstrict-aliasing -fstrict-overflow -ftree-builtin-call-dce -ftree-switch-conversion -ftree-tail-merge -fcode-hoisting -ftree-pre -ftree-vrp -fipa-ra -finline-functions -funswitch-loops -fpredictive-commoning -fgcse-after-reload -ftree-loop-vectorize -ftree-loop-distribute-patterns -fsplit-paths -ftree-slp-vectorize -fvect-cost-model -ftree-partial-pre -fpeel-loops -fipa-cp-clone

role in the growth of free software, as both a tool and an example" [232, 233]. 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 [5] supports different predefined levels of fixed optimization levels such as standard levels `-Ofast`, `-O1`, `-O2` and `-O3`. Refer to the Table 9 for the list of optimization passes inside GCC's `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 [154] 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

[5]https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Table 10. Default Compiler Passes Inside LLVM's -O3

Compiler Passes
-tti -targetlibinfo -tbaa -scoped-noalias -assumption-cache-tracker -forceattrs -inferattrs -ipsccp -globalopt -domtree -mem2reg -deadargelim -basicaa -aa -domtree -instcombine -simplifycfg -basiccg -globals-aa -prune-eh -inline -functionattrs -argpromotion -domtree -sroa -early-cse -lazy-value-info -jump-threading -correlated-propagation -simplifycfg -basicaa -aa -domtree -instcombine -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa -loop-rotate -basicaa -aa -licm -loop-unswitch -simplifycfg -basicaa -aa -domtree -instcombine -loops -scalar-evolution -loop-simplify -lcssa -indvars -aa -loop-idiom -loop-deletion -loop-unroll -basicaa -aa -mldst-motion -aa -memdep -gvn -basicaa -aa -memdep -memcpyopt -sccp -domtree -demanded-bits -bdce -basicaa -aa -instcombine -lazy-value-info -jump-threading -correlated-propagation -domtree -basicaa -aa -memdep -dse -loops -loop-simplify -lcssa -aa -licm -adce -simplifycfg -basicaa -aa -domtree -instcombine -barrier -basiccg -rpo-functionattrs -elim-avail-extern -basiccg -globals-aa -float2int -domtree -loops -loop-simplify -lcssa -loop-rotate -branch-prob -block-freq -scalar-evolution -basicaa -aa -loop-accesses -demanded-bits -loop-vectorize -instcombine -scalar-evolution -aa -slp-vectorizer -simplifycfg -basicaa -aa -domtree -instcombine -loops -loop-simplify -lcssa -scalar-evolution -loop-unroll -basicaa -aa -instcombine -loop-simplify -lcssa -aa -licm -scalar-evolution -alignment-from-assumptions -strip-dead-prototypes -globaldce -constmerge

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].

Table 10 represents the optimization passes inside LLVM's `O3`.

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.

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" [31]. 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 [188] a Java JIT compiler, and IBM's JIT compiler targets the Java virtual machine [241].

Sanchez et al. [217] 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

[6] http://llvm.org/pubs/
[7] https://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, 131]. Some of the notable research works [52, 123] including a work tackling the phase-ordering problem have been done using Java JIT compiler [151]. 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 [8]. "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" [85, 264]. 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 [255]. Polyhedral compilation can address many of these challenges by efficiently transforming their loop-nest [37, 44, 45, 161, 209]. 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 Cosy [9] and SUIF [266]. We classified all work related to use of other compiler in Table 8 under others subfield.

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" [266]. It has been used in a number of recent research works [58, 62, 78].

Fortran compilers include many different implementations and different ports such as Open64 [**?**], GNU Fortran [**?**], XL Fortran [**?**], Salford Fortran 77 compiler [**?**], etc. and are widely used in literature [6, 142, 218, 219].

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. [156, 254] were the first to perform non-iterative optimization without leverage machine learning. [262] extended this work by proposing intelligent ordering of a subset of compiler optimizations. [263] continued their phase-ordering work with a formal language, Gospel, which could be used to automatically generate

[8] http://polyhedral.info/

transformations. The first usage of machine learning techniques arrived with [143] and their work with predicting the optimal unroll size for nested loops. [180] was the first to use machine learning techniques to construct flexible instruction schedules, paving the way for continued efforts leveraging ML techniques.

Genetic Algorithms. Cooper et al. [69] expanded machine learning efforts with optimization selection using genetic algorithms with iterative compilation. [70] 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 [251] where they applied iterative compilation of the SPEC benchmarks. [142] 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. [146] was one of the first to propose solving the phase-ordering problem using machine learning by combining iterative compilation and meta-heuristics. [151] 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. [42] 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. [51]; 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 [201].

Practical and Collaborative Autotuning. MILEPOST GCC [101, 105] was the first attempt to make a practical on-the-fly machine-learning based compiler combined with an infrastructure targeted to autotuning and crowdsourcing scenarios. It has been used in practice and revealed many issues yet to be tackled by researchers including (1) reproducibility of empirical results collected with multiple users and (2) problems with metadata, data representation, models, and massive datasets [102, 104].

Hybrid Characterization and Bayesian Learners. Massive dataset analysis on over 1000 benchmarks was performed by [60, 61]. 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. [15, 21] used the output of the passes suggested by [60] 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. [19] 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 [214] 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 between the proposed approaches regarding their reported results. Evaluating papers using their performance is a hard task and often involves comparison errors [120], 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 [42]. However, the approach is expensive and should be combined with an intelligent search algorithms, such as machine learning techniques [3, 19, 21, 42]. Early work in iterative compilation methods [1, 42] involved the exploration of non-linear transformation spaces and finding 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 the deployment of 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 [142]. Mpeis et al. [181] 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 works 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. [51] 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 [97, 105, 122]. Park et al. [201] 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 [199] (see Section 5.4) as a means of improving existing prediction models. Ashouri et al. introduced COBAYN [21], which used hybrid features and Bayesian network learners. The authors showed that this approach can outperform existing methods [3, 202] by around 11%.

The Phase-ordering Problem. Cooper et al. [70] 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. [146] 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 [151]. 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 [19], 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 [18, 151] 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. [69] 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, 29, 90, 268]. 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 [265] 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, 16, 29, 75, 122, 194, 272]. Hoste and Eeckhout [122] 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 up to 37% speedup when they used their evaluation metric. Ashouri et al. [29] 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 works [184, 258], this objective has been mostly investigated at architectural and system-level [13, 30, 158, 194]. For this reasons, it is outside the scope of this survey.

8.3 Citation Metric

In this section, we show the top 16 most-cited papers among more than 200 papers we elaborated [9]. in Table 11, 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.

9 DISCUSSION & CONCLUSION

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) [101–103, 174]. CK is a cross-platform

[9]Data has been extracted from Google Scholar on July 2018 and they subject to change.

Table 11. A Classification of Top 16 Influential Papers By Their Citation Count

No	Reference	Citation Count	ACPY	Keywords
1	Agakov et al. [2006] [3]	377	33	iterative compilation, static features, Markov chain oracle, PCA
2	Cooper et al. [1999] [69]	322	17	genetic algorithm, iterative compilation, reduced code-size
3	Triantafyllis et al. [2003] [251]	277	19	iterative compilation, SPEC, predictive modeling
4	Stephenson et al. [2003] [236]	274	19	iterative compilation, genetic programming, metaheuristics
5	Fursin et al. [2008,2011][101, 105]	251	22	MilePost GCC, self-tuning, crowdsourcing, Iterative compilation
6	Cooper et al. [2002] [70]	242	15	adaptive compilation, biased random search, sequence predictor
7	Knijnenburg et al. [2003] [142]	238	16	iterative compilation, unrolling factor, architecture-independent
8	Tournavitis et al. [2009] [250]	234	27	static-analysis, profile-driven parallelism, NAS, SPEC
9	Cavazos et al. [2007] [51]	225	20	iterative compilation, dynamic characterization, sequence predictor
10	Tiwari et al. [2009] [249]	200	25	CHiLL framework, iterative compilation, sequence predictor
11	Almagor et al. [2004] [6]	196	14	adaptive compilation, compiler sequence predictor, SPARC
12	Monsifrot et al. [2002] [178]	175	11	decision trees, boosting, abstract loop representation
13	Stephenson et al. [2005] [237]	170	13	supervised learning, unrolling factor, multiclass classification
14	Pan et al. [2006] [197]	161	13	combined elimination, iterative compilation, SPEC
15	Bodin et al. [1998] [42]	151	7	iterative compilation, multi-objective exploration
16	Hoste et al. [2008] [122]	134	13	iterative compilation, metaheuristic, genetic algorithm

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. Crowdsourcing and reproducing experiments across platforms provided by other researchers would also be feasible. CK is now used as an official platform to support open ACM ReQuEST tournaments on reproducible and Pareto-efficient software/hardware co-design of artificial intelligence, deep learning and other emerging workloads [179].

By the advent of *Deep Learning*, i.e., deep neural networks (DNN) [222], we are witnessing more and more applications of such techniques, i.e., speech recognition [119], image classification [144], 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 [63] and CLgen [74] 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 opens promising research avenues.

REFERENCES

[1] Bas Aarts, Michel Barreteau, François Bodin, Peter Brinkhaus, Zbigniew Chamski, Henri-Pierre Charles, Christine Eisenbeis, John Gurd, Jan Hoogerbrugge, Ping Hu, and others. 1997. OCEANS: Optimizing compilers for embedded

applications. *Euro-Par'97 Parallel Processing* (1997), 1351–1356.

[2] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M Parikh, and James M Stichnoth. 1998. Fast, effective code generation in a just-in-time Java compiler. In *ACM SIGPlAN Notices*, Vol. 33. ACM, 280–290.

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

[4] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. *Compilers, Principles, Techniques*. Addison wesley.

[5] Frances E Allen. 1970. Control flow analysis. In *ACM Sigplan Notices*, Vol. 5. ACM, 1–19.

[6] L Almagor and KD Cooper. 2004. Finding Effective Compilation Sequences. *ACM SIGPLAN Notices* 39, 7 (2004), 231–239. http://www.anc.ed.ac.uk/machine-learning/colo/repository/LCTES04.pdf

[7] George Almasi and David A Padua. 2000. MaJIC: A MATLAB just-in-time compiler. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 68–81.

[8] Ethem Alpaydin. 2014. *Introduction to machine learning*. MIT press.

[9] Martin Alt, Uwe Aßmann, and Hans Van Someren. 1994. Cosy compiler phase embedding with the cosy compiler model. In *International Conference on Compiler Construction*. Springer, 278–293.

[10] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 38–49. DOI: http://dx.doi.org/10.1145/1542476.1542481

[11] J Ansel and S Kamil. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 303–316.

[12] Karl-Erik Årzén and Anton Cervin. 2005. Control and embedded computing: Survey of research directions. *IFAC Proceedings Volumes* 38, 1 (2005), 191–202.

[13] G. Ascia, V. Catania, M. Palesi, and D. Patti. Jan. A system-level framework for evaluating area/performance/power trade-offs of VLIW-based embedded systems. In *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, Vol. 2. 940–943 Vol. 2.

[14] Yosi Ben Asher, Gadi Haber, and Esti Stein. 2017. A Study of Conflicting Pairs of Compiler Optimizations. In *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2017 IEEE 11th International Symposium on*. IEEE, 52–58.

[15] A.H. Ashouri, G. Mariani, G. Palermo, and C. Silvano. 2014. A Bayesian network approach for compiler auto-tuning for embedded processors. (2014), 90–97. DOI: http://dx.doi.org/10.1109/ESTIMedia.2014.6962349

[16] Amir Hossein Ashouri. 2012. *Design space exploration methodology for compiler parameters in VLIW processors*. Master's thesis. M. Sc. Dissertation. Politecnico Di Milano, ITALY. http://hdl.handle.net/10589/72083.

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

[18] Amir Hossein Ashouri, Andrea Bignoli, Gianluca Palermo, and Cristina Silvano. 2016. Predictive Modeling Methodology for Compiler Phase-ordering. In *Proceedings of the 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 5th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms (PARMA-DITAM '16)*. ACM, New York, NY, USA, 7–12. DOI: http://dx.doi.org/10.1145/2872421.2872424

[19] Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 2017. MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning. *ACM Trans. Archit. Code Optim.* 14, 3, Article 29 (Sept. 2017), 28 pages. DOI: http://dx.doi.org/10.1145/3124452

[20] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A survey on compiler autotuning using machine learning. *arXiv preprint arXiv:1801.04405* (2018).

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

[22] Amir H Ashouri, Gianluca Palermo, John Cavazos, and Cristina Silvano. 2018a. *Automatic Tuning of Compilers Using Machine Learning*. Springer International Publishing. DOI: http://dx.doi.org/10.1007/978-3-319-71489-9

[23] Amir H. Ashouri, Gianluca Palermo, John Cavazos, and Cristina Silvano. 2018b. *Background*. Springer International Publishing, Cham, 1–22. DOI: http://dx.doi.org/10.1007/978-3-319-71489-9_1

[24] Amir H. Ashouri, Gianluca Palermo, John Cavazos, and Cristina Silvano. 2018c. *Design Space Exploration of Compiler Passes: A Co-Exploration Approach for the Embedded Domain*. Springer International Publishing, Cham, 23–39. DOI: http://dx.doi.org/10.1007/978-3-319-71489-9_2

[25] Amir H. Ashouri, Gianluca Palermo, John Cavazos, and Cristina Silvano. 2018d. *The Phase-Ordering Problem: A Complete Sequence Prediction Approach*. Springer International Publishing, Cham, 85–113. DOI: http://dx.doi.org/10.

1007/978-3-319-71489-9_5

[26] Amir H. Ashouri, Gianluca Palermo, John Cavazos, and Cristina Silvano. 2018e. *The Phase-Ordering Problem: An Intermediate Speedup Prediction Approach.* Springer International Publishing, Cham, 71–83. DOI: http://dx.doi.org/10.1007/978-3-319-71489-9_4

[27] Amir H. Ashouri, Gianluca Palermo, John Cavazos, and Cristina Silvano. 2018f. *Selecting the Best Compiler Optimizations: A Bayesian Network Approach.* Springer International Publishing, Cham, 41–70. DOI: http://dx.doi.org/10.1007/978-3-319-71489-9_3

[28] Amir Hossein Ashouri, Gianluca Palermo, and Cristina Silvano. An Evaluation of Autotuning Techniques for the Compiler Optimization Problems. In *RES4ANT2016 co-located with DATE 2016.* 23 –27. http://ceur-ws.org/Vol-1643/#paper-05

[29] Amir Hossein Ashouri, Vittorio Zaccaria, Sotirios Xydis, Gianluca Palermo, and Cristina Silvano. 2013. A framework for Compiler Level statistical analysis over customized VLIW architecture. In *VLSI-SoC*. 124–129. DOI: http://dx.doi.org/10.1109/VLSI-SoC.2013.6673262

[30] Jose L Ayala, Marisa López-Vallejo, David Atienza, Praveen Raghavan, Francky Catthoor, and Diederik Verkest. 2007. Energy-aware compilation and hardware design for VLIW embedded systems. *International Journal of Embedded Systems* 3, 1-2 (2007), 73–82.

[31] John Aycock. 2003. A brief history of just-in-time. *ACM Computing Surveys (CSUR)* 35, 2 (2003), 97–113.

[32] R Babuka, PJ Van der Veen, and U Kaymak. 2002. Improved covariance estimation for Gustafson-Kessel clustering. In *Fuzzy Systems, 2002. FUZZ-IEEE'02. Proceedings of the 2002 IEEE International Conference on*, Vol. 2. IEEE, 1081–1085.

[33] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler transformations for high-performance computing. *Comput. Surveys* 26, 4 (dec 1994), 345–420. DOI: http://dx.doi.org/10.1145/197405.197406

[34] Victor R Basil and Albert J Turner. 1975. Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering* 4 (1975), 390–396.

[35] Protonu Basu, Mary Hall, Malik Khan, Suchit Maindola, Saurav Muralidharan, Shreyas Ramalingam, Axel Rivera, Manu Shantharam, and Anand Venkat. 2013. Towards making autotuning mainstream. *The International Journal of High Performance Computing Applications* 27, 4 (2013), 379–393.

[36] Protonu Basu, Samuel Williams, Brian Van Straalen, Leonid Oliker, Phillip Colella, and Mary Hall. 2017. Compiler-based code generation and autotuning for geometric multigrid on GPU-accelerated supercomputers. *Parallel Comput.* 64 (2017), 50–64.

[37] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The polyhedral model is more widely applicable than you think. In *Compiler Construction*. Springer, 283–303.

[38] Craig Blackmore, Oliver Ray, and Kerstin Eder. 2015. A logic programming approach to predict effective compiler settings for embedded software. *Theory and Practice of Logic Programming* 15, 4-5 (2015), 481–494.

[39] Craig Blackmore, Oliver Ray, and Kerstin Eder. 2017a. Automatically Tuning the GCC Compiler to Optimize the Performance of Applications Running on the ARM Cortex-M3. *arXiv preprint arXiv:1703.08228* (2017).

[40] Craig Blackmore, Oliver Ray, and Kerstin Eder. 2017b. Automatically Tuning the GCC Compiler to Optimize the Performance of Applications Running on the ARM Cortex-M3. *CoRR* abs/1703.08228 (2017). http://arxiv.org/abs/1703.08228

[41] Bruno Bodin, Luigi Nardi, M Zeeshan Zia, Harry Wagstaff, Govind Sreekar Shenoy, Murali Emani, John Mawer, Christos Kotselidis, Andy Nisbet, Mikel Lujan, and others. 2016. Integrating algorithmic parameters into benchmarking and design space exploration in 3D scene understanding. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, 57–69.

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

[43] U Bondhugula and M Baskaran. 2008. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*. 132–146. http://link.springer.com/chapter/10.1007/978-3-540-78791-4

[44] U Bondhugula and A Hartono. 2008. A practical automatic polyhedral parallelizer and locality optimizer. (2008). http://dl.acm.org/citation.cfm?id=1375595

[45] Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. 2008. PLuTo: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer.

[46] Karsten M Borgwardt and Hans-Peter Kriegel. 2005. Shortest-path kernels on graphs. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*. IEEE, 8–pp.

[47] Rajkumar Buyya, Chee Shin Yeo, and Srikumar Venugopal. 2008. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *High Performance Computing and Communications, 2008.*

HPCC'08. 10th IEEE International Conference on. Ieee, 5–13.

[48] Gustavo Camps-Valls, Tatyana V Bandos Marsheva, and Dengyong Zhou. 2007. Semi-supervised graph-based hyperspectral image classification. *IEEE Transactions on Geoscience and Remote Sensing* 45, 10 (2007), 3044–3054.

[49] João Manuel Paiva Cardoso, José Gabriel de Figueiredo Coutinho, and Pedro C Diniz. 2017. *Embedded Computing for High Performance: Efficient Mapping of Computations Using Customization, Code Transformations and Compilation*. Morgan Kaufmann.

[50] J Cavazos, C Dubach, and F Agakov. 2006. Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. 24–34. http://dl.acm.org/citation.cfm?id=1176765

[51] J Cavazos, G Fursin, and F Agakov. 2007. Rapidly selecting good compiler optimizations using performance counters. *International Symposium on Code Generation and Optimization (CGO'07)* (2007). http://ieeexplore.ieee.org/xpls/abs

[52] J Cavazos and JEB Moss. 2004. Inducing heuristics to decide whether to schedule. *ACM SIGPLAN Notices* (2004). http://dl.acm.org/citation.cfm?id=996864

[53] J Cavazos, JEB Moss, and MFP O'Boyle. 2006. Hybrid optimizations: Which optimization algorithm to use? *Compiler Construction* (2006). http://link.springer.com/chapter/10.1007/11688839

[54] J Cavazos and MFP O'Boyle. 2005. Automatic tuning of inlining heuristics. *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference* (2005), 14–14. http://ieeexplore.ieee.org/xpls/abs

[55] J Cavazos and MFP O'boyle. 2006. Method-specific dynamic compilation using logistic regression. *ACM SIGPLAN Notices* (2006). http://dl.acm.org/citation.cfm?id=1167492

[56] Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. 1981. Register allocation via coloring. *Computer languages* 6, 1 (1981), 47–57.

[57] Olivier Chapelle, Bernhard Scholkopf, and Alexander Zien. 2009. Semi-Supervised Learning (Chapelle, O. et al., Eds.; 2006)[Book reviews]. *IEEE Transactions on Neural Networks* 20, 3 (2009), 542–542.

[58] C Chen, J Chame, and M Hall. 2005. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. *International Symposium on Code Generation and Optimization* (2005), 111–122. http://ieeexplore.ieee.org/xpls/abs

[59] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations*. Technical Report. Citeseer.

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

[61] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. 2010. Evaluating Iterative Optimization Across 1000 Datasets. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 448–459. DOI: http://dx.doi.org/10.1145/1806596.1806647

[62] B.R. Childers and M.L. Soffa. 2005. A Model-Based Framework: An Approach for Profit-Driven Optimization. In *International Symposium on Code Generation and Optimization*. IEEE, 317–327. DOI: http://dx.doi.org/10.1109/CGO.2005.2

[63] Alton Chiu, Joseph Garvey, and Tarek S Abdelrahman. 2015. Genesis: A language for generating synthetic training programs for machine learning. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*. ACM, 8.

[64] Cliff Click and Keith D Cooper. 1995. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17, 2 (1995), 181–196.

[65] Katherine Compton and Scott Hauck. 2002. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)* 34, 2 (2002), 171–210.

[66] Katherine E Coons, Behnam Robatmili, Matthew E Taylor, Bertrand A Maher, Doug Burger, and Kathryn S McKinley. 2008. Feature selection and policy optimization for distributed instruction placement using reinforcement learning. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 32–42.

[67] KD Cooper, A Grosul, and TJ Harvey. 2005. ACME: adaptive compilation made efficient. 40, 7 (2005), 69–77. http://dl.acm.org/citation.cfm?id=1065921

[68] K Cooper, Timothy J Harvey, Devika Subramanian, and Linda Torczon. 2002. Compilation order matters. *Technical Report* (2002).

[69] KD Cooper, PJ Schielke, and D Subramanian. 1999. Optimizing for reduced code space using genetic algorithms. *ACM SIGPLAN Notices* (1999). http://dl.acm.org/citation.cfm?id=314414

[70] KD Cooper, D Subramanian, and L Torczon. 2002. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing* (2002). http://link.springer.com/article/10.1023/A:1015729001611

[71] Biagio Cosenza, Juan J. Durillo, Stefano Ermon, and Ben Juurlink. 2017. Stencil Autotuning with Ordinal Regression: Extended Abstract. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems (SCOPES '17)*. ACM, New York, NY, USA, 72–75. DOI: http://dx.doi.org/10.1145/3078659.3078664

[72] Chris Cummins, Pavlos Petoumenos, Michel Steuwer, and Hugh Leather. 2015. Autotuning OpenCL workgroup size for stencil patterns. *arXiv preprint arXiv:1511.02490* (2015).

[73] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. 2017a. End-to-End Deep Learning of Optimization Heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 219–232. DOI: http://dx.doi.org/10.1109/PACT.2017.24

[74] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017b. Synthesizing benchmarks for predictive modeling. In *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*. IEEE, 86–99.

[75] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[76] Thomas G Dietterich. 2000. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*. Springer, 1–15.

[77] Y Ding, J Ansel, and K Veeramachaneni. 2015. Autotuning algorithmic choice for input sensitivity. *ACM SIGPLAN Notices* 50, 6 (2015), 379–390. http://dl.acm.org/citation.cfm?id=2737969

[78] C Dubach, J Cavazos, and B Franke. 2007. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th international conference on Computing frontiers*. 131–142. http://dl.acm.org/citation.cfm?id=1242553

[79] C Dubach, TM Jones, and EV Bonilla. 2009. Portable compiler optimisation across embedded programs and microarchitectures using machine learning. (2009), 78–88. http://dl.acm.org/citation.cfm?id=1669124

[80] Chris Eagle. 2011. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press.

[81] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 365–376.

[82] Thomas L Falch and Anne C Elster. 2015. Machine learning based auto-tuning for enhanced opencl performance portability. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 1231–1240.

[83] S Fang, W Xu, Y Chen, and L Eeckhout. 2015. Practical iterative optimization for the data center. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 2 (2015), 15. http://dl.acm.org/citation.cfm?id=2739048

[84] Paolo Faraboschi, Geoffrey Brown, Joseph A Fisher, Giuseppe Desoli, and Fred Homewood. 2000. Lx: a technology platform for customizable VLIW embedded processing. In *ACM SIGARCH Computer Architecture News*, Vol. 28. ACM, 203–213.

[85] Paul Feautrier. 1988. Parametric Integer Programming. *RAIRO Recherche opérationnelle* 22, 3 (1988), 243–268.

[86] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. DOI: http://dx.doi.org/10.1145/24039.24041

[87] JA Fisher, P Faraboschi, and C Young. 2009. VLIW processors: Once blue sky, now commonplace. *Solid-State Circuits Magazine, IEEE* 1, 2 (2009), 10–17.

[88] Joseph A Fisher. 1981. Microcode Compaction. *IEEE Trans. Comput.* 30, 7 (1981).

[89] Joseph A Fisher, Paolo Faraboschi, and Cliff Young. 2004. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Morgan Kaufmann.

[90] Joseph A Fisher, Paolo Faraboschi, and Cliff Young. 2005. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier.

[91] B Franke, M O'Boyle, J Thomson, and G Fursin. 2005. Probabilistic source-level optimisation of embedded programs. *ACM SIGPLAN Notices* (2005). http://dl.acm.org/citation.cfm?id=1065922

[92] Christopher W. Fraser. 1999. Automatic inference of models for statistical code compression. *ACM SIGPLAN Notices* 34, 5 (may 1999), 242–246. DOI: http://dx.doi.org/10.1145/301631.301672

[93] Stefan M Freudenberger and John C Ruttenberg. 1992. Phase ordering of register allocation and instruction scheduling. In *Code GenerationâĂŤConcepts, Tools, Techniques*. Springer, 146–170.

[94] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. 2001. *The elements of statistical learning*. Vol. 1. Springer series in statistics Springer, Berlin.

[95] Nir Friedman, Dan Geiger, and Moises Goldszmidt. 1997. Bayesian network classifiers. *Machine learning* 29, 2-3 (1997), 131–163.

[96] GG Fursin. 2004. Iterative Compilation and Performance Prediction for Numerical Applications. (2004). https://www.era.lib.ed.ac.uk/handle/1842/565

[97] Grigori Fursin. 2010. Collective benchmark (cbench), a collection of open-source programs with multiple datasets

assembled by the community to enable realistic benchmarking and research on program and architecture optimization. (2010).

[98] G Fursin, J Cavazos, M O'Boyle, and O Temam. 2007. Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization. *International Conference on High-Performance Embedded Architectures and Compilers* (2007), 245–260. http://link.springer.com/chapter/10.1007/978-3-540-69338-3

[99] G Fursin and A Cohen. 2007. Building a practical iterative interactive compiler. *Workshop Proceedings* (2007). https://www.researchgate.net/profile/Chuck

[100] G Fursin, A Cohen, M O'Boyle, and O Temam. 2005. A practical method for quickly evaluating program optimizations. *International Conference on High-Performance Embedded Architectures and Compilers* (2005), 29–46. http://link.springer.com/chapter/10.1007/11587514

[101] G Fursin, Y Kashnikov, and AW Memon. 2011. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming* 39, 3 (2011), 296–327. http://link.springer.com/article/10.1007/s10766-010-0161-2

[102] Grigori Fursin, Anton Lokhmotov, and Ed Plowman. 2016. Collective Knowledge: towards R&D sustainability. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 864–869.

[103] Grigori Fursin, Anton Lokhmotov, Dmitry Savenko, and Eben Upton. 2018. A Collective Knowledge workflow for collaborative research into multi-objective autotuning and machine learning techniques. *arXiv preprint arXiv:1801.08024* (2018).

[104] Grigori Fursin, Abdul Memon, Christophe Guillon, and Anton Lokhmotov. 2015. Collective Mind, Part II: Towards performance-and cost-aware software engineering as a natural science. *arXiv preprint arXiv:1506.06256* (2015).

[105] G Fursin, C Miranda, and O Temam. 2008. MILEPOST GCC: machine learning based research compiler. *GCC Summit* (2008). https://hal.inria.fr/inria-00294704/

[106] GG Fursin, MFP O'Boyle, and PMW Knijnenburg. 2002. Evaluating iterative compilation. *International Workshop on Languages and Compilers for Parallel Computing* (2002), 362–376. http://link.springer.com/chapter/10.1007/11596110

[107] Grigori Fursin and Olivier Temam. 2009. Collective optimization. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 34–49.

[108] G Fursin and O Temam. 2010. Collective optimization: A practical collaborative approach. *ACM Transactions on Architecture and Code Optimization (TACO)* 7, 4 (2010), 20. http://dl.acm.org/citation.cfm?id=1880047

[109] Davide Gadioli, Ricardo Nobre, Pedro Pinto, Emanuele Vitali, Amir H. Ashouri, Gianluca Palermo, Cristina Silvano, and Joao Cardoso. 2018. SOCRATES - A Seamless Online Compiler and System Runtime AutoTuning Framework for Energy-Aware Applications. In *Proceedings of the Design, Automation and Test in Europe Conference & Exhibition, DATE*. 1143–1146. DOI: http://dx.doi.org/10.23919/DATE.2018.8342183

[110] Unai Garciarena and Roberto Santana. 2016. Evolutionary Optimization of Compiler Flag Selection by Learning and Exploiting Flags Interactions. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion (GECCO '16 Companion)*. ACM, New York, NY, USA, 1159–1166. DOI: http://dx.doi.org/10.1145/2908961.2931696

[111] Kyriakos Georgiou, Craig Blackmore, Samuel Xavier-de Souza, and Kerstin Eder. 2018. Less is More: Exploiting the Standard Compiler Optimization Levels for Better Performance and Energy Consumption. *arXiv preprint arXiv:1802.09845* (2018).

[112] Zhangxiaowen Gong, Zhi Chen, Justin Josef Szaday, David C Wong, Zehra Sura, Neftali Watkinson, Saeed Maleki, David Padua, Alexandru Nicolau, Alexander V Veidenbaum, and others. 2018. An Empirical Study of the Effect of Source-level Transformations on Compiler Stability. In *Workshop on Compilers for Parallel Computing (CPC)*.

[113] Richard L Gorsuch. 1988. Exploratory factor analysis. In *Handbook of multivariate experimental psychology*. Springer, 231–258.

[114] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012*. IEEE, 1–10.

[115] M Hall, D Padua, and K Pingali. 2009. Compiler research: the next 50 years. *Commun. ACM* (2009). http://dl.acm.org/citation.cfm?id=1461946

[116] M Haneda. 2005. Optimizing general purpose compiler optimization. *Proceedings of the 2nd conference on Computing frontiers* (2005), 180–188. http://dl.acm.org/citation.cfm?id=1062293

[117] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. Unsupervised learning. In *The elements of statistical learning*. Springer, 485–585.

[118] Jeffrey Hightower and Gaetano Borriello. 2001. A survey and taxonomy of location systems for ubiquitous computing. *IEEE computer* 34, 8 (2001), 57–66.

[119] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, and others. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine* 29, 6 (2012), 82–97.

[120] Torsten Hoefler and Roberto Belli. 2015. Scientific benchmarking of parallel computing systems: twelve ways to tell

the masses when reporting performance results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 73.

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

[122] K Hoste and L Eeckhout. 2008. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. 165–174. http://dl.acm.org/citation.cfm?id=1356080

[123] K Hoste, A Georges, and L Eeckhout. 2010. Automated just-in-time compiler tuning. *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* (2010), 62–72. http://dl.acm.org/citation.cfm?id=1772965

[124] Ronald A Howard. 1960. DYNAMIC PROGRAMMING AND MARKOV PROCESSES.. (1960).

[125] Texas Instruments. 2012. Pandaboard. *OMAP4430 SoC dev. board, revision A* 2 (2012), 2012.

[126] Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. 2015. Compiling and optimizing java 8 programs for gpu execution. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE, 419–431.

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

[128] Sverre Jarp. 2002. *A methodology for using the Itanium 2 performance counters for bottleneck analysis*. Technical Report. Technical report, HP Labs.

[129] Brian Jeff. 2012. Big. LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 1143–1146.

[130] Richard Arnold Johnson and Dean W Wichern. 2002. *Applied multivariate statistical analysis*. Vol. 5. Prentice hall Upper Saddle River, NJ.

[131] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. 2000. Java (TM) Language Specification. *Addisson-Wesley, June* (2000).

[132] Agnieszka KamiÅĎska and WÅĆodzimierz Bielecki. 2016. Statistical models to accelerate software development by means of iterative compilation. *Computer Science* 17, 3 (2016), 407. https://journals.agh.edu.pl/csci/article/view/1800

[133] Christos Kartsaklis, Oscar Hernandez, Chung-Hsing Hsu, Thomas Ilsche, Wayne Joubert, and Richard L Graham. 2012. HERCULES: A pattern driven code transformation system. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 574–583.

[134] Christos Kartsaklis, Eunjung Park, and John Cavazos. 2014. HSLOT: the HERCULES scriptable loop transformations engine. In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. IEEE Press, 31–41.

[135] Vasilios Kelefouras. 2017. A methodology pruning the search space of six compiler transformations by addressing them together as one problem and by exploiting the hardware architecture details. *Computing* (2017), 1–24.

[136] William Killian, Renato Miceli, Eunjung Park, Marco Alvarez, and John Cavazos. 2014. Performance Improvement in Kernels by Guiding Compiler Auto-Vectorization Heuristics. *PRACE-RI.EU* (2014). http://www.prace-ri.eu/IMG/pdf/WP183.pdf

[137] Kyoung-jae Kim. 2003. Financial time series forecasting using support vector machines. *Neurocomputing* 55, 1 (2003), 307–319.

[138] Anton Kindestam. 2017. Graph-based features for machine learning driven code optimization. (2017).

[139] Toru Kisuki, P Knijnenburg, M O'Boyle, and H Wijshoff. 2000b. Iterative compilation in program optimization. In *Proc. CPC'10 (Compilers for Parallel Computers)*. Citeseer, 35–44.

[140] Toru Kisuki, Peter MW Knijnenburg, Mike FP O'Boyle, François Bodin, and Harry AG Wijshoff. 1999. A feasibility study in iterative compilation. In *High Performance Computing*. Springer, 121–132.

[141] Toru Kisuki, Peter M. W. Knijnenburg, and Michael F. P. O'Boyle. 2000a. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT'00), Philadelphia, Pennsylvania, USA, October 15-19, 2000*. 237–248. DOI: http://dx.doi.org/10.1109/PACT.2000.888348

[142] P M W Knijnenburg, T Kisuki, and M F P O 'boyle. 2003. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. *The Journal of Supercomputing* 24 (2003), 43–67.

[143] A Koseki. 1997. A method for estimating optimal unrolling times for nested loops. In *Parallel Architectures, Algorithms, and Networks, 1997.(I-SPAN'97) Proceedings., Third International Symposium on*. 376–382. http://ieeexplore.ieee.org/xpls/abs

[144] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[145] David C Ku and Giovanni De Micheli. 1992. Design Space Exploration. In *High Level Synthesis of ASICs under Timing*

and Synchronization Constraints. Springer, 83–111.

[146] P Kulkarni, S Hines, and J Hiser. 2004. Fast searches for effective optimization phase sequences. *ACM SIGPLAN Notices* 39, 6 (2004), 171–182. http://dl.acm.org/citation.cfm?id=996863

[147] Prasad A Kulkarni, Michael R Jantz, and David B Whalley. 2010. Improving both the performance benefits and speed of optimization phase sequence searches. In *ACM Sigplan Notices*, Vol. 45. ACM, 95–104.

[148] Prasad A. Kulkarni, David B. Whalley, and Gary S. Tyson. 2007. Evaluating Heuristic Optimization Phase Order Search Algorithms. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 157–169. DOI: http://dx.doi.org/10.1109/CGO.2007.9

[149] Prasad A Kulkarni, David B Whalley, Gary S Tyson, and Jack W Davidson. 2006. Exhaustive optimization phase order space exploration. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*. IEEE, 13–pp.

[150] Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson, and Jack W. Davidson. 2009. Practical exhaustive optimization phase order exploration and evaluation. *ACM Trans. Archit. Code Optim.* 6, 1, Article 1 (April 2009), 36 pages. DOI: http://dx.doi.org/10.1145/1509864.1509865

[151] S Kulkarni and J Cavazos. 2012. Mitigating the compiler optimization phase-ordering problem using machine learning. *ACM SIGPLAN Notices* (2012). http://dl.acm.org/citation.cfm?id=2384628

[152] S Kulkarni and J Cavazos. 2013. Automatic construction of inlining heuristics using machine learning. *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on* (2013), 1–12. http://ieeexplore.ieee.org/xpls/abs

[153] T Satish Kumar, S Sakthivel, S Sushil Kumar, and N Arun. 2014. Compiler Phase ordering and Optimizing MPI runtime parameters using Heuristic Algorithms on SMPs. *International Journal of Applied Engineering Research* 9, 24 (2014), 30831–30851.

[154] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 75–86.

[155] H Leather, E Bonilla, and M O'Boyle. 2009. Automatic feature generation for machine learning based optimizing compilation. *International Symposium on Code Generation and Optimization, CGO'09* (2009), 81–91. http://ieeexplore.ieee.org/xpls/abs

[156] Bruce W Leverett, Roderic Geoffrey Galton Cattell, Steven O Hobbs, Joseph M Newcomer, Andrew H Reiner, Bruce R Schatz, and William A Wulf. 1979. *An overview of the production quality compiler-compiler project*. Carnegie Mellon University, Department of Computer Science.

[157] Fengqian Li, Feilong Tang, and Yao Shen. 2014. Feature mining for machine learning based compilation optimization. *Proceedings - 2014 8th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2014* (2014), 207–214. DOI: http://dx.doi.org/10.1109/IMIS.2014.26

[158] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. 2009a. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 469–480.

[159] Y Li, J Dongarra, and S Tomov. 2009b. A note on auto-tuning GEMM for GPUs. *Computational ScienceâĂŞICCS 2009* (2009). http://link.springer.com/chapter/10.1007/978-3-642-01970-8

[160] Hui Liu, Rongcai Zhao, Qi Wang, and Yingying Li. 2018. ALIC: A Low Overhead Compiler Optimization Prediction Model. *Wireless Personal Communications* (09 Feb 2018). DOI: http://dx.doi.org/10.1007/s11277-018-5479-x

[161] Vincent Loechner. 1999. PolyLib: A library for manipulating parameterized polyhedra. (1999).

[162] P Lokuciejewski and F Gedikli. 2009. Automatic WCET reduction by machine learning based heuristics for function inlining. *3rd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART)* (2009), 1–15. https://www.researchgate.net/profile/Peter

[163] Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. 2010. Multi-objective exploration of compiler optimizations for real-time systems. *ISORC 2010 - 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing* 1 (2010), 115–122. DOI: http://dx.doi.org/10.1109/ISORC.2010.15

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

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

[166] L Luo, Y Chen, C Wu, S Long, and G Fursin. 2014. Finding representative sets of optimizations for adaptive multiversioning applications. *arXiv preprint arXiv:1407.4075* (2014). http://arxiv.org/abs/1407.4075

[167] Scott A Mahlke, David C Lin, William Y Chen, Richard E Hank, and Roger A Bringmann. 1992. Effective compiler support for predicated execution using the hyperblock. In *ACM SIGMICRO Newsletter*, Vol. 23. IEEE Computer Society

Press, 45–54.
[168] Giovanni Mariani, Aleksandar Brankovic, Gianluca Palermo, Jovana Jovic, Vittorio Zaccaria, and Cristina Silvano. 2010. A correlation-based design space exploration methodology for multi-processor systems-on-chip. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*. IEEE, 120–125.
[169] J Mars and R Hundt. 2009. Scenario based optimization: A framework for statically enabling online optimizations. *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization* (2009). http://dl.acm.org/citation.cfm?id=1545068
[170] LGA Martins and R Nobre. 2016. Clustering-Based Selection for the Exploration of Compiler Optimization Sequences. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016), 28. http://dl.acm.org/citation.cfm?id=2883614
[171] Luiz GA Martins, Ricardo Nobre, Alexandre CB Delbem, Eduardo Marques, and João MP Cardoso. 2014. Exploration of compiler optimization sequences using clustering-based selection. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 63–72.
[172] Amy McGovern, Eliot Moss, and Andrew G Barto. 1999. Scheduling straight-line code using reinforcement learning and rollouts. *Tech report No-99-23* (1999).
[173] Amy McGovern, Eliot Moss, and Andrew G Barto. 2002. Building a basic block instruction scheduler with reinforcement learning and rollouts. *Machine learning* 49, 2-3 (2002), 141–160.
[174] Abdul Wahid Memon and Grigori Fursin. 2013. Crowdtuning: systematizing auto-tuning using predictive modeling and crowdsourcing. In *PARCO mini-symposium on" Application Autotuning for HPC (Architectures)"*.
[175] R Miceli, G Civario, A Sikora, and E César. 2012. Autotune: A plugin-driven approach to the automatic tuning of parallel applications. *International Workshop on Applied Parallel Computing* (2012), 328–342. http://link.springer.com/chapter/10.1007/978-3-642-36803-5
[176] MinIR 2011. MINimal IR space. (2011). http://www.assembla.com/wiki/show/minir-dev.
[177] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. 2012. *Foundations of machine learning*. MIT press.
[178] A Monsifrot, F Bodin, and R Quiniou. 2002. A machine learning approach to automatic production of compiler heuristics. *International Conference on Artificial Intelligence: Methodology, Systems, and Applications* (2002), 41–50. http://link.springer.com/chapter/10.1007/3-540-46148-5
[179] Thierry Moreau, Anton Lokhmotov, and Grigori Fursin. 2018. Introducing ReQuEST: an Open Platform for Reproducible and Quality-Efficient Systems-ML Tournaments. *CoRR* abs/1801.06378 (2018). http://arxiv.org/abs/1801.06378
[180] Eliot Moss, Paul Utgoff, John Cavazos, Doina Precup, D Stefanovic, Carla Brodley, and David Scheeff. 1998. Learning to schedule straight-line code. *Advances in Neural Information Processing Systems* 10 (1998), 929–935. http://books.nips.cc/papers/files/nips10/0929.pdf
[181] Paschalis Mpeis, Pavlos Petoumenos, and Hugh Leather. 2015. Iterative compilation on mobile devices. *CoRR* abs/1511.02603 (2015). http://arxiv.org/abs/1511.02603
[182] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*. 7–10.
[183] M Namolaru, A Cohen, and G Fursin. 2010. Practical aggregation of semantical program properties for machine learning based optimization. In *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*. http://dl.acm.org/citation.cfm?id=1878951
[184] Ricardo Nobre, Reis Luis, and MP Cardoso Joao. 2016. Compiler Phase Ordering as an Orthogonal Approach for Reducing Energy Consumption. In *Proceedings of the 19th Workshop on Compilers for Parallel Computing (CPCâĂŹ16)*.
[185] Ricardo Nobre, Luiz GA Martins, and Joao MP Cardoso. 2015. Use of previously acquired positioning of optimizations for phase ordering exploration. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. ACM, 58–67.
[186] Ricardo Nobre, Luiz G A Martins, and João M P Cardoso. 2016. A Graph-Based Iterative Compiler Pass Selection and Phase Ordering Approach. (2016), 21–30. DOI: http://dx.doi.org/10.1145/2907950.2907959
[187] Ricardo Nobre, Luís Reis, and João M. P. Cardoso. 2018. Impact of Compiler Phase Ordering When Targeting GPUs. In *Euro-Par 2017: Parallel Processing Workshops*, Dora B. Heras and Luc Bougé (Eds.). Springer International Publishing, Cham, 427–438.
[188] Hirotaka Ogawa, Kouya Shimura, Satoshi Matsuoka, Fuyuhiko Maruyama, Yukihiko Sohda, and Yasunori Kimura. 2000. OpenJIT: An open-ended, reflective JIT compiler framework for Java. In *European Conference on Object-Oriented Programming*. Springer, 362–387.
[189] William F Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Minimizing the cost of iterative compilation with active learning. In *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*. IEEE, 245–256.
[190] Karl Joseph Ottenstein. 1978. Data-flow graphs as an intermediate program form. (1978).

[191] David A Padua and Michael J Wolfe. 1986. Advanced compiler optimizations for supercomputers. *Commun. ACM* 29, 12 (1986), 1184–1201.

[192] G Palermo, C Silvano, S Valsecchi, and V Zaccaria. 2003. A system-level methodology for fast multi-objective design space exploration. In *Proceedings of the 13th ACM Great Lakes symposium on VLSI*. ACM, 92–95.

[193] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. 2005a. Multi-objective design space exploration of embedded systems. *Journal of Embedded Computing* 1, 3 (2005), 305–316.

[194] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. 2005b. Multi-objective design space exploration of embedded systems. *Journal of Embedded Computing* 1, 3 (2005), 305–316.

[195] James Pallister, Simon J Hollis, and Jeremy Bennett. 2013. Identifying compiler options to minimize energy consumption for embedded platforms. *Comput. J.* 58, 1 (2013), 95–109.

[196] Z Pan and R Eigenmann. 2004. Rating compiler optimizations for automatic performance tuning. *Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (2004), 14. http://dl.acm.org/citation.cfm?id=1049958

[197] Zhelong Pan and Rudolf Eigenmann. 2006. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 12–pp.

[198] E Park, J Cavazos, and MA Alvarez. 2012. Using graph-based program characterization for predictive modeling. *Proceedings of the International Symposium on Code Generation and Optimization* (2012), 295–305. http://dl.acm.org/citation.cfm?id=2259042

[199] E Park, J Cavazos, and LN Pouchet. 2013. Predictive modeling in a polyhedral optimization space. *International journal of parallel programming* (2013), 704–750. http://link.springer.com/article/10.1007/s10766-013-0241-1

[200] Eunjung Park, Christos Kartsaklis, and John Cavazos. 2014. HERCULES: Strong Patterns towards More Intelligent Predictive Modeling. *2014 43rd International Conference on Parallel Processing* (2014), 172–181. DOI: http://dx.doi.org/10.1109/ICPP.2014.26

[201] E Park, S Kulkarni, and J Cavazos. 2011. An evaluation of different modeling techniques for iterative compilation. (2011), 65–74. http://dl.acm.org/citation.cfm?id=2038711

[202] Eun Jung Park. 2015. Automatic selection of compiler optimizations using program characterization and machine learning title. (2015).

[203] David A Patterson and John L Hennessy. 2013. *Computer organization and design: the hardware/software interface*. Newnes.

[204] Judea Pearl. 1985. BAYESIAN NETWCRKS: A MODEL OF SELF-ACTIVATED MEMORY FOR EVIDENTIAL REASONING. *UCLA Technical Report CSD-850017). Proceedings of the 7th Conference of the Cognitive Science Society, University of California, Irvine, CA*. 3 (1985), 329–334.

[205] Leslie Pérez Cáceres, Federico Pagnozzi, Alberto Franzin, and Thomas Stützle. 2018. Automatic Configuration of GCC Using Irace. In *Artificial Evolution*, Evelyne Lutton, Pierrick Legrand, Pierre Parrend, Nicolas Monmarché, and Marc Schoenauer (Eds.). Springer International Publishing, Cham, 202–216.

[206] R. P J Pinkers, P. M W Knijnenburg, M. Haneda, and H. A G Wijshoff. 2004. Statistical selection of compiler options. *Proceedings - IEEE Computer Society's Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS* (2004), 494–501. DOI: http://dx.doi.org/10.1109/MASCOT.2004.1348305

[207] L. L. Pollock and M. L. Soffa. 1990. Incremental global optimization for faster recompilations. In *Computer Languages, 1990., International Conference on*. 281–290.

[208] LN Pouchet and C Bastoul. 2007. Iterative optimization in the polyhedral model: Part I, one-dimensional time. *International Symposium on Code Generation and Optimization (CGO'07)* (2007), 144–156. http://ieeexplore.ieee.org/xpls/abs

[209] LN Pouchet, C Bastoul, A Cohen, and J Cavazos. 2008. Iterative optimization in the polyhedral model: Part II, multidimensional time. *ACM SIGPLAN Notices* (2008). http://dl.acm.org/citation.cfm?id=1375594

[210] LN Pouchet and U Bondhugula. 2010. Combined iterative and model-driven optimization in an automatic parallelization framework. *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010), 1–11. http://dl.acm.org/citation.cfm?id=1884672

[211] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. *url: http://www. cs. ucla. edu/~ pouchet/software/polybench/[cited July,]* (2012).

[212] S Purini and L Jain. 2013. Finding good optimization sequences covering program space. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 56. http://dl.acm.org/citation.cfm?id=2400715

[213] Matthieu Stéphane Benoit Queva. 2007. Phase-ordering in optimizing compilers. (2007).

[214] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2011. Introduction to recommender systems handbook. In *Recommender systems handbook*. Springer, 1–35.

[215] Ranjit K Roy. 2001. *Design of experiments using the Taguchi approach: 16 steps to product and process improvement*. Wiley-Interscience.

[216] T. Rusira, M. Hall, and P. Basu. 2017. Automating Compiler-Directed Autotuning for Phased Performance Behavior. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1362–1371. DOI: http://dx.doi.org/10.1109/IPDPSW.2017.152

[217] Ricardo Nabinger Sanchez, Jose Nelson Amaral, Duane Szafron, Marius Pirvu, and Mark Stoodley. 2011. Using machines to learn method-specific compilation strategies. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 257–266. http://dl.acm.org/citation.cfm?id=2190072

[218] Vivek Sarkar. 1997. Automatic selection of high-order transformations in the IBM XL FORTRAN compilers. *IBM Journal of Research and Development* 41, 3 (1997), 233–264.

[219] V Sarkar. 2000. Optimized unrolling of nested loops. *Proceedings of the 14th international conference on Supercomputing* (2000), 153–166. http://dl.acm.org/citation.cfm?id=335246

[220] Robert R Schaller. 1997. Moore's law: past, present and future. *Spectrum, IEEE* 34, 6 (1997), 52–59.

[221] E Schkufza, R Sharma, and A Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. *ACM SIGPLAN Notices* (2014). http://dl.acm.org/citation.cfm?id=2594302

[222] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural networks* 61 (2015), 85–117.

[223] Paul B Schneck. 1973. A survey of compiler optimization techniques. In *Proceedings of the ACM annual conference*. ACM, 106–113.

[224] Bernhard Schölkopf. 2001. The kernel trick for distances. In *Advances in neural information processing systems*. 301–307.

[225] Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea Beccari, Luca Benini, Joao MP Cardoso, Carlo Cavazzoni, Radim Cmar, Jan Martinovic, Gianluca Palermo, and others. 2015. ANTAREX–AutoTuning and Adaptivity appRoach for Energy Efficient eXascale HPC Systems. In *Computational Science and Engineering (CSE), 2015 IEEE 18th International Conference on*. IEEE, 343–346.

[226] Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea R Beccari, Luca Benini, João Bispo, Radim Cmar, João MP Cardoso, Carlo Cavazzoni, Jan Martinovič, and others. 2016. AutoTuning and Adaptivity appRoach for Energy efficient eXascale HPC systems: the ANTAREX Approach. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 708–713.

[227] Cristina Silvano, Giovanni Agosta, Stefano Cherubin, Davide Gadioli, Gianluca Palermo, Andrea Bartolini, Luca Benini, Jan Martinovič, Martin Palkovič, Kateřina Slaninová, and others. 2016. The ANTAREX approach to autotuning and adaptivity for energy efficient hpc systems. In *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 288–293.

[228] Cristina Silvano, Andrea Bartolini, Andrea Beccari, Candida Manelfi, Carlo Cavazzoni, Davide Gadioli, Erven Rohou, Gianluca Palermo, Giovanni Agosta, Jan Martinovič, and others. 2017. The ANTAREX tool flow for monitoring and autotuning energy efficient HPC systems. In *SAMOS 2017-International Conference on Embedded Computer Systems: Architecture, Modeling and Simulation*.

[229] Cristina Silvano, William Fornaciari, Gianluca Palermo, Vittorio Zaccaria, Fabrizio Castro, Marcos Martinez, Sara Bocchio, Roberto Zafalon, Prabhat Avasare, Geert Vanmeerbeeck, and others. 2011. Multicube: Multi-objective design space exploration of multi-core architectures. In *VLSI 2010 Annual Symposium*. Springer, 47–63.

[230] Cristina Silvano, Gianluca Palermo, Giovanni Agosta, Amir H. Ashouri, Davide Gadioli, Stefano Cherubin, Emanuele Vitali, Luca Benini, Andrea Bartolini, Daniele Cesarini, Joao Cardoso, Joao Bispo, Pedro Pinto, Riccardo Nobre, Erven Rohou, Loïc Besnard, Imane Lasri, Nico Sanna, Carlo Cavazzoni, Radim Cmar, Jan Martinovič, Kateřina Slaninová, Martin Golasowski, Andrea R. Beccari, and Candida Manelfi. 2018. Autotuning and Adaptivity in Energy Efficient HPC Systems: The ANTAREX toolbox. In *Proceedings of the Computing Frontiers Conference*. ACM.

[231] Bernard W Silverman. 1986. *Density estimation for statistics and data analysis*. Vol. 26. CRC press.

[232] Richard Stallman. 2001. Using and porting the GNU compiler collection. In *MIT Artificial Intelligence Laboratory*. Citeseer.

[233] Richard M Stallman and others. 2003. *Using GCC: the GNU compiler collection reference manual*. Gnu Press.

[234] Kenneth O Stanley. 2002. Efficient reinforcement learning through evolving neural network topologies. In *In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002*. Citeseer.

[235] MW Stephenson. 2006. Automating the construction of compiler heuristics using machine learning. (2006). http://groups.csail.mit.edu/commit/papers/2006/stephenson_phdthesis.pdf

[236] M Stephenson and S Amarasinghe. 2003. Meta optimization: improving compiler heuristics with machine learning. 38, 5 (2003), 77–90. http://dl.acm.org/citation.cfm?id=781141

[237] M Stephenson and S Amarasinghe. 2005. Predicting unroll factors using supervised classification. In *International symposium on code generation and optimization*. DOI: http://dx.doi.org/10.1109/CGO.2005.29

[238] M Stephenson and UM O'Reilly. 2003. Genetic programming applied to compiler heuristic optimization. *GEuropean Conference on Genetic Programming* (2003), 238–253. http://link.springer.com/chapter/10.1007/3-540-36599-0

[239] Ralph E Steuer. 1986. *Multiple criteria optimization: theory, computation, and applications.* Wiley.

[240] K Stock, LN Pouchet, and P Sadayappan. 2012. Using machine learning to improve automatic vectorization. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012), 50. http://dl.acm.org/citation.cfm?id=2086729

[241] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. 2000. Overview of the IBM Java just-in-time compiler. *IBM systems Journal* 39, 1 (2000), 175–193.

[242] Cristian Ţăpuş, I-Hsin Chung, Jeffrey K Hollingsworth, and others. 2002. Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 1–11.

[243] Michele Tartara and Stefano Crespi Reghizzi. 2012. Parallel iterative compilation: using MapReduce to speedup machine learning in compilers. In *Proceedings of third international workshop on MapReduce and its Applications Date*. ACM, 33–40.

[244] Michele Tartara and Stefano Crespi Reghizzi. 2013a. Continuous learning of compiler heuristics. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 46.

[245] Michele Tartara and Stefano Crespi Reghizzi. 2013b. Continuous learning of compiler heuristics. *ACM Trans. Archit. Code Optim.* 9, 4, Article 46 (Jan. 2013), 25 pages. DOI: http://dx.doi.org/10.1145/2400682.2400705

[246] Gerald Tesauro and Gregory R Galperin. 1996. On-line policy improvement using Monte-Carlo search. In *NIPS*, Vol. 96. 1068–1074.

[247] Bruce Thompson. 2002. Statistical, practical, and clinical: How many kinds of significance do counselors need to consider? *Journal of Counseling & Development* 80, 1 (2002), 64–71.

[248] J Thomson, M O'Boyle, G Fursin, and B Franke. 2009. Reducing training time in a one-shot machine learning-based compiler. *International Workshop on Languages and Compilers for Parallel Computing* (2009), 399–407. http://link.springer.com/10.1007

[249] A Tiwari, C Chen, and J Chame. 2009. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. 1–12. http://ieeexplore.ieee.org/xpls/abs

[250] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP MFP O'Boyle. 2009. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. *ACM Sigplan Notices* (2009), 177–187. DOI: http://dx.doi.org/10.1145/1543135.1542496

[251] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D.I. August. 2003. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE Comput. Soc, 204–215. DOI: http://dx.doi.org/10.1109/CGO.2003.1191546

[252] Eben Upton and Gareth Halfacree. 2014. *Raspberry Pi user guide.* John Wiley & Sons.

[253] K Vaswani. 2007. Microarchitecture sensitive empirical models for compiler optimizations. *International Symposium on Code Generation and Optimization (CGO'07)* (2007), 131–143. http://ieeexplore.ieee.org/xpls/abs

[254] Steven R. Vegdahl. 1982. Phase coupling and constant generation in an optimizing microcode compiler. *ACM SIGMICRO Newsletter* 13, 4 (1982), 125–133.

[255] Richard Vuduc, James W Demmel, and Jeff A Bilmes. 2004. Statistical models for empirical search-based performance tuning. *International Journal of High Performance Computing Applications* 18, 1 (2004), 65–94.

[256] Richard W. Vuduc. 2011. *Autotuning.* Springer US, Boston, MA, 102–105. DOI: http://dx.doi.org/10.1007/978-0-387-09766-4_68

[257] David W Wall. 1991. *Limits of instruction-level parallelism.* Vol. 19. ACM.

[258] Wei Wang, John Cavazos, and Allan Porterfield. 2014. Energy auto-tuning using the polyhedral approach. In *Workshop on Polyhedral Compilation Techniques*.

[259] Z Wang and MFP O'Boyle. 2009. Mapping parallelism to multi-cores: a machine learning based approach. *ACM Sigplan notices* (2009). http://dl.acm.org/citation.cfm?id=1504189

[260] Todd Waterman. 2006. *Adaptive compilation and inlining.* Ph.D. Dissertation. Rice University.

[261] Deborah Whitfield and Mary Lou Soffa. 1991. Automatic generation of global optimizers. In *ACM SIGPLAN Notices*, Vol. 26. ACM, 120–129.

[262] D. Whitfield, M. L. Soffa, D. Whitfield, and M. L. Soffa. 1990. An approach to ordering optimizing transformations. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming - PPOPP '90*, Vol. 25. ACM Press, New York, New York, USA, 137–146. DOI: http://dx.doi.org/10.1145/99163.99179

[263] Deborah L. Whitfield and Mary Lou Soffa. 1997. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems* 19, 6 (nov 1997), 1053–1084. DOI: http://dx.doi.org/10.1145/267959.267960

[264] Doran K. Wilde. 1993. *A Library for Doing Polyhedral Operations.* Technical Report 785. IRISA.

[265] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[266] Robert P Wilson, Robert S French, Christopher S Wilson, Saman P Amarasinghe, Jennifer M Anderson, Steve WK Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W Hall, Monica S Lam, and others. 1994. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices* 29, 12 (1994), 31–37.

[267] MI Wolczko and DM Ungar. 2000. Method and apparatus for improving compiler performance during subsequent compilations of a source program. *US Patent 6,078,744* (2000). https://www.google.com/patents/US6078744

[268] Stephan Wong, Thijs Van As, and Geoffrey Brown. 2008. ρ-VEX: A reconfigurable and extensible softcore VLIW processor. In *ICECE Technology, 2008. FPT 2008. International Conference on*. IEEE, 369–372.

[269] William Allan Wulf, Richard K Johnsson, Charles B Weinstock, Steven O Hobbs, and Charles M Geschke. 1975. *The design of an optimizing compiler.* Elsevier Science Inc.

[270] T Yuki, V Basupalli, G Gupta, G Iooss, and D Kim. 2012a. Alphaz: A system for analysis, transformation, and code generation in the polyhedral equational model. (2012). http://www.cs.colostate.edu/TechReports/Reports/2012/tr12-101.pdf

[271] T Yuki, G Gupta, DG Kim, T Pathan, and S Rajopadhye. 2012b. AlphaZ: A System for Design Space Exploration in the Polyhedral Model, In International Workshop on Languages and Compilers for Parallel Computing. *people.rennes.inria.fr* (2012), 17–31. http://people.rennes.inria.fr/Tomofumi.Yuki/papers/yuki-lcpc2012.pdf

[272] Vittorio Zaccaria, Gianluca Palermo, Fabrizio Castro, Cristina Silvano, and Giovanni Mariani. 2010. Multicube explorer: An open source framework for design space exploration of chip multi-processors. In *Architecture of Computing Systems (ARCS), 2010 23rd International Conference on*. VDE, 1–7.

[273] Min Zhao, Bruce Childers, Mary Lou Soffa, Min Zhao, Bruce Childers, and Mary Lou Soffa. 2003. Predicting the impact of optimizations for embedded systems. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems - LCTES '03*, Vol. 38. ACM Press, New York, New York, USA, 1. DOI: http://dx.doi.org/10.1145/780732.780734

[274] Min Zhao, Bruce R Childers, and Mary Lou Soffa. 2005. A model-based framework: an approach for profit-driven optimization. In *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 317–327.

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