Optimization Space Learning: A Lightweight, Noniterative Technique for Compiler Autotuning

Tamim Burgstaller
tamim.burgstaller@ist.tugraz.at
Graz University of Technology
Graz, Austria

Damian Garber
dgarber@ist.tugraz.at
Graz University of Technology
Graz, Austria

Viet-Man Le
vietman.le@ist.tugraz.at
Graz University of Technology
Graz, Austria

Alexander Felfernig
alexander.felfernig@ist.tugraz.at
Graz University of Technology
Graz, Austria

ABSTRACT

Compilers are highly configurable systems. One can influence the performance of a compiled program by activating and deactivating selected compiler optimizations. However, automatically finding well-performing configurations is a challenging task. We consider expensive iteration, paired with recompilation of the program to optimize, as one of the main shortcomings of state-of-the-art approaches. Therefore, we propose Optimization Space Learning, a lightweight and noniterative technique. It exploits concepts known from configuration space learning and recommender systems to discover well-performing compiler configurations. This reduces the overhead induced by the approach significantly, compared to existing approaches. The process of finding a well-performing configuration is 800k times faster than with the state-of-the-art techniques.

CCS CONCEPTS

• **Software and its engineering → Compilers**; • **Computing methodologies → Discrete space search**; *Heuristic function construction*; Instance-based learning.

KEYWORDS

Configuration, Configuration Space Learning, Compiler, Performance Optimization, Collaborative Filtering, Compiler Autotuning

ACM Reference Format:
Tamim Burgstaller, Damian Garber, Viet-Man Le, and Alexander Felfernig. 2024. Optimization Space Learning: A Lightweight, Noniterative Technique for Compiler Autotuning. In *28th ACM International Systems and Software Product Line Conference (SPLC '24), September 02–06, 2024, Dommeldange, Luxembourg*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3646548.3672588

1 INTRODUCTION

Modern compilers are among the most sophisticated, but also highly variable software systems. They are capable of applying a variety of optimizations to a program, in different combinations. A common compiler, such as GCC[1], may have more than 200 optimization options, which can affect the compilation process and the resulting executable in various ways. GCC offers a set of flags for default combinations that work well on a large set of different programs. However, as Gong and Chen [17] state, default options often lead to suboptimal and sometimes even to the worst performance, thus outlining the need for program-specific combinations. Automatically finding such a combination is commonly referred to as autotuning, whereas the particular problem of selecting optimization options to be applied is called the *phase selection problem* [24].

The unconstrained nature of the combinations of GCC's optimizations options leads to more than 2^{200} ($\approx 10^{60}$) distinct combinations, which makes exhaustive exploration infeasible. Furthermore, interactions between optimizations can influence their effectiveness. Therefore, machine learning approaches have been developed as solutions to the phase selection problem [6]. These can be split into two groups: Supervised and unsupervised learning approaches. The former suffer from several problems that can make them unpractical: Their effectiveness depends heavily on the quality of the training data used, and collecting large amounts of training data can be very costly in practice [11]. The latter are therefore more commonly used [6], particularly in state-of-the-art approaches [7, 11, 12, 46]. Many of the unsupervised approaches apply variants of iterative compilation, as proposed by Bodin et al. [10]. In general, this means that they compile the program with a number of different combinations in each iteration and keep the best-performing ones. However, this kind of iteration is runtime-expensive, as it requires compiling the program more than once. Finding suitable optimization options for a program can take a long time, sometimes many hours, especially if the program has long compile times, like large projects often do.

To counteract all of the issues mentioned above, we propose the novel approach of *Optimization Space Learning (OSL)*, based on the work of Garber et al. [15] on constraint solvers. It relies on collaborative filtering [13], which requires a training set of premeasured data. For OSL, this training set consist of programs that have been measured for their performance after being compiled with a number of combinations of optimization options. Given a program source code, our approach compares the program with the programs in the training set to find the most similar one, then recommends the combination under which that similar program

[1] https://gcc.gnu.org/

SPLC '24, September 02–06, 2024, Dommeldange, Luxembourg

ACM ISBN 979-8-4007-0593-9/24/09
https://doi.org/10.1145/3646548.3672588

performed best. The details of the algorithm, as well as additional methods to improve the recommendation, are the main contribution of this paper.

As collaborative filtering [13] is a supervised learning technique, it relies on training data. OSL needs two dimensions of training data: First, a set of programs, and second, a set of combinations of optimization options. Each program is compiled with each combination and its performance is measured to create the matrix we refer to as training data. This process is expensive and time-consuming, therefore, the number of programs and combinations should be minimized. This requires intelligent choice of both to make OSL perform well: We need programs and combinations of configuration options with low redundancy. For the programs, we achieve this by using a common benchmark. For the optimization options, we synthesize data using heuristics from the field of configuration space learning [3] that allow us to cover many different combinations of options in very few items. These techniques allow us to counteract the issues of supervised learning approaches mentioned above.

The remainder of this paper is structured as follows. In Section 2, we give the necessary background information on compiler autotuning, its state of the art and current shortcomings, as well as on collaborative filtering and configuration space learning. The main contribution, the OSL algorithm, is described in Section 3, both in its baseline version and with various ideas to improve its results. To provide OSL with the required dataset, we apply data synthesis, which is described in Section 4. In Section 5, we evaluate OSL, compare it to existing algorithms, point out its strengths and weaknesses, and show that it is a capable competitor to state-of-the art approaches. Finally, in Sections 7 and 8, we discuss future work for OSL and related work to the contents of this paper.

2 BACKGROUND

This section provides background knowledge needed to fully understand the later sections of the paper. First, an introduction to compiler autotuning is given. Afterwards, the topic of configuration spaces is explained. The section concludes with a brief overview of collaborative filtering and nearest neighbor-approaches.

2.1 Compiler Autotuning

Compilers are highly variable software systems. A common compiler, such as the aforementioned GCC, offers a large number of optimization options[2]. Each of these options enables or disables a certain optimization that can be performed on the source code. In general, compiler autotuning is about automatically handling these optimization options. The goal is to provide a combination of optimization options such that the compiled program's performance is optimized with respect to a certain nonfunctional property, usually runtime. In the remainder of this paper, we will refer to a combination of optimization options as a *configuration*.

According to Ashouri et al. [6], the field of compiler autotuning handles two major problems: The *phase selection problem* [24] and the *phase ordering problem* [41], both of which have the aim of producing well-performing programs. The former figures out which optimizations to apply, the latter identifies the order in which to apply optimizations. Within the scope of this work, we will focus solely on the phase selection problem.

2.1.1 State of the Art. A naive approach to solve the phase selection problem would be to exhaustively try out all possible configurations and determine which one performs best. This is infeasible due to the large number of distinct configurations. To solve this issue, a variety of machine learning approaches have been developed in the past, many of which are described in an overview by Ashouri et al. [6]. A reoccurring pattern in most of these approaches is the use of iteration. Bodin et al. [10] were amongst the earliest to propose iterative compilation for compiler autotuning. Iterative compilation can be described as a kind of evolutionary search technique. A configuration of optimization options is chosen as a starting point, and the program to optimize is compiled on the basis of these options. Based on the performance of the program, the configuration is refined. This procedure is then repeated until the result is satisfactory or a termination condition is reached. A myriad of approaches have been developed over the years that apply this general pattern [12, 16, 22, 35, 40], which will be discussed in more detail in Section 8. A more recent approach to search redirection is *COBAYN* by Ashouri et al. [7], which makes use of Bayesian Networks to reduce the search space and focus on the most promising region. The current state-of-the-art technique is *BOCA*, proposed by Chen et al. [11]. This technique increases the efficiency of compiler autotuning by using Bayesian Optimization to determine impactful optimizations and focus the search on those. The latest emerging approach is multiple phase learning, proposed by Zhu et al. [46]. Their tool, *CompTuner*, builds a prediction model for the runtime of a program with regard to the optimization configuration first, and then uses this information in the search process. The search itself is performed using a particle swarm optimization algorithm [25]. The performance improvement comes from not having to compile and run a program over and over again within the search process, as the prediction model is used to predict the outcome of this.

2.1.2 Shortcomings in the State of the Art. First and foremost, all of the aforementioned compiler autotuning techniques deliver good results in terms of program performance improvement, thus achieving the major goal of compiler autotuning; more detailed information on this is given in Section 5. However, there is a drawback, as the application of iteration is expensive. This is a major issue for approaches such as *Cole* by Hoste and Eeckhout [22]. They state that on a single machine, their algorithm would run for a full 50 days until it has built a Pareto frontier, which is central to their approach. The problem is even worse for iterative compilation, as the program in question has to be compiled, and possibly also run, once in each iteration - or even several times, depending on the concrete approach used. To counteract this issue, supervised learning methods have been applied [6]. These methods depend on training data, often in large amounts, which is impractical to collect when each data point requires the program to be compiled one or several times. Therefore, supervised learning does not naturally work well for compiler autotuning.

Two concrete examples of techniques that deal with these issues are *BOCA* [11] and *CompTuner* [46]. Zhu et al. [46] evaluated both approaches and provide concrete data on the time it takes to find a configuration for programs in their used benchmark (programs

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

with 200 - 27, 000 lines of code). Using the compiler GCC, they state that for *BOCA*, it takes between 1, 900 and 5, 000 seconds, while *CompTuner* takes between 2, 100 and 5, 600 seconds to compute a desirable configuration. These times are surely acceptable when compiling a project once, e.g., for a software release. Those will also most likely be the cases where a customized configuration is desired. However, most of the time, the compilation process itself will take just a few seconds, and therefore these techniques are not reasonably applicable. This applies in particular if one was to develop a plugin for a compiler, such as GCC, to enable autotuning directly. Here, many compilation processes would be prolonged to many times their original length, which is highly inconvenient and undesirable for the user. Hellsten et al. [21] even describe being able to use an autotuner during development as the "holy grail" of the research area.

These issues show that there is a need for a lightweight, noniterative approach to compiler autotuning. Our proposed solution is *Optimization Space Learning (OSL)*, a technique that fulfills these properties. We describe this technique in detail in Section 3.

2.2 Configuration Spaces

The learning of configuration spaces concerns itself with the issue of learning the structure and rules of a given configurable system in order to either propose a valid configuration or perform an optimization task of some kind (w.r.t. a non-functional property). The task of finding a representative subset of all possible system parameter settings is a major task of *configuration space learning*. The phase selection problem requires choosing a set of options from all the available ones [24]. In the context of compiler optimization, this means to enable or disable each option separately. Therefore, we view the phase selection problem as a configuration task, based on the findings of existing work [3, 9, 15].

Definition 2.1 (Configuration Task). A configuration task consists of three sets (V, D, C). The set $V = \{v_1, ..., v_n\}$ contains n variables, while in $D = \{d_1, ..., d_n\}$, each element d_i contains the domain of the variable v_i (i.e., the values that may be assigned to v_i). Finally, the set $C = \{c_1, ..., c_m\}$ consists of m constraints over the variables in V. A valid configuration over (V, D, C) assigns exactly one value from the domain d_i to each variable v_i in V, such that none of the constraints in C is violated.

For the phase selection problem, each optimization option is represented as a variable in V. A boolean domain in D represents whether it is enabled or disabled. The set C allows us to define which optimizations must or must not be applied together. Technically, any optimization option can be combined with any other, therefore the set C is empty by default. However, the application of an optimization to a program may affect the effectiveness of consecutively applied optimizations. Thus, C could also be used to rule out bad combinations. These can be identified, for instance, using the method proposed by Ben Asher et al. [8] to find conflicting pairs in compiler optimizations.

Definition 2.2 (Configuration Space). Each configuration task is associated with a configuration space. The configuration space S of a configuration task (V, D, C) consists of all of its valid configurations.

Alves Pereira et al. [3] describe a pattern often used for configuration space learning: Sampling, Measuring, Learning. First, the configuration space is sampled, i.e., a reduced set of configurations is chosen. Then, this set is measured with regard to the non-functional property of choice. Finally, the gathered information is used to learn a model, which should be capable of predicting the performance of arbitrary configurations. Learning a prediction model from collected data is called machine learning (ML), and is a huge research field of its own. In this work, we will only use collaborative filtering [13], which was originally used in recommender systems and has been found to work well with configurable systems [4, 14]. It will be explained later in this section. On the other hand, measuring the performance of a configuration is very specific to a concrete experiment. It should be mentioned that this step can be costly, depending on the number of data samples to be measured. To address this issue, data synthesis can be applied in order to receive a reduced dataset, whilst losing as little information as possible. Our concrete approach to sampling and data collection is explained in Section 5.

2.3 Collaborative Filtering

Collaborative filtering recommendation, as described by Ekstrand et al. [13], is based on the idea that similar targets will be affected equally positively or negatively by the same recommended item. Therefore, it is applicable in any domain which can define similarity in any way. One possibility to define similarity is by using a distance metric, where a small distance corresponds to high similarity. Jain et al. [23] describe Euclidean Distance as a generally well-performing distance metric for collaborative filtering. Formula 1 describes Euclidean Distance formally, where x and y are two n-dimensional vectors, consisting of the components x_1 to x_n and y_1 to y_n.

$$Dis(x, y) = \sum_{i=1}^{n} |y_i - x_i|^2 \tag{1}$$

Our definition of similarity, also taken from Jain et al. [23], is described in Formula 2. Effectively, this means that a smaller distance corresponds to a higher similarity.

$$Sim(x, y) = \frac{1}{1 + Dis(x, y)} \tag{2}$$

Within the context of compiler autotuning, we use collaborative filtering to assess the similarity of programs. For this, a program needs to be broken down into a vector of metrics. On these vectors, Formula 1 and consequently, Formula 2 can be applied to compute the similarity of two programs.

We illustrate this using an example in Table 1. The example consists of Program 0 and three other programs, which are in this case represented as 2-dimensional vectors consisting of their "Halstead" and "McCabe" metrics. The goal is to find the most similar program to Program 0. The given vectors are plugged into Formula 1 for the distance computation and Formula 2 for the similarity computation. In the example, the highest similarity is 16.7 % between Program 0 and Program 2. Thus, Program 2 is the most similar program to Program 0. In Section 5, we will give more concrete definitions for the use of these formulas in the context of compiler autotuning.

Table 1: Computation of the similarity of programs represented as vectors of their "Halstead" and "McCabe" metrics. Comparing Program 0 to three other programs, we find that Program 2 is the most similar one to it.

	Halstead	*McCabe*
Program 0 (P_0)	110	10
Program 1 (P_1)	111	7
Program 3 (P_2)	108	9
Program 2 (P_3)	104	13

x, y	$Dis(x, y)$	$Sim(x, y)$
P_0, P_1	10	0.091 (9.1 %)
P_0, P_2	5	0.167 (16.7 %)
P_0, P_3	45	0.022 (2.2 %)

3 OPTIMIZATION SPACE LEARNING ALGORITHM

The Optimization Space Learning algorithm (OSL) applies nearest neighbor-based collaborative filtering [13] on synthesized data [2, 29], which has been obtained using heuristics known from configuration space learning [3, 15]. The existence of a set of programs, such as a benchmark, to train and test the algorithm on is a precondition, as well as a set of configurations. Each program is compiled with each configuration, then its performance is measured. This forms a matrix of performance values. We refer to this as the training set, and will revisit the origins of the programs and, in particular, the configurations in Section 4.

3.1 Baseline Algorithm

In this context, the task of collaborative filtering is to discover programs in the training data that are similar to the target program, analyze their performance under the configurations, and recommend a configuration for the target program based on this information. For this to work, we rely on the training set to contain information about the performance of a program-configuration combinations with respect to the desired optimization targets, e.g., program runtime. In search of a good GCC configuration, the algorithm first establishes the similarity of the target to its training data inputs. The nearest neighbor is found using a suitable similarity metric. Once the nearest neighbor is found, its training results under all sampled configurations are analyzed. The algorithm then recommends the best-performing configuration with respect to the optimization target. It should be mentioned that this is the simplest, most general version of the OSL algorithm. Further ideas will be incorporated later in this section.

Figure 1 depicts how OSL selects a configuration for a target program. C_i are the synthesized configurations. Each of the benchmark programs has had its runtime measured under each configuration throughout the training phase. Benchmark program 2 is the nearest neighbor of the target by the similarity metric. Therefore, OSL recommends compiling the target program with the best-performing configuration of benchmark program 2, which in this case happens to be C_3.

Table 2: Aggregation of three configurations using an option-wise majority vote, where opt_i are optimization options, $conf_i$ is a configuration, and $conf_{agg}$ is the resulting aggregated configuration.

	opt_1	opt_2	opt_3	opt_4	opt_5
$conf_1$	1	1	0	1	0
$conf_2$	0	1	0	0	1
$conf_3$	1	0	1	0	0
$conf_{agg}$	1	1	0	0	0

3.2 Algorithm Improvement Strategies

Various strategies can be considered to improve the quality of the recommended configurations, two of which we present in the following. One of them aims to avoid fixation of the model towards highly specific properties of a single program, the other one ensures that all features of a program are equally important to the recommendation.

3.2.1 Aggregation of Configurations. *Aggregation methods* can be used to avoid overfitting on the nearest neighbor's properties. This can be applied both to the recommended configuration, by aggregation of the top configurations of the nearest neighbor program, and to the nearest neighbor itself, by using several nearest neighbors instead. We refer to these parameters as a for the a top configurations and k for the k-nearest neighbors. A straightforward method for the aggregation of configurations is the *option-wise majority vote*. This means that for each optimization option, the final value is which appears most often for that option in the aggregated configurations. This kind of majority voting is also exemplified in Table 2.

3.2.2 Normalization and Weighting. Another approach is data *normalization and weighting*. The problem to approach here is that when a sample contains features of different scales, then features with commonly higher values will affect the distance more than features with commonly low values. An example of this would be the lines of code of a program, compared to the number of branching points; the former will return larger values than the latter, regardless of the actual context and their related importance. An approach for data normalization would be *min-max feature scaling*, which maps every feature to a value between 0 and 1, regardless of its original values. This method defines the normalized value x' of a feature x as

$$x' = \frac{x - min(x)}{max(x) - min(x)} \tag{3}$$

where $max(x)$ and $min(x)$ are the maximum and minimum for that feature in the dataset. In addition, features can also be weighted by their importance for distinguishing nearest neighbors from other configurations. However, nearest neighbor collaborative filtering does not provide an implicit measure of feature importance. To counteract this issue, we measure the distance of each feature of each sample to the same feature of its nearest neighbors and take the ratio to the same measurement with random data points. The smaller this ratio gets, the more important a feature is. We assign weights to features using a linear scale, starting from 1.0 for the most important feature. Smaller weights correspond to less important features. The weighted feature value x_w for a feature x with a

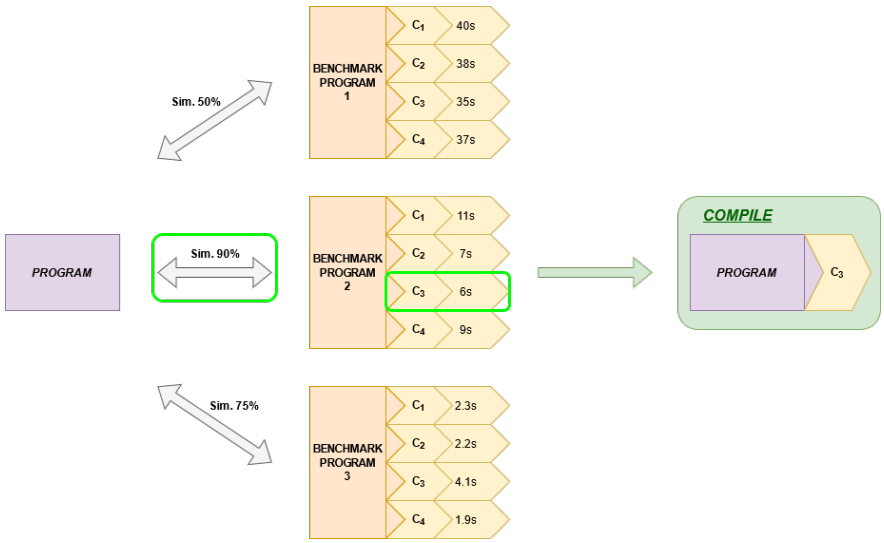

Figure 1: The baseline version of OSL, where C_i denotes a configuration. The configuration for the target program is the best-performing configuration of the most similar program in the benchmark.

weight w, $0 < w \leq 1$, is computed by the formula

$$x_w = x * w \tag{4}$$

thus scaling the values of less important features down.

4 TRAINING DATA SYNTHESIS

Our focus is to keep the training output as small as possible, while maintaining high quality in terms of compiler performance. The need for a small set is given by the intention to keep this approach as lightweight as possible, as collecting and measuring each sample is a costly procedure [43].

For the programs used in the training set, a benchmark is a suitable choice. It incorporates all the requirements in its definition: It is a small set of programs which aim to represent as many aspects of the programming language or environment as possible. Concerning the configurations to compile programs, the solution is slightly less straightforward. We apply data synthesis to generate a set of configurations that fits our needs, i.e., covers much of the configuration space in very few configurations and minimizes redundancy. In configuration space learning, the synthesis would be performed by sampling the configuration space, i.e., taking a small subset of all possible configurations. Sampling heuristics, as described by Alves Pereira et al. [3] and Garber et al. [15], are one way to achieve this. In the following, we describe sampling heuristics used in our evaluation settings.

4.1 Random Sampling

In unconstrained settings, random sampling is a reasonable choice for data synthesis. It provides a fast and easy way to reduce the search space. More sophisticated random sampling approaches can even give statistical guarantees that at least one configuration of a specific quality is contained in a sample, given it is sufficiently large - see Oh et al. [32]. However, there is a caveat to all this, which is the precondition that the randomization works uniformly. This leads to a problem for constrained settings, as common randomization methods would not lead to uniform results by default. This problem has been addressed by, among others, Plazar et al. [36]. However, it is not of specific interest for the unconstrained setting of compiler autotuning. Issues arise in other aspects of the data synthesis: A random approach would need a higher number of samples to achieve the same quality of representation as a well-constructed heuristic, which could lead to scalability issues [29]. Nevertheless, random sampling is not a bad choice, in particular for first experiments.

4.2 Heuristic-based Sampling

In the context of compiler autotuning, the aim is to learn the structure of a configuration space. Therefore, heuristic sampling might be beneficial by providing a more balanced representation of the entire configuration space. Heuristics that have such properties have been studied by Alves Pereira et al. [3] and Garber et al. [15]. In particular the latter state that the *Feature Coverage Heuristic (FCH)* yields promising results on small, constrained configuration spaces, despite some scalability issues with larger ones. The general idea of the t-wise FCH is to cover every combination of t options in at least one sampled configuration, while also using as few samples as possible to achieve this t-wise coverage. For example, with $t = 2$, the heuristic would guarantee that every possible pair of optimization options appears in at least one sample. These *t-wise sampling* approaches have also been researched in the field of combinatorial testing, for instance, by Oh et al. [33]. Moreover, tools from this field, such as ACTS [45], can generate these *covering arrays* also for configuration space learning purposes. To do so, it applies the IPOG algorithm [26]. A particularly interesting property of this heuristic is that it attempts to represent the correlation between the options. This is important, as Ben Asher et al. [8] show that conflicting pairs of optimization options are significant to the performance of a configuration in compiler autotuning. Therefore, we consider FCH as a good choice for a compiler autotuning setting.

5 EVALUATION

We evaluate the performance of OSL with the GCC compiler and a benchmark of C programs [37]. Moreover, we also compare its results to the current state-of-the-art compiler autotuning techniques. As an optimization target, we decided to go for program runtime, as this is a common choice in this research area [6], although OSL would also support any other measurable optimization target.

5.1 Experimental Setup and Prerequisites

The evaluation of OSL is against the `-O3` default optimization configuration of GCC. The `-O3` flag activates almost all optimizations available in the compiler[3]. We evaluate OSL using GCC version 11.4.0 on a Xubuntu-22.04 machine with an Intel i7 processor. No multithreading or multiprocessing was applied. The implementation of OSL is written in Python and makes use of the NumPy [20] and scikit-learn [34] libraries. For our measurements, we used perf-stat[4]. As the training set, we used the PolyBench/C Benchmark [37], which contains 30 programs written in C and is further referred to as PolyBench. This benchmark has also been used by many others, including Chen et al. [11] and Zhu et al. [46], against whom we compare our results. In order to measure the similarity of two programs, we use source code metrics, such as McCabe's Cyclomatic Complexity [28], Halstead Complexity [19], or simple counts like the number of times a certain keyword occurs. We use a total of 66 different metrics in our evaluation. To obtain these metrics, we use the *CQMetrics* tool by Spinellis et al. [38]. For the complete list of the available metrics, we refer to their documentation[5]. In this evaluation, only the first 66 of the available metrics in the list were used. These are the ones we consider source code metrics, as opposed to code style metrics. We consider the former to be a more sensible choice than the latter, as we are working with benchmark source code. The tool generates a vector of metrics from each program's source code, which can then be further processed. We use Euclidean Distance to define the distance between two vectors. The corresponding formulas for distance and similarity are Formula 1 and 2, respectively.

We apply Leave-one-out cross-validation [44], i.e., we test each of the programs in PolyBench on a model trained with the remaining 29. This way, we evaluate various settings of OSL. For sampling the configuration space, we evaluated FCH with 2-, 3-, and 4-wise coverage. The number of configurations sampled with each heuristic is described in Table 3; this number is determined by the ACTS tool [45] which was used for generating these configurations. Furthermore, we evaluated both the baseline technique and the algorithm improvements presented in section 3. We tested different values for k-nearest neighbors and aggregation of the a best-performing configurations to obtain the recommended configuration.

To measure our algorithm's performance, we compute the *speedup* compared to the GCC default `-O3`. The speedup is the ratio of the performance with the default to the performance with our algorithm. The corresponding formula is

$$speedup = \frac{t_{GCC}}{t_{OSL}} \tag{5}$$

[3]https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
[4]https://perf.wiki.kernel.org/index.php/Main_Page
[5]https://github.com/dspinellis/cqmetrics/blob/master/metrics.md

Table 3: Number of configurations sampled with different sampling heuristics. The last column denotes the time it took to measure the performance of all programs of the PolyBench benchmark with these configurations.

Heuristic	#configurations	training time
FCH, 2-wise	20	201 min
FCH, 3-wise	89	848 min
FCH, 4-wise	319	3,023 min

Table 4: The performance of OSL without and with the described improvement strategies. The speedups are in comparison with GCC, and computed using Formula 5.

Strategy	#Speedups	Average	Median
Baseline	12/30	0.944	0.968
Normalization	11/30	0.983	0.982
Normalization and Weights	15/30	0.994	0.998

where t_{GCC} is the performance using the `-O3` default and t_{OSL} is the performance with OSL. In the case of this particular evaluation, the t variables are measured program runtimes.

5.2 Results

First, we need a data synthesis heuristic to base our evaluation on. After testing FCH 2-, 3-, and 4-wise, we find that the data synthesized using the 3-wise version of the heuristic gives the best results. Additionally, parameters for k and a are needed; after testing all combinations of $k \in 1, 3, 5$ and $a \in 1, 3, 5, 10$, we discovered that the parametrization $k = 5$ and $a = 5$ delivers good results with the synthesized data. This is the basic setting for all results presented in this section.

To start with, we verify that the algorithm improvements described in Section 3. The results are shown in Table 4. Indeed, normalizing and weighting the features helped to improve the average speedup by 5%. Moreover, it increased the number of programs from the benchmark that were sped up from 12 to 15. As normalizing and weighting the features of the synthesized data improved the performance, this will be the version of OSL that we will use for the remainder of this section.

As stated above, this setting of OSL managed to speed up 15 out of 30 programs from PolyBench, with an average speedup of 0.994. From these 15 programs, 12 had a speedup of less than 1.0 - 1.1, 2 were sped up by a factor of 1.1 - 1.2, and one program, namely *floyd-warshall*, was sped up by a factor of 1.42. These findings are also depicted in the histogram in Figure 2. Another interesting number is the time the algorithm takes to compute the configurations that lead to the speedups mentioned above. On average, this takes 0.0044 seconds for each program.

The results clearly show that OSL is a technique capable of finding configurations that outperform GCC's `-O3` configuration significantly. It is also clearly visible that this works on a wide variety of different programs. Finally, the average speedup of 0.994 shows that the risk of receiving a particularly bad configuration

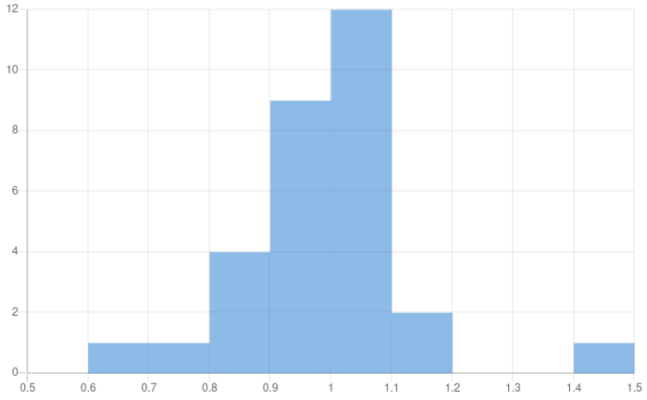

Figure 2: Histogram of the speedups achieved with OSL; the speedup factor is on the x-axis, the number of programs on the y-axis.

Table 5: The list of programs from PolyBench that was used for the comparison with other approaches.

ID	Program	#SLOC	Description
P1	correlation	248	Correlation computation
P2	covariance	218	Covariance computation
P3	symm	231	Symmetric matrix-multiply
P4	2mm	252	2 matrix multiplications
P5	3mm	267	3 matrix multiplications
P6	cholesky	212	Cholesky decomposition
P7	lu	210	LU decomposition
P8	nussinov	569	DP for sequence alignment
P9	heat-3d	211	Heat equation (3D data dom.)
P10	jacobi-2d	200	2-D Jacobi stencil comp.

from OSL is reasonably small, as the algorithm can keep up with `-O3` on average.

5.3 Comparison to other Approaches

In line with the evaluations performed by Chen et al. [11] for BOCA and by Zhu et al. [46] for CompTuner, we will only test the 10 programs listed in Table 5. They state that it has been demonstrated by Chen et al. [12] that despite tuning, the execution time of the remaining programs cannot be noticeably affected by compiler optimizations.

We compare the performance of OSL to existing approaches. For this purpose, we use a table by Zhu et al. [46], adapt it and extend it with the entries for OSL; this is Table 6. In that table, the speedup factor of the program runtime is denoted, or a "-" if no speedup was achieved. Furthermore, the values in brackets next to the speedup denote the time (in seconds) the algorithm needed to compute the configuration which achieves this speedup. The tested version of OSL was the same as above, using FCH 3-wise for configuration space learning and had the parameters $k = 5$ and $a = 5$ for aggregation, as we found it to perform well on the given task. The compared techniques are CompTuner [46], BOCA and TPE [11], random iterative compilation (RIO) [12], genetic algorithms (GA) [16], OpenTuner [5], and COBAYN [7]. In Table 6, it is visible that OSL is not the best approach in terms of speedup. BOCA and especially CompTuner are often capable of discovering better configurations. However, OSL achieves a speedup on 5 out of 10 programs, which only comes short to CompTuner and BOCA and is better than all remaining approaches, except for COBAYN, which also speeds up 5 programs. Also, it achieves the best speedup of all compared approaches on the program P9 by a significant margin.

Therefore, we can confidently state that the overall performance of OSL in terms of speedup is reasonable. However, the crucial advantage of OSL over all existing approaches is the time it takes to find such a well-performing configuration. Whenever it finds one, it is the fastest technique to do so; on average, it takes OSL a mere 0.0044 seconds. For BOCA, the average time is 3,571 seconds, and for CompTuner, it is 3,555 seconds. Thus, OSL is 811,590 times faster than BOCA and 807,954 times faster than CompTuner. This is a significant improvement compared to the existing approaches and thus, addresses the main shortcoming mentioned in Section 2. Most importantly, it demonstrates that OSL is indeed a lightweight and fast technique.

6 THREATS TO VALIDITY

The results of OSL in terms of program speedup are not outstanding, although reasonable. Other existing approaches, such as CompTuner and BOCA, achieve better speedups, as shown in Table 6. In particular, the former is capabale of speeding up every single one of the 10 programs used for the comparison, while OSL fails to do so for 5 of them. This is an expected result, since OSL manages to improve the performance of exactly half of the programs in the benchmark, and the average and median speedup are almost exactly 1.0. Both of these are shown in Table 4. In the same context, looking at Figure 2, it is visible that there are more programs that have been improved by less than 10 percent than programs whose performance has degraded by that margin. In terms of strong outliers However, it must not be missed that these matching results are also an indicator of soundness for the evaluation.

We regard system-specific behaviour as another threat to the validity of our results. Despite using a device with a common hardware architecture and operating system, the exact behaviour of a program at runtime still heavily depends on the underlying system. On this note, we also did not take any hardware-specific optimizations into account. GCC is capable of optimizing code machine-dependently for almost 60 different architectures[6]. Taking these options into account could massively change the outcome of an evaluation.

Furthermore, our approach depends on the training data, especially on the chosen programs. In our case, this was the PolyBench benchmark, but with a different benchmark or another set of programs, the results could change. A particular threat in this context would be that the programs of the training set are very similar to each other in terms of which compiler optimizations lead to good performance. This would benefit OSL due to the nearest neighbor approach it applies. In addition, the size of the benchmark could also have an effect on the resulting speedups. However, this comes

[6] https://gcc.gnu.org/onlinedocs/gcc/Submodel-Options.html

Table 6: Speedup of the programs in Table 5 compared to -O3 on GCC, and the time it took to compute a configuration that achieves such a speedup. The upper half of the table contains the speedup of runtime of the compiled program in comparison with GCC. The lower half contains the time it took each technique to compute a configuration which achieves that speedup. The best speedup achieved and the smallest time to achieve a speedup are marked in bold font, while "-" denotes that no speedup was achieved.

Technique	ID	Speedup	ID	Speedup	ID	Speedup	ID	Speedup	ID	Speedup
OSL		1.000		1.043		-		-		-
CompTuner		**1.077**		**1.080**		1.042		1.071		1.041
BOCA		-		-		**1.075**		1.071		**1.046**
TPE	P1	-	P2	-	P3	1.046	P4	1.072	P5	-
RIO		-		-		1.042		-		-
GA		-		-		-		-		1.041
OpenTuner		-		-		-		1.075		-
COBAYN		-		**1.080**		1.068		**1.079**		-
OSL		1.010		1.016		-		**1.109**		-
CompTuner		1.013		1.073		1.029		1.025		**1.055**
BOCA		1.014		-		1.030		1.028		**1.055**
TPE	P6	-	P7	-	P8	-	P9	1.027	P10	-
RIO		1.016		-		1.029		-		-
GA		1.013		-		-		1.025		-
OpenTuner		-		**1.075**		**1.033**		-		-
COBAYN		**1.064**		-		-		1.028		-

Technique	ID	Time [s]	ID	Time [s]	ID	Time [s]	ID	Time [s]	ID	Time [s]
OSL		**0.0043**		**0.0039**		-		-		-
CompTuner		3107.00		4067.00		2573.00		3720.00		**2976.00**
BOCA		-		-		1923.00		3726.00		3639.00
TPE	P1	-	P2	-	P3	3775.00	P4	3112.00	P5	-
RIO		-		-		4172.00		-		-
GA		-		-		-		-		3160.00
OpenTuner		-		-		-		4691.00		-
COBAYN		-		4727.00		**1092.00**		**3102.00**		-
OSL		**0.0039**		**0.0048**		-		**0.0040**		-
CompTuner		4726.00		5549.00		3661.00		2976.00		**2192.00**
BOCA		4971.00		-		4082.00		3420.00		3026.00
TPE	P6	-	P7	-	P8	-	P9	1.027 (2637)	P10	-
RIO		3018.00		-		**3264.00**		-		-
GA		3862.00		-		-		3684.00		-
OpenTuner		-		6792.00		4970.00		-		-
COBAYN		3109.00		-		-		4116.00		-

at the cost of higher training times than there already are. At some point, the benchmark could be too large to be a feasible training set for the algorithm.

7 FUTURE WORK

First and foremost, the results of OSL in terms of program speedup need to be improved. We achieve very good algorithm runtime and reasonable results. However, the state-of-the-art approaches can achieve even better results. The gap between these approaches and OSL needs to be closed. This could be done, for example, by using a different, possible larger benchmark as the training set.

Another interesting option is to allow for multi-target optimization, as, for instance, *Cole* [22] does. This does not require complex changes in the algorithm. Instead, it is sufficient to collect the required data in the training phase and search for the best performance in a multi-dimensional space, which is not fundamentally different from finding it in a one-dimensional space. It is required that the length of a vector in this multi-dimensional space is defined. By using nontrivial definitions there, it is also possible to assign different weights to the targets. This way, it is feasible to optimize a program to achieve good performance in multiple fields, such as runtime and energy consumption.

Furthermore, the similarity metrics and distance function could also be altered in order to achieve better results. The CQMetrics tool already provides a variety of different metrics. These metrics should be evaluated one by one, to find out which metrics are significant for program similarity in the context of compiler autotuning. A lot of research has already been done in this area, as the overview

by Walenstein et al. [42] shows. This includes machine learning approaches, such as graph neural networks proposed by Nair et al. [31]. Including these techniques into our approach could lead to significant performance improvements.

Alternatively to this idea, other recommender and machine learning techniques can be investigated. An improvement for the data set could be achieved through matrix factorization. This would help to increase the size of the training data set, and thus, improve the performance of collaborative filtering, without significantly increasing the effort to retrieve it. A more radical change would be the use of an entirely different machine learning approach. Examples of these would be Neural Networks or Support Vector Machines.

In addition, it is a goal to eliminate the need for a real benchmark, as this is a reliance on real-world data and therefore a threat to the generality of our approach. A possible solution is to synthesize these programs, similar to the configurations. There exists some prior work, such as by Gulwani [18], to base this research on. However, a slight diversion from those methods will be required, as their main focus is on turning user requirements into programs. Instead, the task of program synthesis can be seen as a kind of configuration task either, thus enabling the application of known methods and heuristics. One could then treat a generated program as a configuration in the configuration space of the programming language. A similar, yet different idea would be using techniques known from the field of automata learning, for instance as implemented in the AALpy library [30]. This library provides implementations of various heuristics, including coverage heuristics, as oracles for state machines. Translating the essence of a programming language into such an automaton would allow to make use of these heuristics to generate programs. In conclusion, implementing such an approach could lead to a leap in terms of the performance of OSL, and at the same time improve its generality.

Finally, we think OSL should be applied in a compiler, which is feasible because of its short runtime. This could be an entirely new compiler for a language, or an existing one like GCC, in the form of a plugin.

8 RELATED WORK

Compiler autotuning has been widely researched over the course of the past two decades. Bodin et al. [10] were amongst the earliest to propose iterative compilation for compiler autotuning. Another early approach was *Optimization Space Exploration (OSE)*, presented by Triantafyllis et al. [40]. An effective, but expensive approach is Random Iterative Optimization, as proposed by Chen et al. [12], which combines random search with iterative compilation. Hoste and Eeckhout developed a technique called *Cole*, which could handle multi-target optimizations (e.g., for runtime and compile time) through the iterative creation of a Pareto frontier [22] based on the SPEA2 algorithm [47], which works similar to a Genetic Algorithm. Genetic Algorithms themselves have also been applied, for instance by Garciarena and Santana [16]. In this context, they learn and exploit interactions between optimization options. This knowledge is then used to direct the search, using estimation of distribution algorithms, in order to improve the results. The *irace* package for automatic algorithm configuration [27] uses iterated racing, which is based on a similar idea, and this has also been applied to compiler autotuning by Pérez Cáceres et al. [35]. A different approach was proposed by Ashouri et al., namely *COBAYN*, which makes use of Bayesian Networks [7]. Among the most modern, state-of-the-art techniques is *BOCA* by Chen et al. [11]. It implements Bayesian Optimization together with Random Forests. The latest development is multiple-phase learning, which was proposed by Zhu et al. [46]. For reasons of brevity, we left out many other works that would have fit in this section. However, many of those have been described in an overview by Ashouri et al. [6], to which we refer for further information on the advances in the field of compiler autotuning.

There exists additional work on autotuning in general. BaCO by Hellsten et al. [21] is an example of a compiler optimization framework in a slightly different setting. Their focus is on portability between different hardware (CPU, GPU, and FPGA) instead of program optimization alone. In addition to all these, compiler optimizations themselves have been extensively studied. A recent example is a study of the function inlining optimization by Theodoridis et al. [39].

Regarding the research field of configuration spaces, our main interest lies in synthesis methods, such as the ones described by Alves Pereira et al. [2], or the more sophisticated approach by Oh et al. [32]. However, due to the unconstrained nature of this particular case and the relatively small configuration space (compared to colossal spaces described by Acher et al. [1]), our focus lies on coverage-based approaches. In particular, we are interested in t-wise sampling approaches, like provided by Oh et al. [33]. Within the scope of this work, we used the *ACTS* tool, as presented by Yu et al. [45], to compute 2- and 3-way coverage of the optimization space. Sampling heuristics linked with feature coverage have been proposed by Alves Pereira et al. [3] and Garber et al. [15].

9 CONCLUSION

In this paper, we presented Optimization Space Learning (OSL), a lightweight and non-iterative technique for compiler autotuning. OSL is based on two well-known techniques: Configuration Space Learning and Collaborative Filtering. The former helps to reduce the search space, while the latter is used to find a well-performing configuration for a new program. We show that OSL is capable of finding a configuration that outperforms the GCC default `-O3`, and achieves this in orders of magnitude faster than comparable existing techniques. Although some of the other approaches can find configurations with a higher speedup in some cases, the performance of OSL can be considered reasonable. Therefore, it is a good alternative to existing approaches, particularly in settings where small algorithm runtimes are required, such as a compiler plugin.

OPEN SCIENCE

The dataset and the Python scripts we used to evaluate our approach can be obtained from https://github.com/AIG-ist-tugraz/OptimizationSpaceLearning.

ACKNOWLEDGMENTS

The work presented in this paper has been conducted within the scope of the OpenSpace project funded by the Austrian research promotion agency (FO999891127).

REFERENCES

[1] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Luc Lesoil, and Olivier Barais. 2019. *Learning very large configuration spaces: What matters for Linux kernel sizes.* Ph. D. Dissertation. Inria Rennes-Bretagne Atlantique.

[2] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, and Jean-Marc Jézéquel. 2020. Sampling effect on performance prediction of configurable systems: A case study. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 277–288.

[3] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2021. Learning software configuration spaces: A systematic literature review. *Journal of Systems and Software* 182 (2021), 111044.

[4] Juliana Alves Pereira, Pawel Matuszyk, Sebastian Krieter, Myra Spiliopoulou, and Gunter Saake. 2016. A feature-based personalized recommender system for product-line configuration. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 120–131.

[5] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 303–316.

[6] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)* 51, 5 (2018), 1–42.

[7] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. 2016. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 2 (2016), 1–25.

[8] Yosi Ben Asher, Gadi Haber, and Esti Stein. 2017. A Study of Conflicting Pairs of Compiler Optimizations. 52–58. https://doi.org/10.1109/MCSoC.2017.31

[9] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated reasoning on feature models. In *International Conference on Advanced Information Systems Engineering*. Springer, 491–503.

[10] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. 1998. Iterative compilation in a non-linear optimisation space. In *Workshop on profile and feedback-directed compilation*.

[11] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. 2021. Efficient compiler autotuning via bayesian optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1198–1209.

[12] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. 2012. Deconstructing iterative optimization. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 3 (2012), 1–30.

[13] Michael D Ekstrand, John T Riedl, Joseph A Konstan, et al. 2011. Collaborative filtering recommender systems. *Foundations and Trends® in Human–Computer Interaction* 4, 2 (2011), 81–173.

[14] Andreas Falkner, Alexander Felfernig, and Albert Haag. 2011. Recommendation technologies for configurable products. *Ai Magazine* 32, 3 (2011), 99–108.

[15] Damian Garber, Tamim Burgstaller, Alexander Felfernig, Viet-Man Le, Sebastian Lubos, Trang Tran, and Seda Polat-Erdeniz. 2023. Collaborative Recommendation of Search Heuristics For Constraint Solvers. In *ConfWS'23: 25th International Workshop on Configuration, Sep 6–7, 2023, Málaga, Spain*.

[16] Unai Garciarena and Roberto Santana. 2016. Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*. 1159–1166.

[17] Jingzhi Gong and Tao Chen. 2024. Deep Configuration Performance Learning: A Systematic Survey and Taxonomy. *arXiv preprint arXiv:2403.03322* (2024).

[18] Sumit Gulwani. 2010. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. 13–24.

[19] Maurice H Halstead. 1977. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc.

[20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[21] Erik Orm Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. 2023. Baco: A fast and portable Bayesian compiler optimization framework. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 19–42.

[22] Kenneth Hoste and Lieven Eeckhout. 2008. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. 165–174.

[23] Gourav Jain, Tripti Mahara, and Kuldeep Narayan Tripathi. 2020. A survey of similarity measures for collaborative filtering-based recommender system. In *Soft Computing: Theories and Applications: Proceedings of SoCTA 2018*. Springer, 343–352.

[24] Michael R Jantz and Prasad A Kulkarni. 2013. Performance potential of optimization phase selection during dynamic JIT compilation. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 131–142.

[25] James Kennedy and Russell Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, Vol. 4. IEEE, 1942–1948.

[26] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. 2007. IPOG: A general strategy for t-way software testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. IEEE, 549–556.

[27] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. 2016. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives* 3 (2016), 43–58.

[28] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[29] Kuldeep S Meel. 2022. Counting, Sampling, and Synthesis: The Quest for Scalability.. In *IJCAI*. 5816–5820.

[30] Edi Muškardin, Bernhard Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. 2022. AALpy: an active automata learning library. *Innovations in Systems and Software Engineering* 18 (03 2022), 1–10. https://doi.org/10.1007/s11334-022-00449-3

[31] Aravind Nair, Avijit Roy, and Karl Meinke. 2020. funcgnn: A graph neural network approach to program similarity. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–11.

[32] Jeho Oh, Don Batory, and Rubén Heradio. 2023. Finding near-optimal configurations in colossal spaces with statistical guarantees. *ACM Transactions on Software Engineering and Methodology* 33, 1 (2023), 1–36.

[33] Jeho Oh, Paul Gazzillo, and Don Batory. 2019. T-Wise Coverage by Uniform Sampling. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 84–87. https://doi.org/10.1145/3336294.3342359

[34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[35] Leslie Pérez Cáceres, Federico Pagnozzi, Alberto Franzin, and Thomas Stützle. 2018. Automatic configuration of GCC using irace. In *Artificial Evolution: 13th International Conference, Évolution Artificielle, EA 2017, Paris, France, October 25–27, 2017, Revised Selected Papers 13*. Springer, 202–216.

[36] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. Uniform sampling of sat solutions for configurable systems: Are we there yet?. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 240–251.

[37] Louis-Noel Pouchet. 2012. Polybench: The polyhedral benchmark suite. http://www.cs.ucla.edu/~pouchet/software/polybench/. Accessed: 2024.

[38] Diomidis Spinellis, Panos Louridas, and Maria Kechagia. 2016. The Evolution of C Programming Practices: A Study of the Unix Operating System 1973–2015. In *ICSE '16: Proceedings of the 38th International Conference on Software Engineering* (Austin, TX, USA), Willem Visser and Laurie Williams (Eds.). Association for Computing Machinery, New York, 748–759. https://doi.org/10.1145/2884781.2884799

[39] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. 2022. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 977–989.

[40] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I August. 2003. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 204–215.

[41] Steven R. Vegdahl. 1982. Phase Coupling and Constant Generation in an Optimizing Microcode Compiler. *SIGMICRO Newsl.* 13, 4 (oct 1982), 125–133. https://doi.org/10.1145/1014194.800942

[42] Andrew Walenstein, Mohammad El-Ramly, James R Cordy, William S Evans, Kiarash Mahdavi, Markus Pizka, Ganesan Ramalingam, and Jürgen Wolff von Gudenberg. 2007. Similarity in programs. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[43] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. 2020. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)* 53, 3 (2020), 1–34.

[44] Tzu-Tsung Wong. 2015. Performance evaluation of classification algorithms by k-fold and leave-one-out cross validation. *Pattern recognition* 48, 9 (2015), 2839–2846.

[45] L. Yu, Y. Lei, R. Kacker, and D. Kuhn. 2013. Acts: A combinatorial test generation tool. In *6th IEEE International Conference on Software Testing, Verification and Validation*. IEEE, Luxembourg, 370–375.

[46] Mingxuan Zhu, Dan Hao, and Junjie Chen. 2024. Compiler Autotuning through Multiple Phase Learning. *ACM Trans. Softw. Eng. Methodol.* (jan 2024). https://doi.org/10.1145/3640330 Just Accepted.

[47] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. 2001. SPEA2: Improving the strength Pareto evolutionary algorithm. *TIK report* 103 (2001).