INVITED PAPER

Machine Learning in Compiler Optimization

By Zheng Wang and Michael O'Boyle

ABSTRACT | In the last decade, machine-learning-based compilation has moved from an obscure research niche to a mainstream activity. In this paper, we describe the relationship between machine learning and compiler optimization and introduce the main concepts of features, models, training, and deployment. We then provide a comprehensive survey and provide a road map for the wide variety of different research areas. We conclude with a discussion on open issues in the area and potential research directions. This paper provides both an accessible introduction to the fast moving area of machine-learning-based compilation and a detailed bibliography of its main achievements.

KEYWORDS | Code optimization; compiler; machine learning; program tuning

I. INTRODUCTION

"Why would anyone want to use machine learning to build a compiler?" It is a view expressed by many colleagues over the last decade. Compilers translate programming languages written by humans into binary executable by computer hardware. It is a serious subject studied since the 1950s [1]–[3] where correctness is critical and caution is a by-word. Machine learning, on the other hand, is an area of artificial intelligence (AI) aimed at detecting and predicting patterns. It is a dynamic field looking at subjects as diverse as galaxy classification [4] to predicting elections based on Tweeter feeds [5]. When an open-source machine learning compiler was announced by IBM in 2009 [6], some wry slashdot commentators picked up on the AI aspect, predicting the start of sentient computers, global net, and the war with machines from the *Terminator* film series.

In fact, as we will see, in this paper, that compilers and machine learning are a natural fit and have developed into an established research domain.

Manuscript received October 30, 2017; accepted January 23, 2018.
(Corresponding author: Michael O'Boyle.)
Z. Wang is with the MetaLab, School of Computing and Communications, Lancaster University, Lancaster LA1 4WA, U.K. (e-mail: z.wang@lancaster.ac.uk).
M. O'Boyle is with the School of Informatics, University of Edinburgh, Edinburgh EH8 9AB, U.K. (e-mail: mob@inf.ed.ac.uk).

Digital Object Identifier: 10.1109/JPROC.2018.2817118

A. It Is All About Optimization

Compilers have two jobs—translation and optimization. First, they must translate programs into binary correctly. Second, they have to find the most efficient translation possible. There are many different correct translations whose performance varies significantly. The vast majority of research and engineering practices is focused on this second goal of performance, traditionally misnamed optimization. The goal was misnamed because in most cases, until recently, finding an optimal translation was dismissed as being too hard to find and an unrealistic endeavor.[1] Instead it focused on developing compiler heuristics to transform the code in the hope of improving performance but could in some instances damage it.

Machine learning predicts an outcome for a new data point based on prior data. In its simplest guise, it can be considered a form of interpolation. This ability to predict based on prior information can be used to find the data point with the best outcome and is closely tied to the area of optimization. It is at this overlap of looking at code improvement as an optimization problem and machine learning as a predictor of the optima where we find machine learning compilation.

Optimization as an area, machine learning based or otherwise, has been studied since the 1800s [8], [9]. An interesting question is therefore why has the convergence of these two areas taken so long? There are two fundamental reasons. First, despite the year-on-year increasing potential performance of hardware, software is increasingly unable to realize it leading to a software gap. This gap has yawned right open with the advent of multicores (see also Section VI-B). Compiler writers are looking for new ways to bridge this gap.

Second, computer architecture evolves so quickly that it is difficult to keep up. Each generation has new quirks and compiler writers are always trying to play catchup. Machine learning has the desirable property of being automatic. Rather than relying on expert compiler writers to develop clever heuristics to optimize the code, we can let the machine learn how to optimize a compiler to make the machine run faster, an approach

[1]In fact, the term superoptimizer [7] was coined to describe systems that tried to find the optimum.

sometimes referred to as autotuning [10]–[13]. Machine learning is, therefore, ideally suited to making any code optimization decision where the performance impact depends on the underlying platform. As described later in this paper, it can be used for topics ranging from selecting the best compiler flags to determining how to map parallelism to processors.

Machine learning is part of a tradition in computer science and compilation in increasing automation The 1950s to 1970s were spent trying to automate compiler translation, e.g., lex for lexical analysis [14] and yacc for parsing [15]; the last decade by contrast has focused on trying to automate compiler optimization. As we will see, it is not "magic" or a panacea for compiler writers, rather it is another tool allowing automation of tedious aspects of compilation providing new opportunities for innovation. It also brings compilation nearer to the standards of evidence-based science. It introduces an experimental methodology where we separate out evaluation from design and considers the robustness of solutions. Machine-learning-based schemes, in general, have the problem of relying on black boxes whose working we do not understand and hence trust. This problem is just as true for machine-learning-based compilers. In this paper, we aim to demystify machine-learning-based compilation and show it is a trustworthy and exciting direction for compiler research.

The remainder of this paper is structured as follows. First, we give an intuitive overview for machine learning in compilers in Section II. Then, we describe how machine learning can be used to search for or to directly predict good compiler optimizations in Section III. This is followed by a comprehensive discussion in Section IV for a wide range of machine learning models that have been employed in prior work. Next, in Section V, we review how previous work chooses quantifiable properties, or features, to represent programs. We discuss the challenges and limitations for applying machine learning to compilation, as well as open research directions in Section VII before we summarize and conclude in Section VIII.

II. OVERVIEW OF MACHINE LEARNING IN COMPILERS

Given a program, compiler writers would like to know what compiler heuristic or optimization to apply in order to make the code better. Better often means execute faster, but can also mean smaller code footprint or reduced power. Machine learning can be used to build a model used within the compiler that makes such decisions for any given program.

There are two main stages involved: learning and deployment. The first stage learns the model based on training data, while the second uses the model on new unseen programs. Within the learning stage, we need a way of representing programs in a systematic way. This representation is known as the program features [16].

Fig. 1 gives an intuitive view on how machine learning can be applied to compilers. This process, which includes feature engineering, learning a model, and deployment, is described in the following sections.

A. Feature Engineering

Before we can learn anything useful about programs, we first need to be able to characterize them. Machine learning relies on a set of quantifiable properties, or features, to characterize the programs [Fig. 1(a)]. There are many different features that can be used. These include the static data structures extracted from the program source code or the compiler intermediate representation (such as the number of instructions or branches), dynamic profiling information (such as performance counter values) obtained through runtime profiling, or a combination of the both.

Standard machine learning algorithms typically work on fixed length inputs, so the selected properties will be summarized into a fixed length feature vector. Each element of the vector can be an integer, real or Boolean value. The process of feature selection and tuning is referred to as feature engineering. This process may need to iteratively perform multiple times to find a set of high-quality features to build

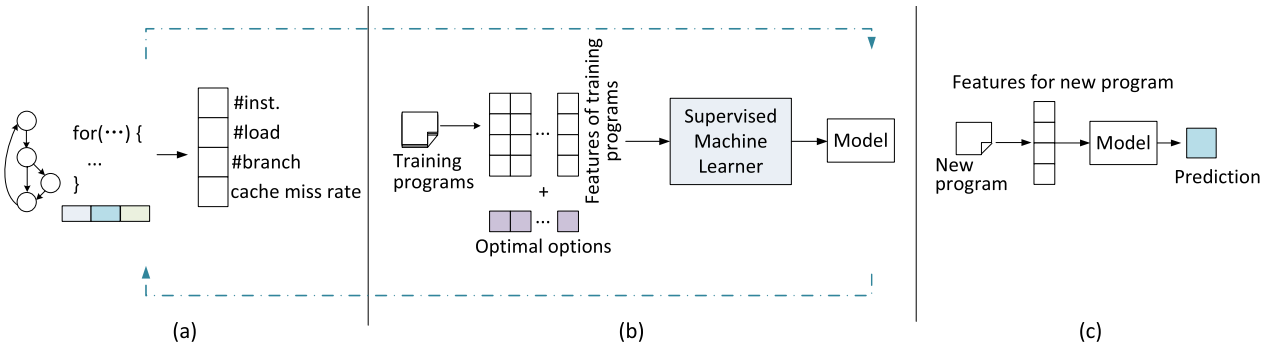

Fig. 1. *A generic view of supervised machine learning in compilers. (a) Feature engineering. (b) Leaning a model. (c) Deployment.*

an accurate machine learning model. In Section V, we provide a comprehensive review of feature engineering for the topic of program optimization.

B. Learning a Model

The second step is to use training data to derive a model using a learning algorithm. This process is depicted in Fig. 1(b) . Unlike other applications of machine learning, we typically generate our own training data using existing applications or benchmarks. The compiler developer will select training programs which are typical of the application domain. For each training program, we calculate the feature values, compiling the program with different optimization options, and running and timing the compiled binaries to discover the best performing option. This process produces, for each training program, a training instance that consists of the feature values and the optimal compiler option for the program.

The compiler developer then feeds these examples to a machine learning algorithm to automatically build a model. The learning algorithm's job is to find from the training examples a correlation between the feature values and the optimal optimization decision. The learned model can then be used to predict, for a new set of features, what the optimal optimization option should be.

Because the performance of the learned model strongly depends on how well the features and training programs are chosen, the processes of featuring engineering and training data generation often need to repeat multiple times.

C. Deployment

In the final step, the learned model is inserted into the compiler to predict the best optimization decisions for new programs. This is demonstrated in Fig. 1(c) . To make a prediction, the compiler first extracts the features of the input program, and then feeds the extracted feature values to the learned model to make a prediction.

The advantage of the machine-learning-based approach is that the entire process of building the model can be easily repeated whenever the compiler needs to target a new hardware architecture, operating system, or application domain. The model built is entirely derived from experimental results and is hence evidence based.

D. Example

As an example to illustrate these steps, consider thread coarsening [18] for GPU programs. This code transformation technique works by giving multiple work items (or work elements) to one single thread. It is similar to loop unrolling, but applied across parallel work items rather than across serial loop iterations.

Fig. 2(a) shows a simple OpenCL kernel where a thread operates on a work item of the 1-D input array, in, at a time. The work item to be operated on is specified by the value

```
1  kernel void square(global float* in, global float* out){
2      int gid = get_global_id(0);
3      out[gid] = in[gid] * in[gid];
4  }
```

(a)

```
1  kernel void square(global float* in, global float* out){
2      int gid = get_global_id(0);
3      int tid0 = 2*gid + 0;
4      int tid1 = 2*gid + 1;
5      out[tid0] = in[tid0] * in[tid0];
6      out[tid1] = in[tid1] * in[tid1];
7  }
```

(b)

Fig. 2. ***An OpenCL thread coarsening example reproduced from [17]. The original OpenCL code is shown in (a) where each thread takes the square of one element of the input array. When coarsened by a factor of two, as shown in (b), each thread now processes two elements of the input array.***

returned from the OpenCL get_global_id() API. Fig. 2(b) shows the transformed code after applying a thread coarsen factor of two, where each thread processes two elements of the input array.

Thread coarsening can improve performance through increasing instruction-level parallelism [19], reducing the number of memory-access operations [20] and eliminating redundant computation when the same value is computed in every work item. However, it can also have several negative side effects, such as reducing the total amount of parallelism and increasing the register pressure, which can lead to slowdown performance. Determining when and how to apply thread coarsening is nontrivial, because the best coarsening factor depends on the target program and the hardware architecture that the program runs on [17], [19].

Magni *et al.* show that machine learning techniques can be used to automatically construct effective thread-coarsening heuristics across GPU architectures [17]. Their approach considers six coarsening factors $(1, 2, 4, 8, 16, 32)$. The goal is to develop a machine-learning-based model to decide whether an OpenCL kernel should be coarsened on a specific GPU architecture and, if so, what is the best coarsening factor. Among many machine learning algorithms, they chose to use an artificial neural network to model[2] the problem. Construing such a model follows the classical three-step supervised learning process, which is depicted in Fig. 1 and described in more details as follows.

1) Feature Engineering: To describe the input OpenCL kernel, Magni *et al.* use static code features extracted from the compiler's intermediate representation. Specifically, they developed a compiler-based tool to obtain the feature values from the program's LLVM bitcode [21]. They started from 17 candidate features. These include things like the number of

[2]In fact, Magni *et al.* employed a hierarchical approach consisting of multiple artificial neural networks [17]. However, these networks are trained using the same process.

and types of instructions and memory level parallelism (MLP) within an OpenCL kernel. Table 1 gives the list of candidate features used in [17]. Typically, candidate features can be chosen based on developers' intuitions, suggestions from prior works, or a combination of both. After choosing the candidate features, a statistical method called principal component analysis (PCA; see also Section IV-B) is applied to map the 17 candidate features into seven aggregated features, so that each aggregated feature is a linear combination of the original features. This technique is known as "feature dimension reduction," which is discussed in Section V-D2. Dimension reduction helps eliminating redundant information among candidate features, allowing the learning algorithm to perform more effectively.

2) Learning the Model: For the work presented in [17], 16 OpenCL benchmarks were used to generate training data. To find out which of the six coarsening factors performs best for a given OpenCL kernel on a specific GPU architecture, we can apply each of the six factors to an OpenCL kernel and record its execution time. Since the optimal thread-coarsening factor varies across hardware architectures, this process needs to repeat for each target architecture. In addition to finding the best performing coarsening factor, Magni *et al.* also extracted the aggregated feature values for each kernel. Applying these two steps on the training benchmarks results in a training data set where each training example is composed of the optimal coarsening factor and feature values for a training kernel. The training examples are then fed into a learning algorithm which tries to find a set of model parameters (or weights) so that overall prediction error on the training examples can be minimized. The output of the learning algorithm is an artificial neural network model where its weights are determined from the training data.

3) Deployment: The learned model can then be used to predict the optimal coarsening factor for unseen OpenCL programs. To do so, static source code features are first extracted from the target OpenCL kernel; the extracted feature values are then fed into the model which decides whether to coarsen or not and which coarsening factor should be used. The technique proposed in [17] achieves an average speedup between 1.11x and 1.33x across four GPU architectures and does not lead to degraded performance on a single benchmark.

Table 1 Candidate Code Features Used in [17]

Feature Description	Feature Description
# Basic Blocks	# Branches
# Divergent Instr.	# Instrs. in Divergent Regions
(# instr. in Divergent regions)/(# total instr.)	# Divergent regions
# Instrs	# Floating point instr.
Avg. ILP per basic block	(# integer instr.) / (# floating point instr.)
# integer instr.	# Math built-in func.
Avg. MLP per basic block	# loads
# stores	# loads that are independent of the coarsening direction
# barriers	

III. METHODOLOGY

One of the key challenges for compilation is to select the right code transformation, or sequence of transformations for a given program. This requires effectively evaluating the quality of a possible compilation option, e.g., how a code transformation will affect eventual performance.

A naive approach is to exhaustively apply each legal transformation option and then profile the program to collect the relevant performance metric. Given that many compiler problems have a massive number of options, exhaustive search and profiling is infeasible, prohibiting the use of this approach at scale. This search-based approach to compiler optimization is known as iterative compilation [22], [23] or autotuning [10], [24]. Many techniques have been proposed to reduce the cost of searching a large space [25], [26]. In certain cases, the overhead is justifiable if the program in question is to be used many times, e.g., in a deeply embedded device. However, its main limitation remains: it only finds a good optimization for one program and does not generalize into a compiler heuristic.

There are two main approaches for solving the problem of scalably selecting compiler options that work across programs. A high level comparison of both approaches is given in Fig. 3. The first strategy attempts to develop a cost (or priority) function to be used as a proxy to estimate the quality of a potential compiler decision, without relying on extensive profiling. The second strategy is to directly predict the best performing option.

A. Building a Cost Function

Many compiler heuristics rely on a cost function to estimate the quality of a compiler option. Depending on the optimization goal, the quality metric can be execution time, the code size, or energy consumption, etc. Using a cost function, a compiler can evaluate a range of possible options to choose the best one, without needing to compile and profile the program with each option.

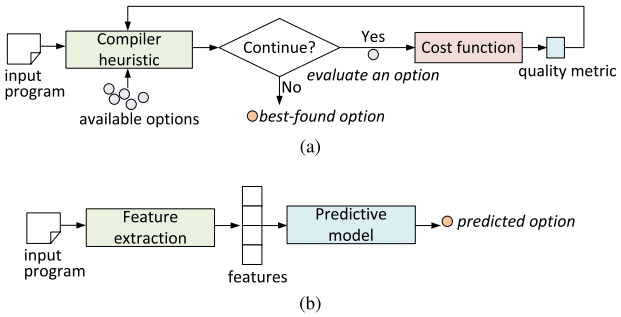

Fig. 3. *There are, in general, two approaches to determine the optimal compiler decision using machine learning. The first one is to learn a cost or priority function to be used as a proxy to select the best performing option (a). The second one is to learn a predictive model to directly predict the best option (b).*

1) The Problem of Handcrafted Heuristics: Traditionally, a compiler cost function is manually crafted. For example, a heuristic of function inlining adds up a number of relevant metrics, such as the number of instructions of the target function to be inlined, the callee and stack size after inlining, and compare the resulted value against a predefined threshold to determine if it is profitable to inline a function [27]. Here, the importance or weights for metrics and the threshold are determined by compiler developers based on their experience or via "trail-and-error." Because the efforts involved in tuning the cost function are so expensive, many compilers simply use "one-size-fits-all" cost function for inlining. However, such a strategy is ineffective. For examples, Cooper *et al.* show that a "one-size-fits-all" strategy for inlining often delivers poor performance [28]; other studies also show that the optimal thresholds to use to determine when to inline change from one program to the other [29], [30].

Handcrafted cost functions are widely used in compilers. Other examples include the work conducted by Wagner *et al.* [31] and Tiwari *et al.* [32]. The former combines a Markov model and a human-derived heuristic to statically estimate the execution frequency of code regions (such as function innovation counts). The latter calculates the energy consumption of an application by assigning a weight to each instruction type. The efficiency of these approaches highly depends on the accuracy of the estimations given by the manually tuned heuristic.

The problem of relying on a hand-tuned heuristic is that the cost and benefit of a compiler optimization often depend on the underlying hardware; while handcrafted cost functions could be effective, manually developing one can take months or years on a single architecture. This means that tuning the compiler for each newly released processor is hard and is often infeasible due to the drastic efforts involved. Because cost functions are important and manually tuning a good function is difficult for each individual architecture, researchers have investigated ways to use machine learning to automate this process.

In Section III-A2, we review a range of previous studies on using machine learning to tune cost functions for performance and energy consumption—many of which can be applied to other optimization targets such as the code size [33] or a tradeoff between energy and runtime.

2) Cost Functions for Performance: The Meta Optimization framework [34] uses genetic programming (GP) to search for a cost function $y \leftarrow f(x)$, which takes in a feature vector x and produces a real-valued priority y. Fig. 4 depicts the workflow of the framework. This approach is evaluated on a number of compiler problems, including hyperblock formation,[3] register allocation, and data prefetching, showing that machine learned cost functions outperform human-crafted ones. A similar approach is employed by Cavazos *et al.* who find cost functions for performance and compilation overhead for a Java just-in-time compiler [35]. The COLE compiler [36] uses a variance of the GP algorithm called strength Pareto evolutionary algorithm 2 (SPEA2) [37] to learn cost functions to balance multiple objectives (such as program runtime, compilation overhead, and code size). In Section IV-C, we describe the working mechanism of GP-like search algorithms.

Another approach to tune the cost functions is to predict the execution time or speedup of the target program. The Qilin compiler [38] follows such an approach. It uses curve fitting algorithms to estimate the runtime for executing the target program of a given input size on the CPU and the GPU. The compiler then uses this information to determine the optimal loop iteration partition across the CPU and the GPU. The Qilin compiler relies on an application-specific function which is built on a per program base using reference inputs. The curve fitting (or regression; see, also, Section IV) model employed by the Qilin compiler can model with continuous values, making it suitable for estimating runtime and speedup. In [39], this approach is extended, which developed a relative predictor that predicts whether an unseen predictor will improve significantly on a GPU relative to a CPU. This is used for runtime scheduling of OpenCL jobs.

The early work conduced by Brewer proposed a regression-based model to predict the execution of a data layout scheme for parallelization, by considering three parameters [40]. Using the model, his approach can select the optimal

[3]Hyperblock formation combines basic blocks from multiple control paths to form a predicated, larger code block to expose instruction level parallelism.

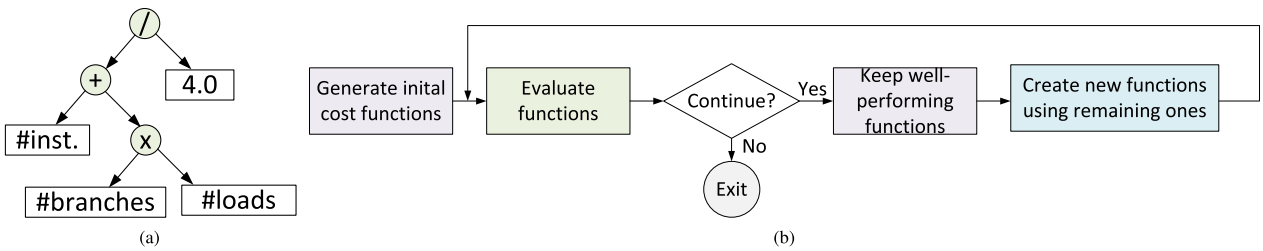

Fig. 4. ***A simple view of the GP approach presented in [34] for tuning compiler cost functions. Each candidate cost function is represented as an expression tree (a). The workflow of the GP algorithm is presented in (b).***

layout for over 99% of the time for a partial differential equation (PDE) solver across four evaluation platforms. Other previous works also use curve fitting algorithms to build a cost function to estimate the speedup or runtime of sequential [41]–[43], OpenMP [44]–[46], and, more recently, deep learning applications [47].

3) Cost Functions for Energy Consumption: In addition to performance, there is an extensive body of work that investigates ways to build energy models for software optimization and hardware architecture design. As power or energy readings are continuous real values, most of the prior work on power modeling uses regression-based approaches.

Linear regression is a widely used technique for energy modeling. Benini *et al.* developed a linear-regression-based model to estimate power consumption at the instruction level [48]. The framework presented by Rethinagiri *et al.* [49] uses parameterized formulas to estimate power consumption of embedded systems. The parameters of the formulas are determined by applying a regression-based algorithm to reference data obtained with handcrafted assembly code and power measurements. In a more recent work, Schürmans *et al.* also adopt a regression-based method for power modeling [50], but the weights of the regression model are determined using standard benchmarks instead of handwritten assembly programs.

Other works employ the artificial neural network (ANN) to automatically construct power models. Curtis-Maury *et al.* develop an ANN-based model to predict the power consumption of OpenMP programs on multicore systems [51]. The inputs to the model are hardware performance counter values such as the cache miss rate, and the output is the estimated power consumption. Su *et al.* adopt a similar approach by developing an ANN predictor to estimate the runtime and power consumption for mapping OpenMP programs on nonuniform memory access (NUMA) multicores. This approach is also based on runtime profiling of the target program, but it explicitly considers NUMA-specific information like local and remote memory accesses per cycle.

B. Directly Predicting the Best Option

While a cost function is useful for evaluating the quality of compiler options, the overhead involved in searching for the optimal option may still be prohibitive. For this reason, researchers have investigated ways to directly predict the best compiler decision using machine learning for relatively small compilation problems.

Monsifrot *et al.* pioneered the use of machine learning to predict the optimal compiler decision [16]. This work developed a decision-tree-based approach to determine whether it is beneficial to unroll a loop based on information such as the number of statements and arithmetic operations of the loop. Their approach makes a binary decision on whether to unroll a loop but not how many times the loop should be unrolled. Later, Stephenson and Amarasinghe advanced [16] by directly predicting the loop unroll factor [52] by considering eight unroll factors $(1, 2, \ldots, 8)$. They formulated the problem as a multiclass classification problem (i.e., each loop unroll factor is a class). They used over 2500 loops from 72 benchmarks to train two machine learning models [a nearest neighbor and a support vector machine (SVM) model] to predict the loop unroll factor for unseen loops. Using a richer set of features than [16], their techniques correctly predict the unroll factor for 65% of the testing loops, leading to, on average, a 5% improvement for the SPEC 2000 benchmark suite.

For sequential programs, there is extensive work in predicting the best compiler flags [53], [54], code transformation options [55], or tile size for loops [56], [57]. This level of interest is possibly due to the restricted nature of the problem, allowing easy experimentation and comparison against prior work.

Directly predicting the optimal option for parallel programs is harder than doing it for sequential programs, due to the complex interactions between the parallel programs and the underlying parallel architectures. Nonetheless, there are works on predicting the optimal number of threads to be used to run an OpenMP program [46], [58], the best parameters to be used to compile a CUDA programs for a given input [59], and the thread coarsening parameters for OpenCL programs for GPUs [17]. These papers show that supervised machine learning can be a powerful tool for modeling problems with a relatively small number of optimization options.

IV. MACHINE LEARNING MODELS

In this section, we review the wide range of machine learning models used for compiler optimization. Table 2 summarizes the set machine learning models discussed in this section.

There are two major subdivisions of machine learning techniques that have previously been used in compiler optimizations: supervised and unsupervised learning. Using supervised machine learning, a predictive model is trained on empirical performance data (labeled outputs) and important quantifiable properties (features) of representative programs. The model learns the correlation between these feature values and the optimization decision that delivers the optimal (or near-optimal) performance. The learned correlations are used to predict the best optimization decisions for new programs. Depending on the nature of the outputs, the predictive model can be either a regression model for continuous outputs or a classification model for discrete outputs.

In the other subdivision of machine learning, termed unsupervised learning, the input to the learning algorithm is a set of input values merely—there is no labeled output. One form of unsupervised learning is clustering which groups the input data items into several subsets. For example, SimPoint [60], a simulation technique, uses clustering to pick represent program execution points for program simulation. It does so by first dividing a set of program runtime information

Table 2 Machine Learning Methods Discussed in Section IV

Approach	Problem	Application Domains	Models
Supervised learning	Regression	Useful for modelling continuous values, such as estimating execution time, speedup, power consumption, latency etc.	Linear/non-linear regression, artificial neural networks (ANNs), support vector machines (SVMs).
	Classification	Useful for predicting discrete values, such as choosing compiler flags, #threads, loop unroll factors, algorithmic implementations etc.	K-nearest neighbour (KNN), decision trees, random forests, logical regression, SVM, Kernel Canonical Correlation Analysis, Bayesian
Unsupervised learning	Clustering	Data analysis, such as grouping profiling traces into clusters of similar behaviour	K-means, Fast Newman clustering
	Feature engineering	Feature dimension reduction, finding useful feature representations	Principal component analysis (PCA), autoencoders
Online learning	Search and self-learning	Useful for exploring a large optimisation space, runtime adaption, dynamic task scheduling where the optimal outcome is achieved through a series of actions	Genetic algorithm (GA), genetic programming (GP), reinforcement learning (RL)

into groups (or clusters), such that points within each cluster are similar to each other in terms of program structures (loops, memory usages, etc.); it then chooses a few points of each cluster to represent all the simulation points within that group without losing much information.

There are also techniques that sit at the boundary of supervised and unsupervised learning. These techniques refine the knowledge gathered during offline learning or previous runs using empirical observations obtained during deployment. We review such techniques in Section IV-C. This sections concludes with a discussion of the relative merits of different modeling approaches for compiler optimization.

A. Supervised Learning

1) Regression: A widely used supervised learning technique is called regression. This technique has been used in various tasks, such as predicting the program execution time input [38] or speedup [39] for a given input, or estimating the tail latency for parallel workloads [61].

Regression is essentially curve fitting. As an example, consider Fig. 5 where a regression model is learned from five data points. The model takes in a program input size X and predicts the execution time of the program Y. Adhering to supervised learning nomenclature, the set of five known data points is the training data set and each of the five points that comprise the training data is called a training example. Each training example (x_i, y_i) is defined by a feature vector (i.e., the input size in our case) x_i and a desired output (i.e., the program execution time in our case) y_i. Learning in this context is understood as discovering the relation between the inputs (x_i) and the outputs (y_i) so that the predictive model can be used to make predictions for any new, unseen input features in the problem domain. Once the function f is in place, one can use it to make a prediction by taking in a new input feature vector x. The prediction y is the value of the curve that the new input feature vector x corresponds to.

There are a range of machine learning techniques that can be used for regression. These include the simple linear regression model and more advanced models like SVMs and ANNs. Linear regression is effective when the input (i.e., feature vectors) and output (i.e., labels) have a strong linear relation. SVM and ANNs can model both linear and nonlinear relations, but typically require more training examples to learn an effective model when compared with simple linear regression models.

Table 3 gives some examples of regression techniques that have been used in prior work for code optimization and the problem to be modeled.

2) Classification: Supervised classification is another technique that has been widely used in prior work of machine-learning-based code optimization. This technique takes in a feature vector and predicts which of a set of classes the feature vector is associated with. For example, classification can be used to predict which of a set of unroll factors should be used for a given loop, by taking in a feature vector that describes the characteristics of the target loop (see also Section II-D).

The k-nearest neighbur (KNN) algorithm is a simple yet effective classification technique. It finds the k closet training examples to the input instance (or program) on the feature space. The closeness (or distance) is often evaluated using the Euclidean distance, but other metrics can also be used. This technique has been used to predict the optimal optimization parameters in prior works [52], [66], [67]. It

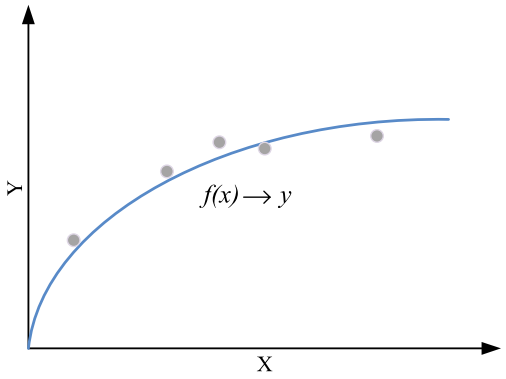

Fig. 5. *A simple regression-based curve-fitting example. There are five training examples in this case. A function f is trained with the training data, which maps the input x to the output y. The trained function can predict the output of an unseen x.*

Table 3 Regression Techniques Used in Prior Works

Modelling Technique	Application	References
Linear Regression	Exec. Time Estimation	[62], [38], [43]
Linear Regression	Perf. & Power Prediction	[63], [64], [65]
Artificial Neural Networks	Exec. Time Estimation	[62], [46], [39]

works by first predicting which of the training programs are closet (i.e., nearest neighbors) to the incoming program on the feature space; it then uses the optimal parameters (which are found during training time) of the nearest neighbors as the prediction output. While it is effective on small problems, KNN also has two main drawbacks. First, it must compute the distance between the input and all training data at each prediction. This can be slow if there is a large number of training programs to be considered. Second, the algorithm itself does not learn from the training data; instead, it simply selects the k nearest neighbors. This means that the algorithm is not robust to noisy training data and could choose an ill-suited training program as the prediction.

As an alternative, the decision tree has been used in prior works for a range of optimization problems. These include choosing the parallel strategy for loop parallelization [69], determining the loop unroll factor [16], [70], deciding the profitability of using GPU acceleration [68], [71], and selecting the optimal algorithm implementation [72]. The advantage of a decision tree is that the learned model is interpretable and can be easily visualized. This enables users to understand why a particular decision is made by following the path from the root node to a leaf decision node. For example, Fig. 6 depicts the decision tree model developed in [68] for selecting the best performing device (CPU or GPU) to run an OpenCL program. To make a prediction, we start from the root of the tree; we compare a feature value (e.g., the communication–computation ratio) of the target program against a threshold to determine which branch of the tree to follow; and we repeat this process until we reach a leaf node where a decision will be made. It is to note that the structure and thresholds of the tree are automatically determined by the machine learning algorithm, which may change when we target a different architecture or application domain.

Decision trees make the assumption that the feature space is convex, i.e., it can be divided up using hyperplanes into different regions, each of which belongs to a different category. This restriction is often appropriate in practice. However, a significant drawback of using a single decision tree is that the model can overfit due to outliers in the training data (see also Section IV-D). Random forests [73] have, therefore, been proposed to alleviate the problem of overfitting. Random forests are an ensemble learning method [74]. As illustrated in Fig. 7, it works by constructing multiple decision trees at training time. The prediction of each tree depends on the values of a random vector sampled independently on the feature value. In this way, each tree is randomly forced to be insensitive to some feature dimensions. To make a prediction, random forests then aggregate the outcomes of individual trees to form an overall prediction. It has been employed to determine whether to inline a function or not [75], delivering better performance than a single-model-based approach. We want to highlight that random forests can also be used for regression tasks. For instances, it has been used to model energy consumption of OpenMP [76] and CUDA [77] programs.

Logical regression is a variation of linear regression but is often used for classification. It takes in the feature vector and calculates the probability of some outcome. For example, Cavazos and O'Boyle used logical regression to determine the optimization level of Jike RVM. Like decision trees, logical regression also assumes that the feature values and the prediction has a linear relation.

More advanced models, such as SVM classification, have been used for various compiler optimization tasks [46], [79]–[81]. SVMs use kernel functions to compute the similarity of feature vectors. The radial basis function (RBF) is commonly used in prior works [46], [82] because it can model both linear and nonlinear problems. It works by mapping the input feature vector to a higher dimensional space where it may be easier to find a linear hyperplane to well separate the labeled data (or classes).

Other machine learning techniques, such as kernel canonical correlation analysis and naive Bayes, have also

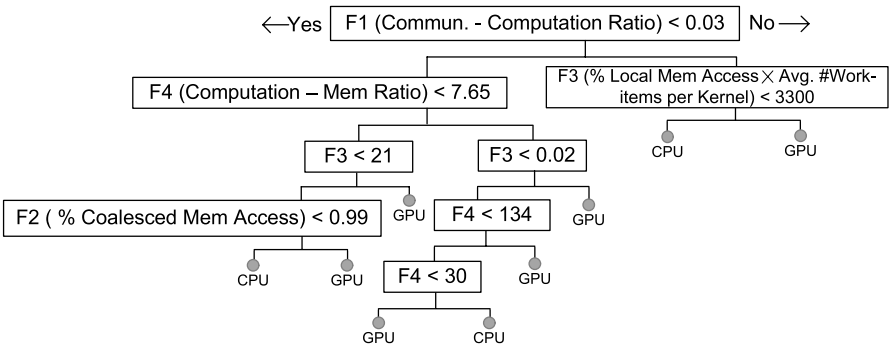

Fig. 6. *A decision tree for determining which device (CPU or GPU) to use to run an OpenCL program. This diagram is reproduced from [68].*

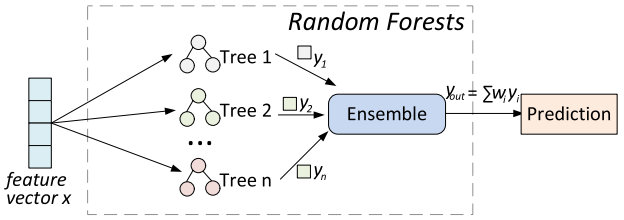

Fig. 7. ***Random forests are an ensemble learning algorithm. It aggregates the outputs of multiple decision trees to form a final prediction. The idea is to combine the predictions from multiple individual models together to make a more robust, accurate prediction than any individual model.***

been used in prior works to predict stencil program configurations [83] or detect parallel patterns [84].

3) Deep Neural Networks: In recent years, deep neural networks [85] have been shown to be a powerful tool for tackling a range of machine learning tasks such as image recognition [86], [87] and audio processing [88]. Deep neural networks (DNNs) have recently been used to model program source code [89] for various software engineering tasks (see also Section VI-C), but so far there is little work of applying DNNs to compiler optimization. A recent attempt in this direction is the DeepTune framework [78], which uses DNNs to extract source code features (see also Section V-C).

The advantage of DNNs is that they can compactly represent a significantly larger set of functions than a shallow network, where each function is specialized at processing part of the input. This capability allows DNNs to model the complex relationship between the input and the output (i.e., the prediction). As an example, consider Fig. 8 that visualizes the internal state of DeepTune [78] when predicting the optimal thread coarsening factor for an OpenCL kernel (see Section II-D). Fig. 8(a) shows the first 80 elements of the input source code tokens as a heatmap in which each cell's color reflects an integer value assigned to a specific token. Fig. 8(b) shows the neurons of the first DNN for each of the four GPU platforms, using a red–blue heatmap to visualize the intensity of each activation. If we have a close look at the heatmap, we can find a number of neurons in the layer with different responses across platforms. This indicates that the DNN is partly specialized to the target platform. As information flows through the network [layers (c) and (d) in Fig. 8], the layers become progressively more specialized to the specific platform.

B. Unsupervised Learning

Unlike supervised learning models which learn a correlation from the input feature values to the corresponding outputs, unsupervised learning models only take it from the input data (e.g., the feature values). This technique is often used to model the underlying structure of distribution of the data.

Clustering is a classical unsupervised learning problem. The k-means clustering algorithm [90] groups the input data into k clusters. For example, in Fig. 9, a k-means algorithm is used to group data points into three clusters on a 2-D

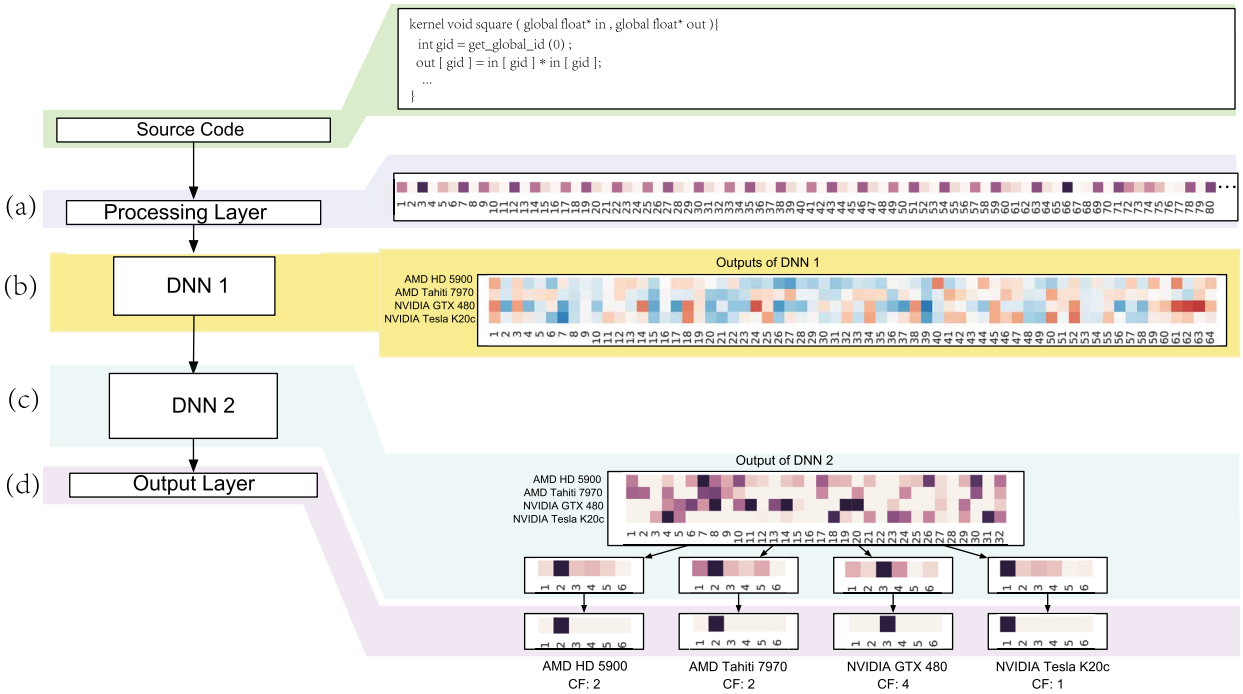

Fig. 8. ***A simplified view of the internal state for the DeepTune DNN framework [78] when it predicts the optimal OpenCL thread coarsening factor. Here, a DNN is learned for each of the four target GPU architectures. The activations in each layer of the four models increasingly diverge (or specialize) toward the lower layers of the model. It is to note that some of the DeepTune layers are omitted to aid presentation.***

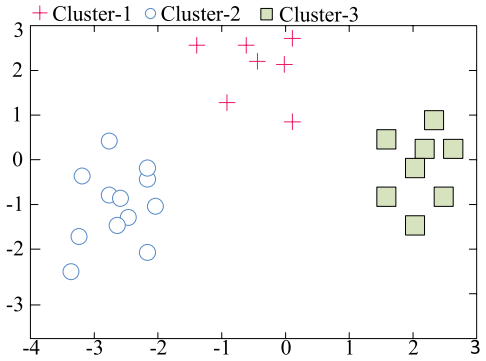

Fig. 9. ***Using k-means to group data points into three clusters. In this example, we group the data points into three clusters on a 2-D feature space.***

feature space. The algorithm works by grouping data points that are close to each other on the feature space into a cluster. K-means is used to characterize program behavior [60], [91]. It does so by clustering program execution into phase groups, so that we can use a few samples of a group to represent the entire program phases within a group. K-means is also used in the work presented in [92] to summarize the code structures of parallel programs that benefit from similar optimization strategies. In addition to k-means, Martins *et al.* employed the fast Newman clustering algorithm [93] which works on network structures to group functions that may benefit from similar compiler optimizations [94].

PCA is a statistical method for unsupervised learning. This method has been heavily used in prior work to reduce the feature dimension [17], [25], [95]–[97]. Doing so allows us to model a high-dimensional feature space with a smaller number of representative variables which, in combination, describe most of the variability found in the original feature space. PCA is often used to discover the common pattern in the data sets in order to help clustering exercises. It is used to select representative programs from a benchmark suite [95], [98]. In Section V-D, we discuss PCA in further details.

Autoencoders are a recently proposed artificial neural network architecture for discovering the efficient codings of input data in an unsupervised fashion [99]. This technique can be used in combination of a natural language model to first extract features from program source code and then find a compact representation of the source code features [100]. We discuss autoencoders in Section V-D when reviewing feature dimensionality reduction techniques.

C. Online Learning

1) Evolutionary Search: Evolutionary algorithms (EAs) or evolutionary computation such as genetic algorithms (GAs), GP,[4] and stochastic-based search have been employed to find a good optimization solution from a large search space. An EA applies principles inspired by biological evolution to find an optimal or near-optimal solution for the target problem. For instance, the SPIRAL autotuning framework uses a stochastic evolutionary search algorithm to choose a fast formula (or transformation) for signal processing applications [101]. Li *et al.* use GAs to search for the optimal configuration to determine which sorting algorithm to use based on the unsorted data size [102]. The Petabricks compiler offers a more general solution by using EAs to search for the best performing configurations for a set of algorithms specified by the programmer [103]. In addition to code optimization, EAs have also been used to create Pareto optimal program benchmarks under various criteria [104].

As an example, consider how an EA can be employed in the context of iterative compilation to find the best compiler flags for a program [25], [36], [105]. Fig. 10 depicts how an EA can be used for this purpose. The algorithm starts from several populations of randomly chosen compiler flag settings. It compiles the program using each individual compiler flag sequence, and uses a fitness function to evaluate how well a compiler flag sequence performs. In our case, a fitness function can simply return the reciprocal of a program runtime measurement, so that compiler settings that give faster execution time will have a higher fitness score. In the next epoch, the EA algorithm generates

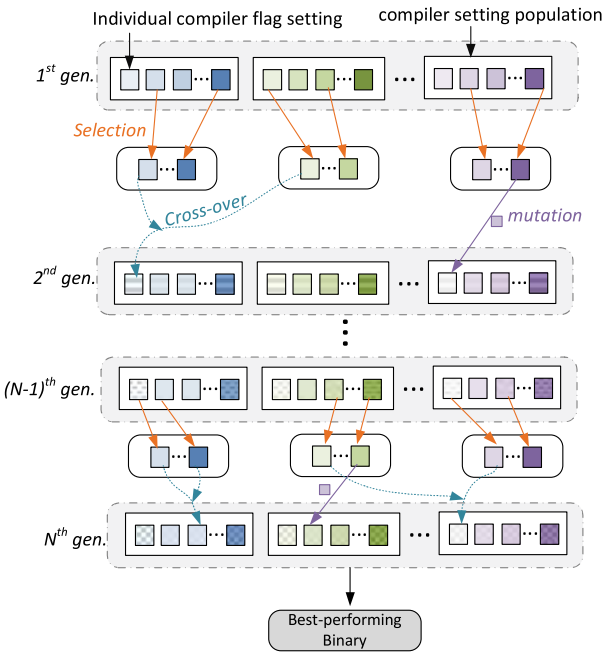

Fig. 10. ***Using an EA to perform iterative compilation. The algorithm starts from several initial populations of randomly chosen compiler flag sequences. It evaluates the performance of individual sequences to remove poorly performing sequences in each population. It then applies crossover and mutation to create a new generation of populations. The algorithm returns the best performing program binary when it terminates.***

[4]A GA is represented as a list of actions and values, often a string, while a GP is represented as a tree structure of actions and values. For example, GP is applied to the abstract syntax tree of a program to search for useful features in [70].

the next populations of compiler settings via mechanisms such as reproduction (crossover) and mutation among compiler flag settings. This results in a new generation of compiler flag settings and the quality of each setting will be evaluated again. In a mechanism analogous to natural selection, a certain number of poorly performing compiler flags within a population are chosen to die in each generation. This process terminates when no further improvement is observed or the maximum number of generations is reached, and the algorithm will return the best found program binary as a result.

There are three key operations in an EA algorithm: selection, crossover, and mutation. The probability of an optimization option being selected for dying is often inversely proportional to its fitness score. In other words, options that are relatively fitter (e.g., give faster program runtime) are more likely to survive and remain a part of the population after selection. In crossover, a certain number of offsprings are produced by mixing some existing optimization options (e.g., compiler flags). The likelihood of an existing option being chosen for crossover is again proportional to its fitness. This strategy ensures that good optimizations will be preserved over generations, while poorly performing optimizations will gradually die out. Finally, mutation randomly changes a preserved optimization, e.g., by turning on/off an option or replacing a threshold value in a compiler flag sequence. Mutation reduces the chance that the algorithm gets stuck with a locally optimal optimization.

EAs are useful for exploring a large optimization space where it is infeasible to just enumerate all possible solutions. This is because an EA can often converge to the most promising area in the optimization space quicker than a general search heuristic. The EA is also shown to be faster than a dynamic-programming-based search [24] in finding the optimal transformation for the fast Fourier transformation (FFT) [101]. When compared to supervised learning, EAs have the advantage of requiring little problem-specific knowledge, and hence they can be applied on a broad range of problems. However, because an EA typically relies on the empirical evidences (e.g., running time) for fitness evaluation, the search time can still be prohibitively expensive. This overhead can be reduced by using a machine-learning-based cost model [43] to estimate the potential gain (e.g., speedup) of a configuration (see also Section III-A). Another approach is to combine supervised learning and EAs [25], [106] by first using an offline learned model to predict the most promising areas of the design space (i.e., to narrow down the search areas), and then searching over the predicted areas to refine the solutions. Moreover, instead of predicting where in the search space to focus on, one can also first prune the search space to reduce the number of options to search over. For example, Jantz and Kulkarni show that the search space of phase ordering[5] can be greatly reduced if we can first remove phases whose application order is irrelevant to the produced code [107]. Their techniques are claimed to prune the exhaustive phase order search space size by 89% on average.

[5]Compiler phase ordering determines at which order a set of compiler optimization passes should be applied to a given program.

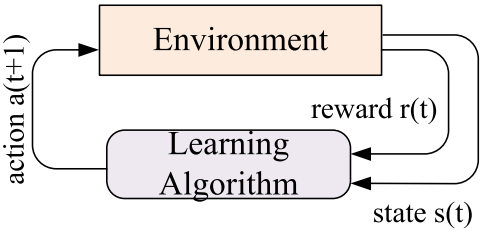

Fig. 11. ***The working mechanism of reinforcement learning.***

2) Reinforcement Learning: Another class of online learning algorithms is reinforcement learning (RL) which is sometimes called "learning from interactions." The algorithm tries to learn how to maximize the rewards (or performance) itself. In other words, the algorithm needs to learn, for a given input, what the correct output or decision to take is. This is different from supervised learning where the correct input/output pairs are presented in the training data.

Fig. 11 illustrates the working mechanism of RL. Here the learning algorithm interacts with its environment over a discrete set of time steps. At each step, the algorithm evaluates the current state of its environment, and executes an action. The action leads to a change in the state of the environment (which the algorithm can evaluate in the next time step), and produces an immediate reward. For example, in a multitasking environment, a state could be the CPU contention; when processor cores are idle, an action could be where to place a process, and a reward could be the overall system throughput. The goal of RL is to maximize the long-term cumulative reward by learning an optimal strategy to map states to actions.

RL is particularly suitable for modeling problems that have an evolving natural, such as dynamic task scheduling, where the optimal outcome is achieved through a series of actions. RL has been used in prior research to schedule RAM memory traffics [108], select software component configurations at runtime [109], and configure virtual machines [110]. An early work of using RL for program optimization was conduced by Lagoudakis and Littman [111]. They use RL to find the cutoff point to switch between two sorting algorithms: `quickSort` and `insertionSort`. CALOREE combines machine learning and control theories to schedule CPU resources on heterogeneous multicores [112]. For a given application, CALOREE uses control-theoretic methods to dynamically adjust the resource allocation, and machine learning to estimate the application's latency and power for a given resource allocation plan (to offer decision supports).

An interesting RL-based approach for scheduling parallel OpenMP programs is presented in [113]. This approach predicts the best number of threads for a target OpenMP

program when it runs with other competing workloads, aiming to make the target program run faster. This approach first learns a reward function offline based on static code features and runtime system information. The reward function is used to estimate the reward of a runtime scheduling action, i.e., the expected speedup when assigning a certain number of processor cores to an OpenMP program. In the next scheduling epoch, this approach uses the empirical observation of the application speedup to check if the reward function was accurate and the decision was good, and update the reward function if the model is found to be inaccurate.

In general, RL is an intuitive and comprehensive solution for autonomous decision making. But its performance depends on the effectiveness of the value function, which estimates the immediate reward. An optimal value function should lead to the greatest cumulative reward in the longer term. For many problems, it is difficult to design an effective value function or policy, because the function needs to foresee the impact of an action in the future. The effectiveness of RL also depends on the environment; if the number of possible actions is large, it can take RL a long time to converge to a good solution. RL also requires the environment to be fully observed, i.e., all the possible states of the environment can be anticipated ahead of time. However, this assumption may not hold in a dynamic computing environment due to unpredictable disturbances, e.g., changes in application inputs or application mixes. In recent years, deep learning techniques have been used in conjunction with RL to learn a value function. The combined technique is able to solve some problems that were deemed impossible in the past [114]. However, how to combine deep learning with RL to solve compilation and code optimization problems remains an open question.

D. Discussion

What model is best is the $64 000 question. The answer is: it depends. More sophisticated techniques may provide greater accuracy but they require large amounts of labeled training data—a real problem in compiler optimization. Techniques such as linear regression and decision trees require less training data compared to more advanced models such as SVMs and ANNs. Simple models typically work well when the prediction problem can be described using a feature vector that has a small number of dimensions, and when the feature vector and the prediction are linearly correlated. More advanced techniques such as SVMs and ANNs can model both linear and nonlinear problems on a higher dimensional feature space, but they often require more training data to learn an effective model. Furthermore, the performance of an SVM and an ANN also highly depends on the hyperparameters used to train the model. The optimal hyperparameter values can be chosen by performing cross validation on the training data. However, how to select parameters to avoid overfitting while achieving a good prediction accuracy remains an outstanding challenge.

Choosing which modeling technique to use is nontrivial. This is because the choice of model depends on a number of factors: the prediction problem (e.g., regression or classification), the set of features to use, the available training examples, the training and prediction overhead, etc. In prior works, the choice of modeling technique largely relied on developer experience and empirical results. Many of the studies in the field of machine-learning-based code optimization do not fully justify the choice of the model, although some do compare the performance of alternate techniques. The OpenTuner framework addresses the problem by employing multiple techniques for program tuning [115]. OpenTuner runs multiple search techniques at the same time. Techniques which perform well will be given more candidate tuning options to examine, while poorly performed algorithms will be given fewer choices or disabled entirely. In this way, OpenTuner can discover which algorithm works best for a given problem during search.

One technique that has seen little investigation is the use of Gaussian processes [116]. Before the recent widespread interest in DNNs, these were a highly popular method in many areas of machine learning [117]. They are particularly powerful when the amount of training data is sparse and expensive to collect. They also automatically give a confidence interval with any decision. This allows the compiler writer to trade off risk versus reward depending on the application scenario.

Using a single model has a significant drawback in practice. This is because a one-size-fits-all model is unlikely to precisely capture behaviors of diverse applications, and no matter how parameterized the model is, it is highly unlikely that a model developed today will always be suited for tomorrow. To allow the model to adapt to the change of the computing environment and workloads, ensemble learning was exploited in prior works [73], [118], [119]. The idea of ensemble learning is to use multiple learning algorithms, where each algorithm is effective for particular problems, to obtain better predictive performance than could be obtained from any of the constituent learning algorithm alone [120], [121]. Making a prediction using an ensemble typically requires more computational time than doing that using a single model, so ensembles can be seen as a way to compensate for poor learning algorithms by performing extra computation. To reduce the overhead, fast algorithms such as decision trees are commonly used in ensemble methods (e.g., random forests), although slower algorithms can benefit from ensemble techniques as well.

V. FEATURE ENGINEERING

Machine-learning-based code optimization relies on having a set of high-quality features that capture the important characteristics of the target program. Given that there is an

Table 4 Summary of Features Discussed in Section V

Feature	Description
Static code features	Features gathered from source code or the compiler intermediate representations, such as instruction counts. See Section V-A1.
Tree and graph based features	Features extracted from the program graph, such as the number of nodes of different types. See Section V-A2.
Dynamic features	Features obtained through dynamic profiling or during runtime execution, such as performance counter values. See Section V-A3.
Reaction-based features	Speedups or execution time obtained by profiling the target program under specific compiler settings. See Section V-B.

unbounded number of potential features, finding the right set is a nontrivial task. In this section, we review how previous work chooses features, a task known as feature engineering. Tables 4 and 5 summarize the range of program features and feature engineering techniques discussed in this section, respectively.

A. Feature Representation

Various forms of program features have been used in compiler-based machine learning. These include static code structures [122] and runtime information such as system load [118], [123] and performance counters [53].

1) Static Code Features: Static program features such as the number and type of instructions are often used to describe a program. These features are typically extracted from the compiler intermediate representations [29], [46], [52], [80] in order to avoid using information extracted from dead code. Table 6 gives some of the static code features that were used in previous studies. Raw code features are often used together to create a combined feature. For example, one can divide the number of load instructions by the number of total instructions to get the memory load ratio. An advantage of using static code features is that the features are readily available from the compiler intermediate representation.

2) Tree- and Graph-Based Features: Singer and Veloso represent the FFT in a split tree [124]. They extract from the tree a set of features, by counting the number of nodes of various types and quantifying the shape of the tree. These tree-based features are then used to build a neural-network-based cost function that predicts which of the two FFT formulas runs faster. The cost function is used to search for the best performing transformation.

Table 5 Feature Engineering Techniques Discussed in Section V.

Problem	Techniques
Feature selection	Pearson correlation coefficient, mutual information, regression analysis. See Section V-D1.
Feature dimensionality reduction	Principal component analysis (PCA), factor analysis, linear discriminant analysis, autoencoder. See Section V-D2.

Table 6 Example Code Features Used in Prior Works

Description	Examples
Arithmetic instructions	#floating point instr., #integer instr., #method call instr.
Memory operations	#load instr, #store instr.
Branch instructions	#conditional branch instr, #unconditional branch instr
loop information	#loops, loop depth
parallel information	#work threads, work group size

Park *et al.* present a unique graph-based approach for feature representations [125]. They use an SVM where the kernel is based on a graph similarity metric. Their technique requires hand-coded features at the basic block level, but thereafter, graph similarity against each of the training programs takes the place of global features. Mailike shows that spatial-based information, i.e., how instructions are distributed within a program, extracted from the program's data flow graph could be a useful feature for machine-learning-based compiler optimization [126]. Nobre *et al.* also exploit graph structures for code generation [26]. Their approach targets the phase ordering problem. The order of compiler optimization passes is represented as a graph. Each node of the graph is an optimization pass and connections between nodes are weighted in a way that subsequences with higher aggregated weights are more likely to lead to faster runtime. The graph is automatically constructed and updated using iterative compilation (where the target program is complied using different compiler passes with different orders). A design space exploration algorithm is employed to drive the iterative compilation process.

3) Dynamic Features: While static code features are useful and can be extracted at static compile time (hence feature extraction has no runtime overhead), they have drawbacks. For examples, static code features may contain information of code segments that rarely get executed, and such information can confuse the machine learning model; some program information such as the loop bound depends on the program input, which can only be obtained during execution time; and static code features often may not precisely capture the application behavior in the runtime environment [such as resource contention and input/output (I/O) behavior] as such behavior highly depends on the computing environment such as the number of available processors and corunning workloads.

As illustrated in Fig. 12, dynamic features can be extracted from multiple layers of the runtime environment. At the application layer, we can obtain information such as loop iteration counts that cannot be decided at compile time, dynamic control flows, frequently executed code regions, etc. At the operating system level, we can observe the memory and I/O behavior of the application as well as CPU load and thread contention, etc. At the hardware

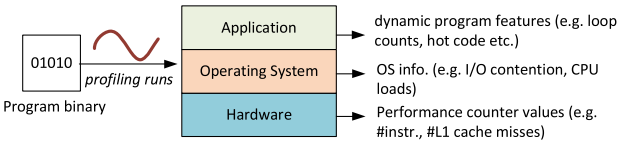

Fig. 12. ***Dynamic features can be extracted from multiple layers of the computing environment.***

level, we can use performance counters to track information such as how many instructions have been executed and of what types, and the number of cache loads/stores as well as branch misses, etc.

Hardware performance counter values, such as executed instruction counts and cache miss rate, are therefore used to understand the application's dynamic behaviors [53], [127], [128]. These counters can capture low-level program information such as data access patterns, branches, and computational instructions. One of the advantages of performance counters is that they capture how the target program behaves on a specific hardware and avoid the irrelevant information that static code features may bring in. In addition to hardware performance counters, operating system level metrics, such as system load and I/O contention, are also used to model an application's behavior [39], [123]. Such information can be externally observed without instrumenting the code, and can be obtain during offline profiling or program execution time.

While effective, collecting dynamic information could incur prohibitively overhead and the collected information can be noisy due to competing workloads and operating system scheduling [129] or even subtle settings of the execution environment [130]. Another drawback of performance counters and dynamic features is that they can only capture the application's past behavior. Therefore, if the application behaves significantly different in the future due to the change of program phases or inputs, then the prediction will be drawn on an unreliable observation. As such, dynamic and static features are often used in combination in prior works in order to build a robust model.

B. Reaction-Based Features

Cavazos *et al.* present a reaction-based predictive model for software–hardware codesign [131]. Their approach profiles the target program using several carefully selected compiler options to see how program runtime changes under these options for a given microarchitecture setting. They then use the program "reactions" to predict the best available application speedup. Fig. 13 illustrates the difference between a reaction-based model and a standard program feature-based model. A similar reaction-based approach is used in [132] to predict speedup and energy efficiency for an application that is parallelized thread-level speculation (TLS) under a given microarchitectural configuration. Note that while a reaction-based approach does not use static

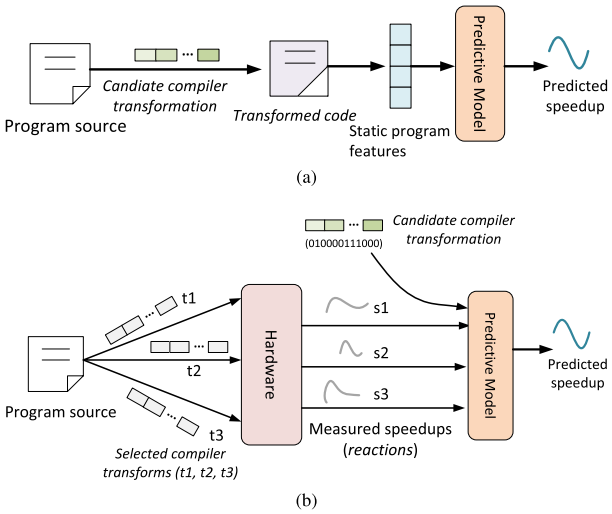

Fig. 13. ***Standard feature-based modeling (a) versus reaction-based modeling (b). Both models try to predict the speedup for a given compiler transformation sequence. The program feature-based predictor takes in static program features extracted from the transformed program, while the reaction-based model takes in the target transformation sequence and the measured speedups of the target program, obtained by applying a number of carefully selected transformation sequences. Diagrams are reproduced from [131].***

code features, developers must carefully select a few settings from a large number of candidate options for profiling, because poorly chosen options can significantly affect the quality of the model.

C. Automatic Feature Generation

As deriving good features is a time-consuming task, a few methods have been proposed to automatically generate features from the compiler's intermediate representation (IR) [70], [133]. The work of [70] uses GP to search for features, but required a huge grammar to be written, some 160 kB in length. Although much of this can be created from templates, selecting the right range of capabilities and search space bias is nontrivial and up to the expert. The work of [133] expresses the space of features via logic programming over relations that represent information from the IRs. It greedily searches for expressions that represent good features. However, their approach relies on expert selected relations, combinators, and constraints to work. Both approaches closely tie the implementation of the predictive model to the compiler IR, which means changes to the IR will require modifications to the model. Furthermore, the time spent in searching features could be significant for these approaches.

The first work to employ neural network to extract features from program source code for compiler optimization was conducted by Cummins *et al.* [78]. Their system, namely DeepTune, automatically abstracts and selects appropriate features from the raw source code. Unlike prior work

where the predictive model takes in a set of human-crafted features, program code is used directly in the training data. Programs are fed through a series of neural-network-based language models which learn how the code correlates with the desired optimization options (see also Fig. 8). Their work also shows that the properties of the raw code that are abstracted by the top layers of the neural networks are mostly independent of the optimization problem. While promising, it is worth mentioning that dynamic information such as the program input size and performance counter values are often essential for characterizing the behavior of the target program. Therefore, DeepTune does not completely remove human involvement for feature engineering when static code features are insufficient for the optimization problem.

D. Feature Selection and Dimension Reduction

Machine learning uses features to capture the essential characteristics of a training example. Sometimes we have too many features. As the number of features increases, so does the number of training examples needed to build an accurate model [134]. Hence, we need to limit the dimension of the feature space. In compiler research, commonly, an initial large, high-dimensional candidate feature space is pruned via feature selection [52], or projected into a lower dimensional space [17]. In this section, we review a number of feature selection and dimension reduction methods.

1) Feature Selection: Feature selection requires understanding how a particular feature affects the prediction accuracy. One of the simplest methods for doing this is applying the Pearson correlation coefficient. This metric measures the linear correlation between two variables and is used in numerous works [55], [92], [122], [135] to filter out redundant features by removing features that have a strong correlation with an already selected feature. It has also been used to quantify the relation of the select features in regression. One obvious drawback of using Pearson correlation as a feature ranking mechanism is that it is only sensitive to a linear relationship.

Another approach for correlation estimation is mutual information [131], [136], which quantifies how much information of one variable (or feature) can be obtained through another variable (feature). Like correlation coefficient, mutual information can be used to remove redundant features. For example, if the information of feature x can be largely obtained through another existing feature y, feature x can then be taken out from the feature set without losing much information on the reduced feature set.

Both correlation coefficient and mutual information evaluate each feature independently with respect to the prediction. A different approach is to utilize regression analysis for feature ranking. The underlying principal of regression analysis is that if the prediction is the outcome of regression model based on the features, then the most important features should have the highest weights (or coefficients) in the model, while features uncorrelated with the output variables should have weights close to zero. For example, least absolute shrinkage and selection operator (LASSO) regression analysis is used in [137] to remove less useful features to build a compiler-based model to predict performance. LASSO has also been used for feature selection to tune the compiler heuristics for the TRIPS processor [138].

In general, feature selection remains an open problem for machine learning, and researchers often follow a "trail-and-error" approach to test a range of methods and feature candidates. This makes automatic feature selection framework like FEAST [139] and HERCULES [140] attractive. The former framework employs a range of existing feature selection methods to select useful candidate features, while the latter searches for the most important static code features from a set of predefined patterns for loops.

2) Feature Dimensionality Reduction: While feature selection allows us to select the most important features, the resulted feature set can still be too large to train a good model, especially when we only have a small number of training examples. By reducing the number of dimensions, the learning algorithm can often perform more efficiently on a limited training data set. Dimension reduction is also important for some machine learning algorithms such as KNN to avoid the effect of the curse of dimensionality [141].

PCA is a well-established feature reduction technique [142]. It uses orthogonal linear transformations to reduce the dimensionality of a set of variables, i.e., features in our case.

Fig. 14 demonstrates the use of PCA to reduce the number of dimensions. The input in this example is a 3-D space defined by M_1, M_2, and M_3, as shown in Fig. 14(a). Three components, PC_1, PC_2, and PC_3, which account for the variance of the data, are first calculated. Here, PC_1 and PC_2 contribute most to the variance of the data and PC_3 accounts for the least variance. Using only PC_1 and PC_2, one can transform the original, 3-D space into a new, 2-D coordinate

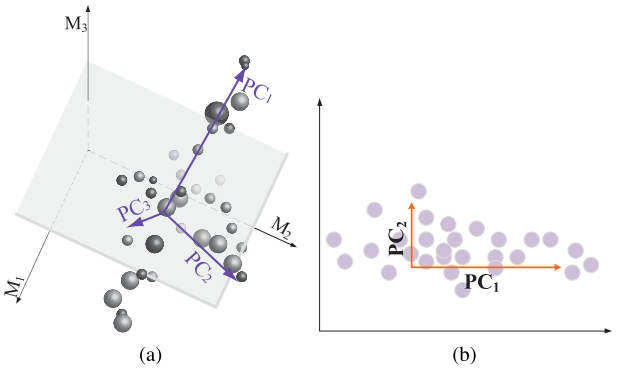

Fig. 14. ***Using PCA to reduce dimensionality of a 3-D feature space. The principal components are first computed (a). Then, the first two principal components (PC1 and PC2) are selected to represent the original 3-D feature space on a new 2-D space (b).***

system [as illustrated in Fig. 14(b)] while preserving much of the variance of the original data.

PCA has been used in many prior compiler research works for feature reduction [17], [25], [55], [92], [95]–[97], [143]. It has also been used in prior works to visualize the working mechanism of a machine learning model, e.g., to show how benchmarks can be grouped in the feature space [123], by projecting features from a high-dimensional space into a 2-D space.

We want to stress that PCA does not select some features and discard the others. Instead, it linearly combines the original features to construct new features that can summarize the list of the original features. PCA is useful when there is some redundancy in the raw features, i.e., some of the features are correlated with one another. Similar feature reduction methods include factor analysis and linear discriminant analysis (LDA), which all try to reduce the number of features by linearly combining multiple raw features. However, PCA seems to be the most popular feature reduction method used in compiler research, probably due to its simplicity.

An alternative way of reducing the number of features used is via an autoencoder [144]. It is a neural network that finds a representation (encoding) for a set of data, by dimensionality reduction. Autoencoders works by learning an encoder and a decoder from the input data. The encoder tries to compress the original input into a low-dimensional representation, while the decoder tries to reconstruct the original input based on the low-dimension representations generated by the encoder. As a result, the autoencoder has been widely used to remove the data noise as well as to reduce the data dimension [145].

Autoencoders have been applied to various natural language processing tasks [99], often being used together with DNNs. Recently, it has been employed to model program source code to obtain a compact set of features that can characterize the input program source [78], [146]–[149].

VI. SCOPE

Machine learning has been used to solve a wide range of problems, from the early successful work of selecting compiler flags for sequential programs, to recent works on scheduling and optimizing parallel programs on heterogeneous multicores. In this section, we review the types of problems that have been exploited in prior works.

A. Optimizing Sequential Programs

Early works for machine learning in compilers look at how, or if, a compiler optimization should be applied to a sequential program. Some of the previous studies build supervised classifiers to predict the optimal loop unroll factor [52], [70] or to determine whether a function should be inlined [29], [35]. These works target a fixed set of compiler options, by representing the optimization problem as a multiclass classification problem, where each compiler option is a class. For example, Leather *et al.* [70] considered a loop unroll factor between 0 and 15 (16 configurations in total), treating each candidate unroll factor as a class; they compiled and profiled each training program by trying all 16 configurations to find out the best loop unroll factor for each program, and then learned a decision tree model from the training data.

There are other compiler problems where the number of possible options is massive. For instance, the work presented in [55] considers 54 code transformations of GCC. While these options are only a subset from the over hundreds of transformations provided by GCC, the resulted combinatorial compiler configurations lead to a space of approximately 10^{34}. Although it is possible to build a classifier to directly predict the optimal setting from a large space, to learn an effective model would require a large volume of training programs in order to have an adequate sampling over the space. Doing so is difficult because 1) there are only a few dozen common benchmarks available; and 2) compiler developers need to generate the training data themselves.

EAs such as generic search are often used to explore a large design space (see also Section IV-C1). Prior works have used EAs to solve the phase ordering problem (i.e., at which order a set of compiler transformations should be applied) [150]–[152], determining the compiler flags during iterative compilation [153]–[156], selecting loop transformations [157], tuning algorithmic choices [11], [103], etc.

B. Optimizing Parallel Programs

How to effectively optimize parallel programs has received significant attentions in the past decade, largely because the hardware industry has adopted multicore design to avoid the power wall [158]. While multicore and many-core architectures provide the potential for high-performance and energy-efficient computing, the potential performance can only be unlocked if the application programs are suitably parallel and can be made to match the underlying heterogeneous platform. Without this, the myriad cores on multicore processors and their specialized processing elements will sit idle or poorly utilized. To this end, researchers have extended the reach of machine learning to optimize parallel programs.

A line of research in parallel program optimization is parallelism mapping. That is, given an already parallelized program, how to map the application parallelism to match the underlying hardware to make the program run as fast as possible or be as energy efficient as possible. Zhang *et al.* developed a decision-tree-based approach to predict the scheduling policy to use for an OpenMP parallel region [159]. The work presented in [46] employs two machine learning techniques to predict the optimal number of threads as well as the scheduling policy to use for OpenMP

parallel loop. Specifically, it uses a regression-based ANN model to predict the speedup of a parallel loop when it runs with a given number of threads (to search for the optimal number threads), and an SVM classifier to predict the scheduling policy. There are also works that use machine learning to determine the optimum degree of parallelism for transactional memory [160] and hardware source allocation [161], or to select a code version from a pool of choices to use [162]. Castro *et al.* developed a decision tree classifier to predict the thread mapping strategy in the context of software transactional memory [163]. Jung *et al.* constructed an ANN-based predictor to select an effective data structure on a specific microarchitecture [164].

The work presented in [92] and [165] is a unique approach for applying machine learning to map complex parallel programs with unbounded parallel graph structures. The work considers the question of finding the optimal graph structure of a streaming program. The idea was that rather than trying to predict a sequence of transformations over an unbounded graph, where legality and consistency is a real problem, we should consider the problem from the dual feature space. The work showed that it is possible to predict the best target feature (i.e., the characteristics that an ideal transformed program should have) which then can be used to evaluate the worth of candidate transformed graphs (without compiling and profiling the resulted graphs) in the original feature space.

The Petabricks project [103], [166], [167] takes an evolutionary approach for program tuning. The Petabricks compiler employs genetic search algorithms to tune algorithmic choices. Due to the expensive overhead of the search, much of autotuning is done at static compile time. Their work shows that one can utilize the idle processors on a multicore systems to perform online tuning [168], where half of the cores are devoted to a known safe program configuration, while the other half are used for an experimental program configuration. In this way, when the results of the faster configuration are returned, the slower version will be terminated.

The idea of combining compile-time knowledge and runtime information to achieve better optimizations has been exploited by the ADAPT compiler [169]. Using the ADAPT compiler, users describe what optimizations are available and provide heuristics for applying these optimizations. The compiler then reads these descriptions and generates application-specific runtime systems to apply the heuristics. Runtime code tuning is also exploited by Active Harmony [170], which utilizes the computing resources in HPC systems to evaluate different code variants on different nodes to find the best performing version.

There is also an extensive body of work on how to optimize programs on heterogeneous multicore systems. One of the problems for heterogeneous multicore optimization is to determine when and how to use the heterogeneous processors. Researchers have used machine learning to build classifiers to determine which processor to use [68] and at which clock frequency the processor should operate [80], [171]. Others used regression techniques to build curve fitting models to search for the sweat spot for work partitioning among processors [38] or a tradeoff of energy and performance [172].

Another line of research combines compiler-based analysis and machine learning to optimize programs in the presence of competing workloads. This research problem is important because programs rarely run in isolation and must share the computing resources with other corunning workloads. In [173] and [174], an ANN model based on static code features and runtime information was built to predict the number of threads to use for a target program when it runs with external workloads. Later, in [118], an ensemble-learning-based approach was used, which leads to significantly better performance over [173]. In [118], several models are first trained offline; and then one of the model is selected at runtime, taking into consideration the competing workloads and available hardware resources. The central idea is that instead of using a single monolithic model, we can use multiple models where each model is specialized for modeling a subset of applications or a particular runtime scenario. Using this approach, a model is used when its predictions are effective.

Some recent works developed machine learning models based on static code features and dynamic runtime information to schedule OpenCL programs in the presence of GPU contention. The work presented in [175] uses SVM classification to predict the work partition ratio between the CPU and GPU when multiple programs are competing to run on a single GPU. The work described in [39] aims to improve the overall system throughput when there are multiple OpenCL programs competing to run on the GPU. They developed an ANN model to predict the potential speedup for running an OpenCL kernel on the GPU. The speedup prediction is then used as a proxy to determine which of the waiting OpenCL tasks get to run on the GPU and in what order.

The approaches presented in [176] and [177] target task colocation in a data center environment. They use compiler-based code transformations to reduce the contention for multiple corunning tasks. A linear regression model was employed to calculate the contention score of code regions based on performance counter values. Then, a set of compiler-based code transformations is applied to reduce the resource demands of highly contentious code.

C. Other Research Problems

Many works have demonstrated that machine learning is a powerful technique in performance and cost modeling [47], [178]–[180], and in task and resource scheduling [161], [181]–[183]. We envision that many of these techniques can be used to provide evidence to support runtime program optimizations through, e.g., just-in-time compilation.

While not directly target code optimization, compiler-based code analysis and machine learning techniques have been used in conjunction to solve various software engineering tasks. These include detecting code similarities [184], [185], automatic comment generation [186], mining API usage patterns [187], [188], predicting program properties [189], code de-obfuscation for malware detection [190], etc. It is worth mentioning that many of these recent works show that the past development knowledge extracted from large code bases such as GitHub are valuable for learning an effective model. There were two recent studies performed by Cummins *et al.*, which mine Github to synthesize OpenCL benchmarks [148] and code extract features from source code [78]. Both studies demonstrate the usefulness of large code bases and deep learning techniques for learning predictive models for compiler optimizations. We envision that the rich information in large open source code bases could provide a powerful knowledge base for training machine learning models to solve compiler optimization problems, and deep learning could be used as an effective tool to extract such knowledge from massive program source code.

VII. DISCUSSION

One of the real benefits of machine-learning-based approaches is that it forces an empirically driven approach to compiler construction. New models have to be based on empirical data which can then be verified by independent experimentation. This experiment, hypothesis, test cycle is well known in the physical sciences but is a relatively new addition compiler construction.

As machine-learning-based techniques require a sampling of the optimization space for training data, we typically know the best optimization for any program in the training set. If we exclude this benchmark from training, we therefore have access to an upper bound on performance or oracle for this program. This immediately lets us know how good existing techniques are. If they are 50% of this optimum or 95% of this optimum, this immediately tells us whether the problem is worth exploring.

Furthermore, we can construct naive techniques, e.g., a random optimization, and see its performance. If it performed a number of times, it will have an expected value of the mean of the optimization speedups. We can then demand that any new heuristic should outperform this, though in our experience there have been cases where state-of-the-art work was actually less than random.

A. Not a Panacea

This paper has, by and large, been very upbeat about the use of machine learning. However, there are a number of hurdles to overcome to make it a practical reality and this opens up new questions about optimization.

Training cost is an issue that many find alarming. In practice, the cost is much less than a compiler writer, and techniques such as active learning can be employed to reduce overhead of training data generation [191]–[194]. Although its true to say that generating many differently compiled programs and executing and timing them are entirely automatic, finding the right data requires careful consideration. If the optimizations explored have little positive performance on the programs, then there is nothing worth learning.

The most immediate problem continues to be gathering enough sufficient high quality training data. Although there are numerous benchmark sites publicly available, the number of programs available is relatively sparse compared to the number that a typical compiler will encounter in its lifetime. This is particularly true in specialist domains where there may not be any public benchmarks. Automatic benchmark generation work will help here, but existing approaches do not guarantee that the generated benchmarks effectively represent the design space. Therefore, the larger issue of the structure of the program space remains.

A really fundamental problem is that if we build our optimization models based purely on empirical data, then we must guarantee that these data are correct and representative; we must learn the signal, not the noise. Peer review of a machine learning approach is difficult. Black box modeling prevents the quality of the model from being questioned unlike handcrafted heuristics. In a sense, reviewers now have to scrutinize that the experiments were fairly done. This means all training and test data must be publicly available for scrutiny. This is common practice in other empirical sciences. The artefact evaluation committee is an example of this [195], [196].

Although the ability to automatically learn how to best optimize an application and adapt to change is a big step forward, machine learning can only learn from what is provided by the compiler writer. Machine learning can neither invent new program transformations to apply nor derive analysis that determines whether a transformation is legal; all of this is beyond its scope.

B. Will This Put Compiler Writers Out of a Job?

In fact, machine-learning-based compilation will paradoxically lead to a renaissance in compiler optimization. Compilers have become so complex that adding a new optimization or compiler phase can lead to performance regressions. This, in turn, has led to a conservative mind set where new transformations are not considered if they may rock the boat. The core issue is that systems are so complex that it is impossible to know for sure when to use such an optimization. Machine learning can remove this uncertainty by automatically determining when an optimization is profitable. This now frees the compiler writer to develop ever more sophisticated techniques. He/she does not need to worry about how they interfere with other optimizations—machine learning looks after this. We can now develop optimizations that will typically only work for specific domains,

and not worry about coordinating their integration into a general purpose system. It allows different communities to develop novel optimizations and naturally integrate them. So rather than closing down the opportunity for new ideas, it opens up new vistas.

C. Open Research Directions

Machine learning has demonstrated its utility as a means of automating compiler profitability analysis. It will continue to be used for more complex optimization problems and is likely to be the default approach to selecting compiler optimizations in the coming decade.

The open research directions go beyond predicting the best optimizations to apply. One central issue is what the program space looks like. We know that programs with linear array accesses inside perfect loop nests need different treatment compared to, say, distributed graph processing programs. If we could have a map that allows us to measure distances between programs, then we could see whether there are regions that are well served by compiler characterization and other regions that are sparse and currently ignored. If we could do the same for hardware, then we may be better able to design hardware likely to be of use for emerging applications.

Can machine learning also be applied to compiler analysis? For instance is it possible to learn dataflow or point-to analysis? As deep learning has the ability to automatically construct features, can we find a set of features that are common across all optimizations and analyses? Can we learn the ideal compiler intermediate representation? There is a wide range of interesting research questions that remain unexplored.

VIII. CONCLUSION

This paper has introduced machine-learning-based compilation and described its power in determining an evidence-based approach to compiler optimization. It is the latest stage in 50 years of compiler automation. Machine-learning-based compilation is now a mainstream compiler research area and, over the last decade or so, has generated a large amount of academic interest and papers. While it is impossible to provide a definitive cataloger of all research, we have tried to provide a comprehensive and accessible survey of the main research areas and future directions. Machine learning is not a panacea. It can only learn the data we provide. Rather than, as some fear, it dumbs down the role of compiler writers, it opens up the possibility of much greater creativity and new research areas. ■

REFERENCES

[1] J. Chipps, M. Koschmann, S. Orgel, A. Perlis, and J. Smith, "A mathematical language compiler," in *Proc. 11th ACM Nat. Meeting*, 1956, pp. 114–117.

[2] P. B. Sheridan, "The arithmetic translator-compiler of the IBM FORTRAN automatic coding system," *Commun. ACM*, vol. 2, no. 2, pp. 9–21, 1959.

[3] M. D. McIlroy, "Macro instruction extensions of compiler languages," *Commun. ACM*, vol. 3, no. 4, pp. 214–220, 1960.

[4] A. Gauci, K. Z. Adami, and J. Abela. (2010). "Machine learning for galaxy morphology classification." [Online]. Available: https://arxiv.org/abs/1005.0390

[5] H. Schoen, D. Gayo-Avello, P. T. Metaxas, E. Mustafaraj, M. Strohmaier, and P. Gloor, "The power of prediction with social media," *Internet Res.*, vol. 23, no. 5, pp. 528–543, 2013.

[6] Slashdot. (2009). *IBM Releases Open Source Machine Learning Compiler*. [Online]. Available: https://tech.slashdot.org/story/09/07/03/0143233/ibm-releases-open-source-machine-learning-compiler

[7] H. Massalin, "Superoptimizer: A look at the smallest program," *ACM SIGPLAN Notices*, vol. 22, no. 10, pp. 122–126, 1987.

[8] J. Ivory, "I. On the method of the least squares," *Philos. Mag. J., Comprehending Various Branches Sci., Liberal Fine Arts, Agriculture, Manuf. Commerce*, vol. 65, no. 321, pp. 3–10, 1825.

[9] R. J. Adcock, "A problem in least squares," *Analyst*, vol. 5, no. 2, pp. 53–54, 1878.

[10] K. Datta *et al.*, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 2008, pp. 1–12.

[11] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, "Language and compiler support for auto-tuning variable-accuracy algorithms," in *Proc. Int. Symp. Code Generat. Optim. (CGO)*, Apr. 2011, pp. 85–96.

[12] J. Kurzak, H. Anzt, M. Gates, and J. Dongarra, "Implementation and tuning of batched Cholesky factorization and solve for NVIDIA GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 7, pp. 2036–2048, Jul. 2016.

[13] Y. M. Tsai, P. Luszczek, J. Kurzak, and J. Dongarra, "Performance-portable autotuning of OpenCL kernels for convolutional layers of deep neural networks," in *Proc. Workshop Mach. Learn. HPC Environ. (MLHPC)*, 2016, pp. 9–18.

[14] M. E. Lesk and E. Schmidt, "Lex—A lexical analyzer generator," Tech. Rep., 1975.

[15] S. C. Johnson, *Yacc: Yet Another Compiler-Compiler*, vol. 32. Murray Hill, NJ, USA: Bell Laboratories, 1975.

[16] A. Monsifrot, F. Bodin, and R. Quiniou, "A machine learning approach to automatic production of compiler heuristics," in *Proc. Int. Conf. Artif. Intell. Methodol. Syst. Appl.*, 2002, pp. 41–50.

[17] A. Magni, C. Dubach, and M. O'Boyle, "Automatic optimization of thread-coarsening for graphics processors," in *Proc. 23rd Int. Conf. Parallel Archit. Compilation (PACT)*, 2014, pp. 455–466.

[18] S. Unkule, C. Shaltz, and A. Qasem, "Automatic restructuring of GPU kernels for exploiting inter-thread data locality," in *Proc. 21st Int. Conf. Compil. Construction (CC)*, 2012, pp. 21–40.

[19] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proc. ACM/IEEE Conf. Supercomput. (SC)*, Nov. 2008, pp. 1–11.

[20] Y. Yang, P. Xiang, J. Kong, M. Mantor, and H. Zhou, "A unified optimizing compiler framework for different gpgpu architectures," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 2, p. 9, 2012.

[21] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generat. Optim. (CGO)*, 2004, pp. 75–86.

[22] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou, "Iterative compilation in a non-linear optimization space," in *Proc. Workshop Profile Feedback-Directed Compilation*, 1998.

[23] P. M. Knijnenburg, T. Kisuki, and M. F. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation," *J. Supercomput.*, vol. 24, no. 1, pp. 43–67, 2003.

[24] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005.

[25] F. Agakov *et al.*, "Using machine learning to focus iterative optimization," in *Proc. Int. Symp. Code Generat. Optim. (CGO)*, 2006, pp. 295–305.

[26] R. Nobre, L. G. A. Martins, and J. M. P. Cardoso, "A graph-based iterative compiler pass selection and phase ordering approach," in *Proc. 17th ACM SIGPLAN/SIGBED Conf. Lang. Compil. Tools Theory Embedded Syst. (LCTES)*, 2016, pp. 21–30.

[27] R. Leupers and P. Marwedel, "Function inlining under code size constraints for embedded processors," in *Dig. Tech. Papers IEEE/ACM Int. Conf. Comput.-Aided Design*, Nov. 1999, pp. 253–256.

[28] K. D. Cooper, T. J. Harvey, and T. Waterman, "An adaptive strategy for inline substitution," in *Proc. Joint Eur. Conf. Theory Pract. Softw.*

17th Int. Conf. Compil. Construction (CC/ETAPS), 2008, pp. 69–84.

[29] D. Simon, J. Cavazos, C. Wimmer, and S. Kulkarni, "Automatic construction of inlining heuristics using machine learning," in *Proc. IEEE/ACM Int. Symp. Code Generat. Optim. (CGO)*, 2013, pp. 1–12.

[30] P. Zhao and J. N. Amaral, "To inline or not to inline? Enhanced inlining decisions," *Lang. Compil. Parallel Comput.*, pp. 405–419, 2004.

[31] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison, "Accurate static estimators for program optimization," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 1994, pp. 85–96.

[32] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, Nov. 1994, pp. 384–390.

[33] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proc. ACM SIGPLAN Workshop Lang. Compil. Tools Embedded Syst. (LCTES)*, 1999, pp. 1–9.

[34] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: Improving compiler heuristics with machine learning," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2003, pp. 77–90.

[35] J. Cavazos and M. F. P. O'Boyle, "Automatic tuning of inlining heuristics," in *Proc. ACM/IEEE Conf. Supercomput. (SC)*, Nov. 2005, p. 14.

[36] K. Hoste and L. Eeckhout, "Cole: Compiler optimization level exploration," in *Proc. 6th Annu. IEEE/ACM Int. Symp. Code Generat. Optim. (CGO)*, 2008, pp. 165–174.

[37] M. Kim, T. Hiroyasu, M. Miki, and S. Watanabe, "SPEA2+: Improving the performance of the Strength Pareto Evolutionary Algorithm 2," in *Proc. Int. Conf. Parallel Problem Solving from Nature*, 2004, pp. 742–751.

[38] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Dec. 2009, pp. 45–55.

[39] Y. Wen, Z. Wang, and M. F. P. O'Boyle, "Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms," in *Proc. 21st Ann. IEEE Int. Conf. High Perform. Comput. (HiPC)*, Dec. 2014, pp. 1–10.

[40] E. A. Brewer, "High-level optimization via automated statistical modeling," in *Proc. 5th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPOPP)*, 1995, pp. 80–91.

[41] K. Vaswani, M. J. Thazhuthaveetil, Y. N. Srikant, and P. J. Joseph, "Microarchitecture sensitive empirical models for compiler optimizations," in *Proc. Int. Symp. Code Generat. Optim. (CGO)*, Mar. 2007, pp. 131–143.

[42] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proc. 12th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2006, pp. 185–194.

[43] E. Park, L.-N. Pouche, J. Cavazos, A. Cohen, and P. Sadayappan, "Predictive modeling in a polyhedral optimization space," in *Proc. 9th Annu. IEEE/ACM Int. Symp. Code Generat. Optim. (CGO)*, Apr. 2011, pp. 119–129.

[44] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Prediction models for multi-dimensional power-performance optimization on many cores," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2008, pp. 250–259.

[45] K. Singhet *et al.*, "Comparing scalability prediction strategies on an SMP of CMPs," in *Proc. Eur. Conf. Paralell Process.*, 2010, pp. 143–155.

[46] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: A machine learning based approach," in *Proc. 14th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, 2009, pp. 75–84.

[47] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2017, pp. 615–629.

[48] L. Benini, A. Bogliolo, M. Favalli, and G. De Micheli, "Regression models for behavioral power estimation," *Integr. Comput.-Aided Eng.*, vol. 5, no. 2, pp. 95–106, 1998.

[49] S. K. Rethinagiri, R. B. Atitallah, and J.-L. Dekeyser, "A system level power consumption estimation for MPSoC," in *Proc. Int. Symp. Syst. Chip (SoC)*, 2011, pp. 56–61.

[50] S. Schürmans, G. Onnebrink, R. Leupers, G. Leupers, and X. Chen, "Frequency-aware ESL power estimation for ARM cortex-A9 using a black box processor model," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 1, p. 26, 2016.

[51] M. Curtis-Maury *et al.*, "Identifying energy-efficient concurrency levels using machine learning," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2007, pp. 488–495.

[52] M. Stephenson and S. Amarasinghe, "Predicting unroll factors using supervised classification," in *Proc. Int. Symp. Code Generat. Optim. (CGO)*, 2005, pp. 123–134.

[53] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *Proc. Int. Symp. Code Generat. Optim. (CGO)*, 2007, pp. 185–197.

[54] J. Cavazos and M. F. P. O'Boyle, "Method-specific dynamic compilation using logistic regression," in *Proc. 21st Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst. Lang. Appl. (OOPSLA)*, 2006, pp. 229–240.

[55] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam, "Fast compiler optimization evaluation using code-feature based performance prediction," in *Proc. 4th Int. Conf. Comput. Frontiers (CF)*, 2007, pp. 131–142.

[56] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. E. Eichenberger, and K. O'Brien, "Automatic creation of tile size selection models," in *Proc. 8th Annu. IEEE/ACM Int. Symp. Code Generat. Optim. (CGO)*, 2010, pp. 190–199.

[57] A. M. Malik, "Optimal tile size selection problem using machine learning," in *Proc. Optim. Tile Size Selection Problem Using Mach. Learn.*, vol. 2. Dec. 2012, pp. 275–280.

[58] R. W. Moore and B. R. Childers, "Building and using application utility models to dynamically choose thread counts," *J. Supercomput.*, vol. 68, no. 3, pp. 1184–1213, 2014.

[59] Y. Liu, E. Z. Zhang, and X. Shen, "A cross-input adaptive framework for GPU program optimizations," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, May 2009, pp. 1–10.

[60] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst.*, 2003, pp. 318–319.

[61] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference," in *Proc. 43rd Int. Symp. Comput. Archit. (ISCA)*, 2016, pp. 456–468.

[62] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of inference and learning for performance modeling of parallel applications," in *Proc. 12th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, 2007, pp. 249–258.

[63] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, "Online power-performance adaptation of multithreaded programs using hardware event-based prediction," in *Proc. 20th Annu. Int. Conf. Supercomput. (ICS)*, 2006, pp. 157–166.

[64] P. E. Bailey, D. K. Lowenthal, V. Ravi, B. Rountree, M. Schulz, and B. R. de Supinski, "Adaptive configuration selection for power-constrained heterogeneous systems," in *Proc. 43rd Int. Conf. Parallel Process.*, 2014, pp. 371–380.

[65] J. L. Berral *et al.*, "Towards energy-aware scheduling in data centers using machine learning," in *Proc. 1st Int. Conf. Energy-Efficient Comput. Netw. (e-Energy)*, 2010, pp. 215–224.

[66] D. D. Vento, "Performance optimization on a supercomputer with ctuning and the PGI compiler," in *Proc. 2nd Int. Workshop Adapt. Self-Tuning Comput. Syst. Exaflop Era (EXADAPT)*, 2012, pp. 12–20.

[67] P.-J. Micolet, A. Smith, and C. Dubach, "A machine learning approach to mapping streaming workloads to dynamic multicore processors," *ACM SIGPLAN Notices*, vol. 51, no. 5, pp. 113–122, 2016.

[68] D. Grewe, Z. Wang, and M. F. P. O'Boyle, "Portable mapping of data parallel programs to OpenCL for heterogeneous systems," in *Proc. Proc. IEEE/ACM Int. Symp. Code Generation Optim. (CGO)*, Feb. 2013, pp. 1–10.

[69] H. Yu and L. Rauchwerger, "Adaptive reduction parallelization techniques," in *Proc. 14th Int. Conf. Supercomput. (ICS)*, 2000, pp. 66–77.

[70] H. Leather, E. Bonilla, and M. O'Boyle, "Automatic feature generation for machine learning based optimizing compilation," in *Proc. 7th Annu. IEEE/ACM Int. Symp. Code Generat. Optim. (CGO)*, Mar. 2009, pp. 81–91.

[71] Z. Wang, D. Grewe, and M. F. P. O'Boyle, "Automatic and portable mapping of data parallel programs to opencl for GPU-based heterogeneous systems," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, p. 42, 2014.

[72] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2015, pp. 379–390.

[73] T. K. Ho, "Random decision forests," in *Proc. 3rd Int. Conf. Document Anal. Recognit. (ICDAR)*, vol. 1. 1995, pp. 278–282.

[74] T. G. Dietterich, "Ensemble methods in machine learning," in *Proc. 1st Int. Workshop Multiple Classifier Syst. (MCS)*, 2000, pp. 1–15.

[75] P. Lokuciejewski, F. Gedikli, P. Marwedel, and K. Morik, "Automatic WCET reduction by machine learning based heuristics for function inlining," in *Proc. 3rd Workshop Statistical Mach. Learn. Approaches Archit. Compilation (SMART)*, 2009, pp. 1–15.

[76] S. Benedict, R. S. Rejitha, P. Gschwandtner, R. Prodan, and T. Fahringer, "Energy prediction of OpenMP applications using random forest modeling approach," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshop*, May 2015, pp. 1251–1260.

[77] R. S. Rejitha, S. Benedict, S. A. Alex, and S. Infanto, "Energy prediction of CUDA application instances using dynamic regression models," *Computing*, vol. 99, no. 8, pp. 765–790, 2017.

[78] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-end deep learning of optimization heuristics," in *Proc. 26th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2017, pp. 219–232.

[79] Z. Wang, G. Tournavitis, B. Franke, and M. F. P. O'Boyle, "Integrating profile-driven parallelism detection and machine-learning-based mapping," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 1, p. 2, 2014.

[80] B. Taylor, V. S. Marco, and Z. Wang, "Adaptive optimization for OpenCL programs on embedded heterogeneous systems," in *Proc. 18th Annu. ACM SIGPLAN/SIGBED Conf. Lang. Compil. Tools Embedded Syst. (LCETS)*, 2017, pp. 11–20.

[81] P. Zhang, J. Fang, T. Tang, C. Yang, and Z. Wang, "Auto-tuning streamed applications on Intel Xeon Phi," in *Proc. 32nd IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2018.

[82] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "A predictive performance model for superscalar processors," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Dec. 2006, pp. 161–170.

[83] A. Ganapathi, K. Datta, A. Fox, and D. Patterson, "A case for machine learning to optimize multicore performance," in *Proc. 1st USENIX Conf. Hot Topics Parallelism (HotPar)*, 2009, p. 1.

[84] E. Deniz and A. Sen, "Using machine learning techniques to detect parallel patterns of multi-threaded applications," *Int. J. Parallel Program.*, vol. 44, no. 4, pp. 867–900, 2016.

[85] Y. LeCun, Y. Bengio, and G. Hinton, *Deep Learning*, 2015.

[86] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1097–1105.

[87] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[88] H. Lee, Y. Largman, P. Pham, and A. Y. Ng, "Unsupervised feature learning for audio classification using convolutional deep belief networks," in *Proc. 22nd Int. Conf. Neural Inf. Process. Syst. (NIPS)*, 2009, pp. 1096–1104.

[89] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton (2017). "A survey of machine learning for big code and naturalness." [Online]. Available: https://arxiv.org/abs/1709.06182

[90] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. 5th Berkeley Symp. Math. Statist. Prob.*, 1967, pp. 281–297.

[91] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. 10th Int. Conf. Archit. Support Program. Lang. Operat. Syst. (ASPLOS)*, 2002, pp. 45–57.

[92] Z. Wang and M. F. O'Boyle, "Partitioning streaming parallelism for multi-cores: A machine learning based approach," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2010, pp. 307–318.

[93] M. Newman, *Networks: An Introduction*. New York, NY, USA: Oxford Univ. Press, 2010.

[94] L. G. Martins, R. Nobre, A. C. B. Delbem, E. Marques, and J. A. M. Cardoso, "Exploration of compiler optimization sequences using clustering-based selection," in *Proc. SIGPLAN/SIGBED Conf. Lang. Compil. Tools Embedded Syst. (LCTES)*, 2014, pp. 63–72.

[95] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere, "Workload design: Selecting representative program-input pairs," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2002, pp. 83–94.

[96] Y. Chen *et al.*, "Evaluating iterative optimization across 1000 datasets," in *Proc. 31st ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2010, pp. 448–459.

[97] A. H. Ashouri, G. Mariani, G. Palermo, and C. Silvano, "A Bayesian network approach for compiler auto-tuning for embedded processors," in *Proc. IEEE 12th Symp. Embedded Syst. Real-time Multimedia (ESTIMedia)*, Oct. 2014, pp. 90–97.

[98] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the spec cpu2006 benchmark suite," in *Proc. 34th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2007, pp. 412–423.

[99] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proc. 25th Int. Conf. Mach. Learn. (ICML)*, 2008, pp. 1096–1103.

[100] X. Gu, H. Zhang, D. Zhang, and S. Kim (2016). *Deep API learning*. [Online]. Available: https://arxiv.org/abs/1605.08535

[101] B. Singer and M. Veloso, "Learning to construct fast signal processing implementations," *J. Mach. Learn. Res.*, vol. 3, pp. 887–919, Dec. 2002.

[102] X. Li, M. J. Garzaran, and D. Padua, "Optimizing sorting with genetic algorithms," in *Proc. Int. Symp. Code Generat. Optim. (CGO)*, 2005, pp. 99–110.

[103] J. Ansel *et al.*, "Petabricks: A language and compiler for algorithmic choice," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2009, pp. 38–49.

[104] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark, "The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs (keynote paper)," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, Sep. 2012, pp. 1–14.

[105] U. Garciarena and R. Santana, "Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions," in *Proc. Genetic Evol. Comput. Conf. Companion (GECCO)*, 2016, pp. 1159–1166.

[106] M. Zuluaga, E. Bonilla, and N. Topham, "Predicting best design trade-offs: A case study in processor customization," in *Proc. Design Autom. Test Eur. Conf. Exhibit. (DATE)*, 2012, pp. 1030–1035.

[107] M. R. Jantz and P. A. Kulkarni, "Exploiting phase inter-dependencies for faster iterative compiler optimization phase order searches," in *Proc. Int. Conf. Compil. Archit. Synthesis Embedded Syst. (CASES)*, Oct. 2013, pp. 1–10.

[108] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proc. IEEE 35th Int. Symp. Comput. Archit. (ISCA)*, Jun. 2008, pp. 39–50.

[109] B. Porter, M. Grieves, R. R. Filho, and D. Leslie, "Rex: A development platform and online learning approach for runtime emergent software systems," in *Proc. Usenix Conf. Symp. Oper. Syst. Design Implement.*, Nov. 2016, pp. 333–348.

[110] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin, "Vconf: A reinforcement learning approach to virtual machines auto-configuration," in *Proc. 6th Int. Conf. Autonom. Comput. (ICAC)*, 2009, pp. 137–146.

[111] M. G. Lagoudakis and M. L. Littman, "Algorithm selection using reinforcement learning," in *Proc. 7th Int. Conf. Mach. Learn. (ICML)*, 2000, pp. 511–518.

[112] N. Mishra, J. D. Lafferty, H. Hoffmann, and C. Imes, "CALOREE: Learning control for predictable latency and low energy," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2018, pp. 184–198.

[113] M. K. Emani and M. O'Boyle, "Change detection based parallelism mapping: Exploiting offline models and Online adaptation," in *Proc. 27th Int. Workshop Lang. Compil. Parallel Comput. (LCPC)*, 2014, pp. 208–223.

[114] Y. Li, "Deep reinforcement learning: An overview," *CoRR*, 2017.

[115] J. Ansel *et al.*, "Opentuner: An extensible framework for program autotuning," in *Proc. PACT*, 2014, pp. 303–316.

[116] C. K. Williams and C. E. Rasmussen, "Gaussian processes for regression," in *Proc. Adv. Neural Inf. Process. Syst.*, 1996, pp. 514–520.

[117] C. E. Rasmussen and C. K. Williams, *Gaussian Processes for Machine Learning*, vol. 1. Cambridge, MA, USA: MIT Press, 2006.

[118] M. K. Emani and M. O'Boyle, "Celebrating diversity: A mixture of experts approach for runtime mapping in dynamic environments," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2015, pp. 499–508.

[119] H. D. Nguyen and F. Chamroukhi (2017). "An introduction to the practical and theoretical aspects of mixture-of-experts modeling." [Online]. Available: https://arxiv.org/abs/1707.03538

[120] R. Polikar, "Ensemble based systems in decision making," *IEEE Circuits Syst. Mag.*, vol. 6, no. 3, pp. 21–45, Sep. 2006.

[121] L. Rokach, "Ensemble-based classifiers," *Artif. Intell. Rev.*, vol. 33, nos. 1–2, pp. 1–39, 2010.

[122] Y. Jiang *et al.*, "Exploiting statistical correlations for proactive prediction of program behaviors," in *Proc. 8th Annu. IEEE/ACM Int. Symp. Code Generat. Optim. (CGO)*, Apr. 2010, pp. 248–256.

[123] V. S. Marco, B. Taylor, B. Porter, and Z. Wang, "Improving spark application

throughput via memory aware task co-location: A mixture of experts approach," in *Proc. ACM/IFIP/USENIX Middleware Conf.*, 2017, pp. 95–108.

[124] B. Singer and M. M. Veloso, "Learning to predict performance from formula modeling and training data," in *Proc. 7th Int. Conf. Mach. Learn. (ICML)*, 2000, pp. 887–894.

[125] E. Park, J. Cavazos, and M. A. Alvarez, "Using graph-based program characterization for predictive modeling," in *Proc. 10th Int. Symp. Code Generat. Optim. (CGO)*, 2012, pp. 196–206.

[126] A. M. Malik, "Spatial based feature generation for machine learning based optimization compilation," in *Proc. 9th Int. Conf. Mach. Learn. Appl.*, 2010, pp. 925–930.

[127] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Nov. 2012, pp. 141–151.

[128] Y. Luo, G. Tan, Z. Mo, and N. Sun, "Fast: A fast stencil autotuning framework based on an optimal-solution space model," in *Proc. 29th ACM Int. Conf. Supercomput.*, 2015, pp. 187–196.

[129] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 189–204, 2000.

[130] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *Proc. 14th Int. Conf. Archit. Support Program. Lang. Operat. Syst. (ASPLOS XIV)*, 2009, pp. 265–276.

[131] J. Cavazos *et al.*, "Automatic performance model construction for the fast software exploration of new hardware designs," in *Proc. Int. Conf. Compil. Archit. Synthesis Embedded Syst. (CASES)*, 2006, pp. 24–34.

[132] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra, "Using predictivemodeling for cross-program design space exploration in multicore systems," in *Proc. IEEE 16th Int. Conf. Parallel Archit. Compilation Techn.*, Sep. 2007, pp. 327–338.

[133] M. Namolaru, A. Cohen, G. Fursin, A. Zaks, and A. Freund, "Practical aggregation of semantical program properties for machine learning based optimization," in *Proc. Proc. Int. Conf. Compil. Archit. Synth. Embedded Syst. (CASES)*, 2010, pp. 197–206.

[134] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag, 2006.

[135] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. de Bosschere, "Performance prediction based on inherent program similarity," in *Proc. IEEE Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Swep. 2006, pp. 114–122.

[136] N. E. Rosenblum, B. P. Miller, and X. Zhu, "Extracting compiler provenance from program binaries," in *Proc. 9th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng. (PASTE)*, 2010, pp. 21–28.

[137] A. Bhattacharyya, G. Kwasniewski, and T. Hoefler, "Using compiler techniques to improve automatic performance modeling," in *Proc. Int. Conf. Parallel Archit. Compilation (PACT)*, 2015, pp. 468–479.

[138] M. E. Taylor, K. E. Coons, B. Robatmili, B. A. Maher, D. Burger, and K. S. McKinley, "Evolving compiler heuristics to manage communication and contention," in *Proc. 24th Conf. Artif. Intell. (AAAI)*, 2010, pp. 1690–1693.

[139] P.-S. Ting, C.-C. Tu, P.-Y. Chen, Y.-Y. Lo, and S.-M. Cheng (2016). "FEAST: An automated feature selection framework for compilation tasks." [Online]. Available: https://arxiv.org/abs/1610.09543

[140] E. Park, C. Kartsaklis, and J. Cavazos, "HERCULES: Strong patterns towards more intelligent predictive modeling," in *Proc. 43rd Int. Conf. Parallel Process.*, 2014, pp. 172–181.

[141] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is 'nearest neighbor' meaningful?" in *Proc. Int. Conf. Database Theory*, 1999, pp. 217–235.

[142] I. Fodor, "A survey of dimension reduction techniques," Lawrence Livermore Nat. Lab., Tech. Rep., 2002.

[143] J. Thomson, M. F. O'Boyle, G. Fursin, and B. Franke, "Reducing training time in a one-shot machine learning-based compiler," *Lang. Compil. Parallel Comput.*, vol. 5898, pp. 399–407, 2009.

[144] Y. Bengio, "Learning deep architectures for AI," *Found. Trends Mach. Learn.*, vol. 2, no. 1, pp. 1–127, 2009.

[145] L. Deng, M. L. Seltzer, D. Yu, A. Acero, A.-R. Mohamed, and G. Hinton, "Binary coding of speech spectrograms using a deep auto-encoder," in *Proc. 11th Annu. Conf. Int. Speech Commun. Assoc.*, 2010.

[146] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proc. AAAI*, 2016, pp. 1287–1293.

[147] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. ASE 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 87–98.

[148] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "Synthesizing benchmarks for predictive modeling," in *Proc. Int. Symp. Code Generat. Optim. (CGO)*, 2017, pp. 86–99.

[149] M. White, M. Tufano, and M. Martinez, M. Monperrus, and D. Poshyvanyk (2017). "Sorting and transforming program repair ingredients via deep learning code similarities." [Online]. Available: https://arxiv.org/abs/1707.04742

[150] L. Almagor *et al.*, "Finding effective compilation sequences," in *Proc. ACM SIGPLAN/SIGBED Conf. Lang. Compil. Tools Embedded Syst. (LCTES)*, 2004, pp. 231–239.

[151] K. D. Cooper *et al.*, "ACME: Adaptive compilation made efficient," in *Proc. ACM SIGPLAN/SIGBED Conf. Lang. Compil. Embedded Syst. (LCTES)*, 2005, pp. 69–77.

[152] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos, "MiCOMP: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, p. 29, 2017.

[153] K. D. Cooper, D. Subramanian, and L. Torczon, "Adaptive optimizing compilers for the 21st century," *J. Supercomput.*, vol. 23, no. 1, pp. 7–22, 2002.

[154] D. R. White, A. Arcuri, and J. A. Clark, "Evolutionary improvement of programs," *IEEE Trans. Evol. Comput.*, vol. 15, no. 4, pp. 515–538, Aug. 2011.

[155] G. Fursin and O. Temam, "Collective optimization: A practical collaborative approach," *ACM Trans. Archit. Code Optim.*, vol. 7, no. 4, p. 20, Dec. 2010.

[156] J. Kukunas, R. D. Cupper, and G. M. Kapfhammer, "A genetic algorithm to improve linux kernel performance on resource-constrained devices," in *Proc. 12th Annu. Conf. Companion Genetic Evol. Comput. (GECCO)*, 2010, pp. 2095–2096.

[157] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative optimization in the polyhedral model: Part II, multidimensional time," in *Proc. 29th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2008, pp. 90–100.

[158] K. Asanovic *et al.*, "The landscape of parallel computing research: A view from Berkeley," Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2006-183, 2006.

[159] Y. Zhang, M. Voss, and E. S. Rogers, "Runtime empirical selection of loop schedulers on hyperthreaded SMPs," in *Proc. 19th IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Apr. 2005, p. 44b.

[160] D. Rughetti, P. D. Sanzo, B. Ciciani, and F. Quaglia, "Machine learning-based self-adjusting concurrency in software transactional memory systems," in *Proc. IEEE 20th Int. Symp. Modeling Anal. Simulation Comput. Telecommun. Syst.*, Aug. 2012, pp. 278–285.

[161] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and Qos-aware cluster management," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Operat. Syst. (ASPLOS)*, 2014, pp. 127–144.

[162] X. Chen and S. Long, "Adaptive multi-versioning for openmp parallelization via machine learning," in *Proc. 15th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, 2009, pp. 907–912.

[163] M. Castro, L. F. W. Góes, C. P. Ribeiro, M. Cole, M. Cintra, and J. F. Méhaut, "A machine learning-based approach for thread mapping on transactional memory applications," in *Proc. 18th Int. Conf. High Perform. Comput.*, 2011, pp. 1–10.

[164] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande, "Brainy: Effective selection of data structures," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2011, pp. 86–97.

[165] Z. Wang and M. F. P. O'Boyle, "Using machine learning to partition streaming programs," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 3, p. 20, 2013.

[166] C. Chan, J. Ansel, Y. L. Wong, S. Amarasinghe, and A. Edelman, "Autotuning multigrid with petabricks," in *Proc. ACM/IEEE Conf. Supercomput. (SC)*, 2009, Art. no. 5.

[167] M. Pacula, J. Ansel, S. Amarasinghe, and U.-M. O'Reilly, "Hyperparameter tuning in bandit-based adaptive operator selection," in *Proc. Eur. Conf. Appl. Evol. Comput. (EuroSys)*, 2012, pp. 73–82.

[168] J. Ansel, "Siblingrivalry: Online autotuning through local competitions," in *Proc. Int. Conf. Compil., Archit. Synth. Embedded Syst. (CASES)*, 2012, pp. 91–100.

[169] M. J. Voss and R. Eigemann, "High-level adaptive program optimization with adapt," in *Proc. 8th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, 2001, pp. 93–102.

[170] A. Tiwari and J. K. Hollingsworth, "Online adaptive code generation and tuning," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2011, pp. 879–892.

[171] J. Ren, L. Gao, H. Wang, and Z. Wang, "Optimise Web browsing on heterogeneous mobile platforms: A machine learning based approach," in *Proc. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, May 2017, pp. 1–9.

[172] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile Web browsing on big/little systems," in *Proc. HPCA*, Feb. 2013, pp. 13–24.

[173] Z. Wang, M. F. P. O'Boyle, and M. K. Emani, "Smart, adaptive mapping of parallelism in the presence of external workload," in *Proc. IEEE/ACM Int. Symp. Code Generat. Optim. (CGO)*, Feb. 2013, pp. 1–10.

[174] D. Grewe, Z. Wang, and M. F. P. O'Boyle, "A workload-aware mapping approach for data-parallel programs," in *Proc. 6th Int. Conf. High Perform. Embedded Archit. Compil. (HiPEAC)*, 2011, pp. 117–126.

[175] D. Grewe, Z. Wang, and M. F. O'Boyle, "OpenCL task partitioning in the presence of GPU contention," in *Proc. Int. Workshop Lang. Compil. Parallel Comput.*, 2013, pp. 87–101.

[176] L. Tang, J. Mars, and M. L. Soffa, "Compiling for niceness: Mitigating contention for Qos in warehouse scale computers," in *Proc. 10th Int. Symp. Code Generat. Optim. (CGO)*, 2012, pp. 1–12.

[177] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa, "Reqos: Reactive static/dynamic compilation for Qos in warehouse scale computers," in *Proc. 18th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2013, pp. 89–100.

[178] A. Matsunaga and J. A. B. Fortes, "On the use of machine learning to predict the time and resources consumed by applications," in *Proc. 10th IEEE/ACM Int. Conf. Cluster Cloud Grid Comput. (CCGRID)*, 2010, pp. 495–504.

[179] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proc. NSDI*, 2016, pp. 363–378.

[180] S. Sankaran, "Predictive modeling based power estimation for embedded multicore systems," in *Proc. ACM Int. Conf. Comput. Frontiers (CF)*, 2016, pp. 370–375.

[181] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "Smite: Precise QoS prediction on real-system smt processors to improve utilization in warehouse scale computers," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO-47)*, Dec. 2014, pp. 406–418.

[182] V. Petrucci, "Octopus-man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2015, pp. 246–258.

[183] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, and R. Katz, "Multi-task learning for straggler avoiding predictive job scheduling," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 3692–3728, 2016.

[184] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2014, pp. 349–360.

[185] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2016, pp. 266–280.

[186] E. Wong, T. Liu, and L. Tan, "Clocom: Mining existing source code for automatic comment generation," in *Proc. IEEE 22nd Int. Conf. Softw. Anal. Evol. Reeng. (SANER)*, Mar. 2015, pp. 380–389.

[187] J. Fowkes and C. Sutton, "Parameter-free probabilistic api mining across GitHub," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2016, pp. 254–265.

[188] A. T. Nguyen *et al.*, "API code recommendation using statistical learning from fine-grained changes," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2016, pp. 511–522.

[189] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program. Syst. Lang. Appl. (OOPSLA)*, 2016, pp. 731–747.

[190] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of Android applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2016, pp. 343–355.

[191] P. Balaprakash, R. B. Gramacy, and S. M. Wild, "Active-learning-based surrogate models for empirical performance tuning," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2013, pp. 1–8.

[192] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather, "Fast automatic heuristic construction using active learning," in *Proc. Int. Workshop Lang. Compil. Parallel Comput.*, 2014, pp. 146–160.

[193] M. Zuluaga, G. Sergent, A. Krause, and M. Püschel, "Active learning for multi-objective optimization," in *Proc. Int. Conf. Mach. Learn.*, 2013, pp. 462–470.

[194] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather, "Minimizing the cost of iterative compilation with active learning," in *Proc. Int. Symp. Code Generat. Optim. (CGO)*, 2017, pp. 245–256.

[195] A. Evaluation. *About Artifact Evaluation*. [Online]. Available: http://www.artifact-eval.org/about.html

[196] cTuning Foundation. *Artifact Evaluation for Computer Systems Research*. [Online]. Available: http://ctuning.org/ae/

ABOUT THE AUTHORS

Zheng Wang received the Ph.D. degree in computer science from The University of Edinburgh, Edinburgh, U.K., in 2011.

Currently, he is an Assistant Professor at Lancaster University, Lancaster, U.K., where he leads the Distributed Systems research group. From 2005 to 2007, he worked as an R&D Engineer at IBM China. His research focus is in the areas of parallel compilers, runtime systems, code security, and the application of machine learning to tackle the challenging optimization problems within these areas.

Prof. Wang received three best paper awards for his work on machine-learning-based compiler optimization (PACT'10, CGO'17, and PACT'17).

Michael O'Boyle is a Professor of Computer Science at the University of Edinburgh, Edinburgh, U.K. He is a founding member of HiPEAC, the Director of the ARM Research Centre of Excellence at Edinburgh, and the Director of the Engineering and Physical Sciences Research Council (EPSRC) Centre for Doctoral Training in Pervasive Parallelism. He is best known for his work in incorporating machine learning into compilation and parallelization, automating the design and construction of optimizing technology. He has published over 100 papers and received three best paper awards.

Prof. O'Boyle is a Senior Research Fellow of EPSRC and a Fellow of the British Computer Society (BCS). He was presented with the ACM International Symposium on Code Generation and Optimization (ACM CGO) Test of Time award in 2017. v