

## MIT Open Access Articles

*Generalizable and interpretable  
learning for configuration extrapolation*

The MIT Faculty has made this article openly available. **Please share**  
how this access benefits you. Your story matters.

**Citation:** Ding, Yi, Pervaiz, Ahsan, Carbin, Michael and Hoffmann, Henry. 2021. "Generalizable and interpretable learning for configuration extrapolation." Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.

**As Published:** 10.1145/3468264.3468603

**Publisher:** Association for Computing Machinery (ACM)

**Persistent URL:** <https://hdl.handle.net/1721.1/142896>

**Version:** Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

**Terms of use:** Creative Commons Attribution 4.0 International License



# Generalizable and Interpretable Learning for Configuration Extrapolation

Yi Ding  
MIT CSAIL  
Cambridge, MA, USA  
ding1@csail.mit.edu

Michael Carbin  
MIT CSAIL  
Cambridge, MA, USA  
mcarbin@csail.mit.edu

Ahsan Pervaiz  
University of Chicago  
Chicago, IL, USA  
ahsanp@uchicago.edu

Henry Hoffmann  
University of Chicago  
Chicago, IL, USA  
hankhoffmann@cs.uchicago.edu

## ABSTRACT

Modern software applications are increasingly configurable, which puts a burden on users to tune these configurations for their target hardware and workloads. To help users, machine learning techniques can model the complex relationships between software configuration parameters and performance. While powerful, these learners have two major drawbacks: (1) they rarely incorporate prior knowledge and (2) they produce outputs that are not interpretable by users. These limitations make it difficult to (1) leverage information a user has already collected (e.g., tuning for new hardware using the best configurations from old hardware) and (2) gain insights into the learner's behavior (e.g., understanding why the learner chose different configurations on different hardware or for different workloads). To address these issues, this paper presents two configuration optimization tools, `GIL` and `GIL+`, using the proposed generalizable and interpretable learning approaches. To incorporate prior knowledge, the proposed tools (1) start from known configurations, (2) iteratively construct a new linear model, (3) extrapolate better performance configurations from that model, and (4) repeat. Since the base learners are linear models, these tools are inherently interpretable. We enhance this property with a graphical representation of how they arrived at the highest performance configuration. We evaluate `GIL` and `GIL+` by using them to configure Apache Spark workloads on different hardware platforms and find that, compared to prior work, `GIL` and `GIL+` produce comparable, and sometimes even better performance configurations, but with interpretable results.

## CCS CONCEPTS

• **Software and its engineering** → **Software configuration management and version control systems**; • **Computing methodologies** → *Machine learning approaches*.



This work is licensed under a Creative Commons Attribution International 4.0 License.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece  
© 2021 Copyright held by the owner/author(s).  
ACM ISBN 978-1-4503-8562-6/21/08.  
<https://doi.org/10.1145/3468264.3468603>

## KEYWORDS

Configuration, machine learning, generalizability, interpretability

### ACM Reference Format:

Yi Ding, Ahsan Pervaiz, Michael Carbin, and Henry Hoffmann. 2021. Generalizable and Interpretable Learning for Configuration Extrapolation. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*, August 23–28, 2021, Athens, Greece. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3468264.3468603>

## 1 INTRODUCTION

The increasing configurability of modern software makes it challenging for users to tune performance due to high complexity: tremendous configuration spaces and complicated interactions between these configuration parameters [39, 45, 49, 50]. Configurations have a large influence on application performance such as latency [42], throughput [3], and energy consumption [8]. As a result, tools to help tune application configurations for high performance have become a crucial yet challenging research area.

To configure software applications efficiently, machine learning (ML) approaches have been applied to model the complex relationships between configuration parameters and performance. Most prior work on incorporating ML approaches in software performance modeling is to first randomly sample assignments of configuration parameters for the application, measure the application's performance with the sampled parameters, train a learner on these samples, predict the performance for unsampled configurations, and then deploy the software in the configuration with the best predicted performance [1, 18, 23, 47]. The most sophisticated of these learners—e.g., neural networks [16], random forests [33], and Gaussian process regression [7, 29]—are *black-box*, meaning that users have no visibility into their internal workings [9].

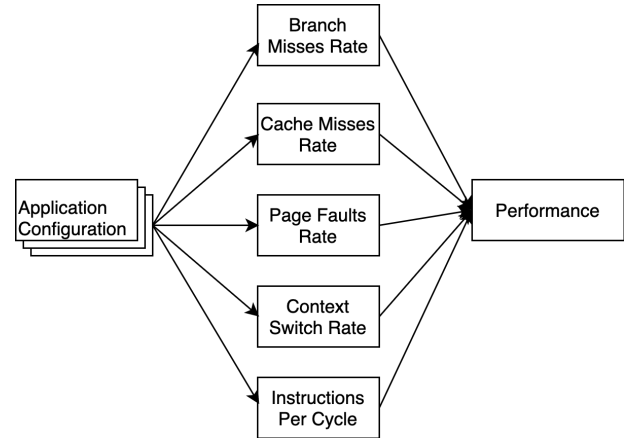
Although black-box ML approaches are effective at software performance modeling, there are two practical limitations to be addressed for efficient, optimal configuration search:

- **Difficulty incorporating prior knowledge.** Prior ML-based tools have little ability to leverage information a user has previously collected. Such tools require significant data collection: they must start from scratch by generating numerous random configurations and collecting their performance. Two specific scenarios are considered as follows:

- Scenario ①: Workload X runs slowly due to some faulty configuration settings [50, 52]. To find a better configuration, a user can randomly sample new configurations from scratch, run X for each to collect performance data, and build a new ML model. To avoid this tedious process, a user can instead leverage the known configurations—even though they are slow—to build a model that can extrapolate to find a better configuration. This strategy saves time as it does not need new initial training data collection.
- Scenario ②: Workload Y has been tuned to work at the optimal configuration on hardware A. Due to some hardware upgrade or scheduling issues, A is no longer available and Y has to be deployed on hardware B. Using prior approaches, users would randomly sample configurations on B, collect performance data, and build a ML model for configuration search [13, 20, 28]. An alternative is to incorporate prior knowledge by using configurations known to run fast on A and developing a model to extrapolate the configuration space from A to B. In §5.3, we show that incorporating prior knowledge achieves better performance than starting from scratch with random sampling.
- **Lack of interpretability.** Prior work evaluates their black-box models only for their performance prediction accuracy; it is difficult for users to interpret the prediction results [32]. Considering again the above two scenarios:
  - Scenario ①: Users hope to understand the underlying factors—from software applications to hardware resources—that cause the low performance and thus avoid the similar issues in the future [25, 43, 48].
  - Scenario ②: Users hope to interact with the learners and gain insights into why prior configurations are slow and how faster configurations are achieved. They also hope such knowledge to generalize across different hardware and workloads [47, 51].

To address these limitations, we present GIL and GIL+: configuration extrapolation tools that incorporate prior knowledge to achieve high performance and provide interpretability to advance human knowledge. To incorporate prior knowledge, GIL and GIL+ use existing information about configurations as initial training samples and then extrapolate to even higher performance configurations. To achieve interpretability, the key insight is to iteratively construct linear models that approximate the true nonlinear function such that the most important configuration parameters can be interpreted as a combination of both their weights in the linear model and the changes in those weights as the model is iteratively updated. To help understand the relationships between the application and the computer system on which it runs, GIL+ augments GIL with a hierarchical model that connects application-level configurations, low-level system metrics (e.g., context switches and cache misses), and the final application performance (see detailed definitions in §3.1). Additionally, we develop graphical tools to visualize these relationships to help users interpret learning results.

We implement GIL and GIL+, and evaluate them on Apache Spark workloads of HiBench [14], which is a widely used benchmark suite to evaluate data processing frameworks [53]. We run ten workloads covering a wide range of categories on three different hardware platforms. We consider two experimental settings corresponding to scenarios ① and ②, respectively. Our results show the following:



**Figure 1: GIL+ hierarchical model relating application configurations, low-level system metrics, and performance.**

- GIL and GIL+ achieve better extrapolation performance than random sampling, neural networks [16] and genetic algorithms [53], with results comparable to Bayesian optimization [33] (§5.1, ??).
- For all learners, incorporating prior knowledge achieves 2–15% performance improvements compared to starting from scratch with random sampling for initial training (§5.3).
- GIL+ enables the users to interact with the learners and interpret relationships between application-level configurations, low-level system metrics, and performance via visualization tools (§5.4).

In summary, this work makes the following contributions:

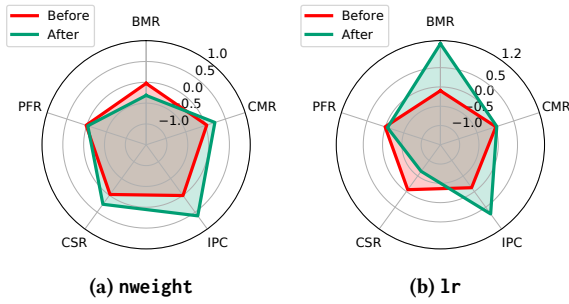
- Proposing GIL and GIL+, two configuration extrapolation tools that incorporate prior knowledge to avoid starting from scratch.
- Demonstrating the benefits of incorporating prior knowledge into learning systems for configuration extrapolation.
- Developing visualization tools for GIL and GIL+ to help users interpret learning results.
- Releasing code and data for GIL and GIL+ in <https://github.com/y-ding/gil>.

## 2 MOTIVATIONAL EXAMPLE

As an example of the value our tools, we study Apache Spark workloads from HiBench [14] on three different hardware from a public heterogeneous cloud system—the Chameleon Cloud Research Platform [21]. We use the names that Chameleon<sup>1</sup> uses for different hardware with details shown in Table 2, which includes three hardware microarchitectures—Skylake, Haswell, and Storage; the first two are compute servers, the last is optimized for disk bandwidth. We determine the best configuration for each workload and specific hardware by exhaustive search among our measurements.

GIL+ develops a hierarchical model connecting application-level configurations, low-level system metrics, and performance shown in Figure 1 (see detailed definitions in §3.1). In particular, GIL+ builds a linear model between low-level system metrics and performance, and for each low-level system metric, GIL+ builds a linear model between application-level configuration parameters and it. With the extra low-level system layer in-between, GIL+ makes it easier

<sup>1</sup><https://www.chameleoncloud.org/hardware/>



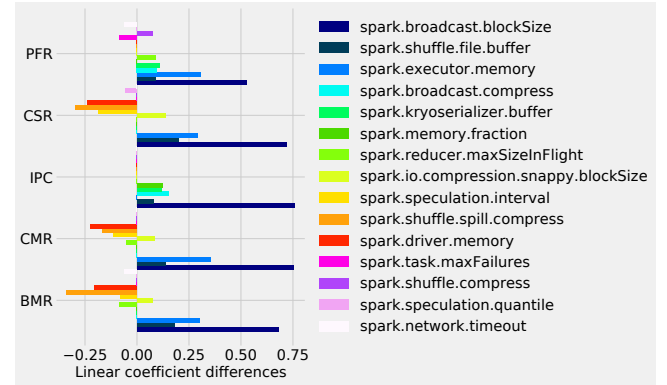
**Figure 2: Visualizing learned relationships between low-level systems metrics and performance.** The value on each axis shows how the corresponding low-level system metric influences performance when moving from Haswell (Before) to Storage (After).

for users to reason about the hardware and software cross-stack interactions as well as tune application-level configurations.

To illustrate GIL+’s behavior, we use the *nweight* workload from HiBench. Consider the case where Haswell is available first for deployment, and the high-performance configurations have been tuned for Haswell, either by human users or learners. Later, a new class of Storage machines with a small processor downgrade (slightly reduced clockspeed and last-level cache) and huge storage system upgrade (see details in Table 2) becomes available. A reasonable hypothesis would be that for the *nweight* workload, the upgraded storage system will outweigh the small downgrade in computing power. Thus when we deploy *nweight* on Storage, it is natural to start the high-performance Haswell configurations and run them directly on Storage.

Unfortunately, we find that the performance (throughput) can be more than 20% worse on Storage than Haswell, for the same configuration. Even if black-box learners from prior work are used to tune configurations using these available configurations, the best possible performance we find on Storage is still 10% worse than the best Haswell performance. Even worse, these black-box learners provide no interpretation of their results. Thus, at this point, the user might be wondering if Storage is fundamentally slower for *nweight*, or if the black-box learners simply need more time to search for a better configuration. With a black-box learner, there is no way for a user to gain insight into these questions. GIL+, on the other hand produces an interpretable model that a user can inspect.

Figure 2a shows the *nweight* interpretation results between application performance and low-level system metrics before and after applying GIL+ to tune application-level configurations on Storage. In other words, the before configurations are those that perform well on Haswell and the after configurations are those that perform well on Storage. The label for each axis in the radar chart represents a specific low-level system metric (see details in Table 3) and the value on each axis is the learned linear coefficient mapping from the low-level system metric to application-level performance. See §3.4 to know more about how we visualize the results. This chart shows that IPC (instructions per cycle) changes most—a sharp spike towards IPC—after applying GIL+, which indicates that higher IPC is associated with better performance on Storage



**Figure 3: Visualizing learned relationships between low-level system metrics and application-level configurations for *nweight* workload.** The magnitude of each bar shows how much the corresponding application-level configuration parameter changes each low-level system metric when moving from Haswell to Storage (with larger bars indicating larger effects). This enables users to relate the large effects from application-level configurations to the effects each low-level system metric has on performance in Figure 2.

and thus application-level configurations should be tuned to increase IPC. These results immediately provide insights into why Haswell performs better than Storage: Haswell’s faster clock and larger last-level cache are likely beneficial for an application that wants increased computational power. Being able to interpret the model gives us certain confirmation that Haswell is really faster than Storage for this workload. Furthermore, GIL+ can be used to discover workloads that behave similarly and infer the possible optimal deployment for new workloads. Figure 2b shows the radar chart for *1r*: it behaves similarly to *nweight*: both having sharp spikes towards IPC, which indicates that they would likely benefit from the same hardware. In other words, it may not be worthwhile running *1r* on Storage, because *nweight* is faster on Haswell than on Storage and it is likely that *1r* will be faster on Haswell as well. In contrast, black-box learners cannot report back this type of information to human users.

Nevertheless, no configuration optimizer can directly affect IPC, instead they must tune the application-level configurations to increase *nweight*’s IPC and, then, its performance. Therefore, GIL+ builds interpretable models that map application-level configurations and low-level system metrics to tune configuration parameters. Figure 3 shows the absolute linear coefficient differences between after and before applying GIL+ (starting from configurations that perform well on Haswell and arriving at configurations that perform well on Storage). This figure shows that, although Storage has only a small difference in processor, the application-level configurations that help achieve this throughput change greatly (i.e., `spark.broadCast.blockSize`). In contrast, black-box learners are not able to provide such insights to help users understand why they get different optimal configurations for different hardware. Since GIL+ starts from prior knowledge—configurations that work well on Haswell—and then finds high-performance configurations on Storage, this provides users insights into the differences between

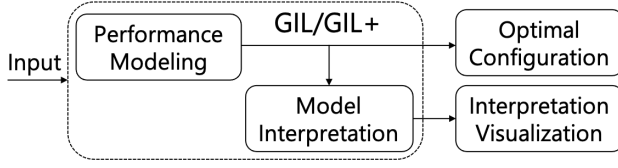


Figure 4: Workflow for GIL and GIL+ tools.

the configurations they know well and those that GIL+ finds. Additionally, our empirical results (see §5.3) show that extrapolating from known configurations can achieve better performance than starting from scratch with randomly sampling from the tremendous configuration space.

### 3 PROPOSED TOOLS

In this section, we will first define related terminologies, and then introduce two proposed tools GIL and GIL+, and then describe the visualization tools for model interpretation, and finally discuss the limitations of the proposed tools.

#### 3.1 Definitions

Figure 4 illustrates GIL and GIL+. The input includes a workload to optimize, a list of application-level configurations, and the low-level system metrics and performance data to be collected.

**Workload.** A workload is a configurable software application. We evaluate Apache Spark workloads (shown in Table 4), but in principle this could be any configurable software system whose quantitative behavior we want to understand and optimize.

**Configuration.** A configuration for a software application, represented by a  $p$ -dimensional vector where  $p$  is the number of configuration parameters:

$$\mathbf{x}_i = [x_{i1}, \dots, x_{ij}, \dots, x_{ip}], \quad (1)$$

where  $\mathbf{x}_i$  is the  $i$ -th configuration,  $x_{ij}$  is the value of the  $j$ -th configuration parameter in the  $i$ -th configuration. For instance,  $p$  is 20 corresponding to the 20 Apache Spark configuration parameters in our evaluation (§5) (shown in Table 1).

**Low-level system metric (LLSM).** LLSM is the quantifiable behavior of the underlying OS (e.g., context switches) and hardware (e.g., cache misses) that helps understand the application-level performance. These are represented as a  $d$ -dimensional vector assuming there are  $d$  metrics to measure and understand:

$$\mathbf{u}_i = [u_{i1}, \dots, u_{ij}, \dots, u_{id}], \quad (2)$$

where  $\mathbf{u}_i$  is the  $i$ -th low-level system metric vector,  $u_{ij}$  is the value of the  $j$ -th metric in the  $i$ -th low-level system metric vector. For example, in this work  $d$  is 5 as five low-level system metrics are studied as shown in Table 3.

**Performance.** Performance is the measured behavior for a workload on certain hardware. This work focuses on throughput, which is the rate of the total data delivered in the network [3].

Each configuration and its corresponding measurement—including low-level system metrics and performance—is a *sample*. GIL and GIL+ use these samples to build performance models to search the optimal configuration as an output. Specifically, GIL and GIL+ build

---

#### Algorithm 1 GIL performance model.

---

**Require:**  $X_{tr}$  ▷ Training configuration set  
**Require:**  $X_{te}$  ▷ Test configuration set  
**Require:**  $Y_{tr}$  ▷ Training performance set  
**Require:**  $k$  ▷ Sample budget at each round  
**Require:**  $T$  ▷ Number of rounds

- 1: **for** each round  $t = 1, \dots, T$  **do**
- 2:   Train model that maps configurations  $X_{tr}$  to performance  $Y_{tr}$ .
- 3:   Use above trained model to predict performance on  $X_{te}$ .
- 4:   Select  $k$  configuration set  $X_t$  with best predicted performance.
- 5:   Run workload to get true performance  $Y_t$  for  $X_t$ .
- 6:   Update training configuration set  $X_{tr} \leftarrow X_{tr} \cup X_t$ .
- 7:   Update training performance set  $Y_{tr} \leftarrow Y_{tr} \cup Y_t$ .
- 8:   Update test configuration set  $X_{te} \leftarrow X_{te} \setminus X_t$ .
- 9:   Train model that maps configurations  $X_{tr}$  to  $Y_{tr}$ .
- 10:   Use the trained model to predict performance for test configuration set  $X_{te}$ .
- 11:   Interpret results using tools in §3.4.
- 12: **Output:** configuration  $\mathbf{x}^*$  with the best predicted performance and interpretation results visualized by radar and bar charts.

---

a series of models relating application-level configurations to performance, a series of models relating the low-level system metrics to performance, and a series of models relating application-level configurations to each low-level system metric. Then, the coefficients learned from these models are passed to model interpretation to visualize the results as another output. Next, we will elaborate the performance models and interpretation visualization.

#### 3.2 GIL Performance Model

To improve performance accuracy and promote search space exploration, GIL iteratively interacts with the workload by sequentially learning an unknown function  $f$ —modeling the relationships between application configuration and performance—with a sample budget. At each iteration, limited configurations are selected to update  $f$ . After all iterations are completed, GIL uses the learned  $f$  to evaluate the remaining (untested) configurations to find the one with the best predicted performance. In this process, two challenges (CHs) need to be addressed:

**CH1:** Designing a model to learn the unknown function  $f$ .

**CH2:** Formulating a query strategy to select the samples.

We will address these two CHs as follows.

**3.2.1 Learning Linear Functions.** To learn the unknown function  $f$ , GIL uses a linear regression model [10]. GIL chooses linear model over other nonlinear models, such as neural networks [12] and Gaussian process regression [7, 29], due to several advantages:

- *Better extrapolation ability.* Although nonlinear models usually achieve higher *overall* prediction accuracy than linear models, they are more likely to overfit the training set, which leads to poor extrapolation performance when facing unseen data points [6]. In contrast, linear models are less likely to overfit and thus generalize better on unseen data.
- *Interpretability.* Linear models are naturally interpretable: the linear coefficients quantify the relationships between independent metrics and performance: a positive coefficient indicates positive correlation between a metric and performance, while a negative coefficient indicates negative correlation. The coefficients' magnitudes represent the strength of these relationships.



For the  $i$ -th configuration  $\mathbf{x}_i$ , assuming its measured performance is  $y_i$ , the linear regression model will be:

$$y_i = \beta_1 x_{i1} + \beta_2 x_{i2} + \dots + \beta_p x_{ip} + \epsilon_i, \quad (3)$$

where  $\beta_1, \dots, \beta_p$  are coefficients corresponding to  $p$  configuration parameters, and  $\epsilon_i$  is the noise (usually assumed to be a Gaussian). The linear model is solved via ridge regression, which is a regularized method to avoid overfitting [10]. Thus GIL addresses CH1 by iteratively estimating the unknown function with linear models.

**3.2.2 Active Learning.** GIL applies *active learning* to iteratively query samples over successive rounds. At each round, GIL collects samples that are intended to improve the model predictions for the best configurations. After evaluating the new samples, the model is updated using these samples and then the next round begins. As such, the key is to come up with a query function that converges quickly so that GIL can find the optimal configuration across spaces with limited samples. This is not trivial because the model needs to balance the tradeoff between exploration—avoiding local optima in training space—and exploitation—improving the predictions for the highest performance configurations in new space.

Given the base learner is linear, GIL optimizes *locally* by querying samples with the best predicted performance at each step: for each step  $t$ , assume the predictive function is  $f_t$ , then the optimizer searches for a subset  $S$  with  $k$  configurations such that:

$$S = \arg \max_{s \in S, |S|=k} \sum_s f_t(s). \quad (4)$$

This query strategy selects a subset of configurations with the best predicted performance at each round. Intuitively, picking configurations with the best performance is more likely to improve the prediction accuracy for the best performance configurations in the next round. *With this query strategy, GIL addresses CH2 by searching the optimal configuration as rapidly as possible.* Algorithm 1 summarizes the procedures for GIL performance model.

### 3.3 GIL+ Performance Model

GIL interprets the relationships between the application-level configuration parameters and ultimate performance by quantifying the learned coefficient for each application-level configuration parameter. To further benefit from interpretation, we propose a hierarchical performance model that relates application-level configuration parameters to performance through low-level system metrics. In particular, we hope to understand not just how each application parameter influences performance, but also how that influence manifests through low-level system metrics like cache misses or context switches.

To accomplish this goal, we present GIL+, illustrated in Figure 1. We consider five low-level system metrics in this paper (see details in Table 3). In principle, GIL+ could be applied to any set of low-level system metrics. We incorporate this extra layer as follows.

Algorithm 2 summarizes the procedures for GIL+. Instead of directly building models mapping from application configurations to performance, GIL+ first builds an model that maps from low-level system metrics to performance. Then, GIL+ obtains the low-level system metric vector  $[u_1, \dots, u_5]$  with the best predicted performance from test low-level system metric set  $U_{te}$ , as in line 3. Next, for  $i$ -th low-level system metric  $U^i$ ,  $i = 1 \dots 5$ , GIL+ builds a model

#### Algorithm 2 GIL+ hierarchical performance model.

---

**Require:**  $X_{tr}$  ▷ Training configuration set  
**Require:**  $X_{te}$  ▷ Test configuration set  
**Require:**  $U_{tr}$  ▷ Training low-level system metric set  
**Require:**  $U_{te}$  ▷ Test low-level system metric set  
**Require:**  $Y_{tr}$  ▷ Training performance set  
**Require:**  $d$  ▷ Number of low-level system metrics (LLSMs)  
**Require:**  $k$  ▷ Sample budget for each low-level system metric at each round  
**Require:**  $T$  ▷ Number of rounds

---

- 1: **for** each round  $t = 1, \dots, T$  **do**
- 2:   Train model that maps LLSMs  $U_{tr}$  to performance  $Y_{tr}$ .
- 3:   Get LLSM vector  $[u_1, \dots, u_d]$  with best predicted performance from  $U_{te}$ .
- 4:   **for** each LLSM  $U^i$ ,  $i = 1, \dots, d$  **do**
- 5:     Train model that maps  $X_{tr}$  to  $U_{tr}^i$ .
- 6:     Use trained model to predict on  $U_{te}^i$ .
- 7:     Select  $k$  configurations  $X_{tr}^i$  with closest predicted  $i$ -th LLSM values to  $u_i$ .
- 8:     Run workload to get true LLSM value  $U_{tr}^i$  and performance  $Y_{tr}^i$  for  $X_{tr}^i$ .
- 9:     Update training configuration set  $X_{tr} \leftarrow X_{tr} \cup X_{tr}^i$ .
- 10:    Update training LLSM set  $U_{tr} \leftarrow U_{tr} \cup U_{tr}^i$ .
- 11:    Update training performance set  $Y_{tr} \leftarrow Y_{tr} \cup Y_{tr}^i$ .
- 12:    Update test configuration set  $X_{te} \leftarrow X_{te} \setminus X_{tr}^i$ .
- 13:   Train model that maps configurations  $X_{tr}$  to  $Y_{tr}$ .
- 14:   Use the trained model to predict performance for test configuration set  $X_{te}$ .
- 15:   Interpret results using tools in §3.4.
- 16: **Output:** configuration  $\mathbf{x}^*$  with the best predicted performance and interpretation results visualized by radar and bar charts.

---

that maps from application configurations to  $U^i$ . Then, GIL+ selects top  $k$  application configurations that have the closest predicted  $i$ -th low-level system metric values to  $u_i$ , and these  $k$  configurations and corresponding measurements will be queried for the next round, as in line 7-8. After all training and test sets are updated, GIL+ trains the model on the final training configuration and performance sets, and uses it to evaluate the remaining test configurations, picking the one with the best predicted performance as the output, as in line 13-14. The intuition of this hierarchical model is to select application configurations that induce the nearest low-level system metric values producing the best predicted performance.

### 3.4 Interpretation by Visualization

We interpret results from performance models via visualization. We focus on GIL+ because of its extra low-level system metric layer, which makes it possible to investigate the cross-stack interactions between system and application levels. We visualize the learned relationships between (1) performance and low-level system metrics and (2) low-level system metrics and configuration parameters.

We use radar charts to depict the relationships between performance and low-level system metrics and how they change before and after applying GIL+. These charts visualize the differences between the initial, user-provided configurations and the final relationships that GIL+ learns. If users provide configurations that are known to work well on different hardware, then these charts will visualize the differences in the influence of these initial configurations and those GIL+ learned. We choose radar charts because they can show multiple metrics simultaneously and quickly compare different values in each axis. The label for each axis represents a specific low-level system metric and the value on each axis is the learned linear coefficient mapping from low-level system metric to performance. The changes are visualized with different colors.

We use bar charts to illustrate the relationships between low-level system metrics and application configuration parameters. We

choose bar charts because there are multiple low-level system metrics and we need to compare each with their relationships with configuration parameters. Take *nweight* workload in Figure 3 as an example. The y-axis represents each low-level system metric, and the x-axis represents the learned absolute linear coefficient differences before and after applying GIL+. Each bar represents different configuration parameters, and their direction and length correspond to the x-axis. With bar charts, we can interpret the learned relationship between low-level system metrics and configuration parameters, and how their influences to each low-level system metric change.

### 3.5 Discussion and Limitations

GIL and GIL+ use linear models as base learners to achieve interpretability, which would lose a slight amount of overall prediction accuracy compared to using more sophisticated black-box learners such as neural networks [12], gradient boosting trees [4], and Gaussian process regression [7, 35]. Moreover, these black-box learners are more likely to overfit the training data and make conservative judgment in exploring new space. To compensate for the loss of prediction accuracy, GIL and GIL+ employ an iterative learning paradigm to allow the model to correct itself every time there is an error [27]. Meanwhile, Bayesian optimization updates the model iteratively using a black-box learner—Gaussian process regression, and usually achieves high prediction accuracy [38], which is also demonstrated in our evaluation in Section 5. While powerful, it is uninterpretable and thus does not provide insights into how the highest performance configuration is reached. To sum up, GIL and GIL+ are practical trade-offs between accuracy and interpretability.

## 4 EXPERIMENTAL METHODOLOGY

This section describes our experimental setup and evaluation methodologies to demonstrate the effectiveness of GIL and GIL+ at configuration extrapolation in multiple workloads, hardware, and settings.

### 4.1 Systems

**4.1.1 Software.** We use Apache Spark 2.2.3 [54]<sup>2</sup> as our software distributed computing framework. Each experiment has a server node and four worker nodes. We choose a wide range of configuration parameters that reflect significant Spark properties categorized by shuffle behavior, data compression and serialization, memory management, execution behavior, networking, and scheduling. Table 1 shows the total 20 parameters in detail. We use similar parameters to prior work [53] but not the same set because Spark has actually reduced the number of user visible configuration parameters—precisely because configuring them is challenging.

**4.1.2 Hardware.** We run experiments on a public cloud computing system: the Chameleon Cloud Research Platform [21]. We use the names that Chameleon<sup>3</sup> uses for three Intel x86 processors shown in Table 2. We collect five low-level system metrics that have been shown to influence application performance [24]. Table 3 shows the five low-level system metrics in detail.

<sup>2</sup><https://spark.apache.org/docs/2.2.3/configuration.html>

<sup>3</sup><https://www.chameleoncloud.org/hardware/>

### 4.2 Workloads

We select ten Apache Spark workloads from the HiBench<sup>4</sup> big data benchmark suite [14] with details shown in Table 4. These workloads cover various domains including microbenchmarks (micro), machine learning (ML), websearch, and graph analysis; and they exhibit a wide range of resource usage. For ten workloads with 2000 configurations per workload on three hardware platforms, it took four weeks of computing time to collect all these data for experimental evaluation in this paper.

### 4.3 Points of Comparisons

We compare the following approaches:

- **DEFAULT:** the default application-level configuration provided by the Apache Spark developers.
- **RS:** randomly sample configurations and select the best with a linear regression performance model.
- **NN:** a design space exploration method with neural network and intelligent sampling proposed in [16].
- **DAC:** a configuration parameter tuning approach with the ensemble tree model and genetic algorithm proposed in [53].
- **BO:** Bayesian optimization for configuration tuning [33].
- **GIL:** generalizable and interpretable learning relating application configurations and performance.
- **GIL+:** generalizable and interpretable learning relating application configurations, low-level system metrics, and performance.
- **OPT:** exhaustive search for the true optimal configuration over the entire measurements.

RS, NN, and DAC are non-iterative learning methods that train only once, while BO, GIL, and GIL+ are iterative methods that train multiple rounds until the sample budget is met. NN, DAC, and BO use black-box models, while RS, GIL, and GIL+ are interpretable. All parameters for these algorithms used in our experiments are selected via cross validations.

### 4.4 Evaluation Metric

For each workload, we compare the Relative Performance (RP) between the best performance (throughput) found from the learning-based configuration search methods and the true optimal performance (found through exhaustive search over our measurements):

$$RP = 100\% * \left| \frac{Y_{pred} - Y_{opt}}{Y_{opt}} \right|, \quad (5)$$

where  $Y_{pred}$  is the best performance from the above methods and  $Y_{opt}$  is the optimal performance. Lower RP is better.

### 4.5 Evaluation Methodology

We randomly generate 2000 application-level configurations for each workload, which is a commonly used size in prior work [53]. For each workload, we run it at each configuration on different hardware to record their performances. We divide the total configurations into three levels: low, modest, and high performance with corresponding ratios [0.5, 0.3, 0.2]; e.g., the high-performance configurations are the top 20% of the measured configurations. The optimal configuration for each workload on each hardware is obtained via exhaustive search over our measurements. We then use

<sup>4</sup><https://github.com/Intel-bigdata/HiBench>

**Table 1: Details of the 20 Apache Spark 2.2.3 configuration parameters.**

Configuration parameter	Range	Description
spark.reducer.maxSizeInFlight	24–128	Maximum size of map outputs to fetch simultaneously from each reduce task, in MB.
spark.shuffle.file.buffer	24–128	Size of the in-memory buffer for each shuffle file output stream, in KB.
spark.shuffle.sort.bypassMergeThreshold	100–1000	Avoid merge-sorting data if there is no map-side aggregation.
spark.speculation.interval	100–1000	How often Spark will check for tasks to speculate, in millisecond.
spark.speculation.multiplier	1–5	How many times slower a task is than the median to be considered for speculation.
spark.speculation.quantile	0–1	Percentage of tasks which must be complete before speculation is enabled.
spark.broadcast.blockSize	2–128	Size of each piece of a block for TorrentBroadcastFactory, in MB.
spark.io.compression.snappy.blockSize	24–128	Block size used in snappy, in KB.
spark.kryoserializer.buffer.max	24–128	Maximum allowable size of Kryo serialization buffer, in MB.
spark.kryoserializer.buffer	24–128	Initial size of Kryo’s serialization buffer, in KB.
spark.driver.memory	6–12	Amount of memory to use for the driver process, in GB.
spark.executor.memory	8–18	Amount of memory to use per executor process, in GB.
spark.network.timeout	20–500	Default timeout for all network interactions, in second.
spark.locality.wait	1–10	How long to launch a data-local task before giving up, in second.
spark.task.maxFailures	1–8	Number of task failures before giving up on the job.
spark.shuffle.compress	false, true	Whether to compress map output files.
spark.memory.fraction	0–1	Fraction of (heap space - 300 MB) used for execution and storage.
spark.shuffle.spill.compress	false, true	Whether to compress data spilled during shuffles.
spark.broadcast.compress	false, true	Whether to compress broadcast variables before sending them.
spark.memory.storageFraction	0.5–1	Amount of storage memory immune to eviction.

**Table 2: Details of the hardware platforms.**

	Skylake	Haswell	Storage
Processor	Gold 6126	E5-2670	E5-2650
RAM size	192 GB	128 GB	64 GB
# of Threads	48	48	40
Clockspeed	2.6 GHz	3.1 GHz	3.0 GHz
L3 cache	19.25 MB	30 MB	25 MB
Memory speed	2.666 GHz	2.133 GHz	2.133 GHz
# Mem channels	6	4	4
Network speed	10 GbE	0.1 GbE	10 GbE
Disk vendor	Samsung	Seagate	Seagate
# Disks	1	1	16
Disk bandwidth	6 Gb/s	6 Gb/s	24 Gb/s

**Table 3: Details of the five low-level system metrics.**

Abbr.	Low-level metrics	Description
BMR	Branch misses rate	# branch misses/ # total branch misses
CMR	Cache misses rate	# cache misses/ # total cache misses
CSR	Context switch rate	# context switch/ # cpu cycles
PFR	Page faults rate	# page faults/ # cpu cycles
IPC	Instruc. per cycle	# instruction / # cpu cycles

the methods mentioned above to search for the best configuration for each workload on different hardware. Since our goal is to examine the extrapolation ability of each algorithm, we consider the following two experimental settings:

- **low2high**: training set only has configurations of low performance, and test set has configurations with a wide range of performance. This setting corresponds to scenario ① in §1; i.e.,

**Table 4: Details of the ten HiBench workloads.**

Workload	Data size	Workload	Data size
wordcount	32 GB	lr	8 GB
terasort	3.2 GB	linear	48 GB
als	0.6 GB	rf	0.8 GB
bayes	19 GB	pagerank	1.5 GB
kmeans	20 GB	nweight	0.9 GB

this represents the case where users start from configurations of low performance and aim to extrapolate to high performance .

- **mod2high**: training set only has configurations of modest performance that run fast from a different hardware, and test set has configurations with a wide range of performance. For instance, if the training set has configurations that run fast on Haswell, but run modestly on Skylake, the search phase will evaluate all configurations in test set to pick the one predicted to run fastest on Skylake. Since we have 3 hardware, we have 6 training-test pairs. This setting corresponds to scenario ② in §1; i.e., this represents the case where users have found configurations of high performance for one hardware and attempt to use those as a starting point to optimize for a different hardware.

In Algorithm 1 and 2,  $T$  and  $k$  are set as small numbers to demonstrate that GIL and GIL+ can produce comparable and better results on a small sample budget than those methods without intelligent sampling (e.g., RS). Since the collected dataset size for each hardware is 2000, we choose 20% and 10% for low2high and mod2high—i.e., 400 and 200—which are much smaller than thousands used in prior work [53]. low2high requires more labeled data for training because it extrapolates larger space than mod2high does.  $k$  is set as 20. We report results averaged over 5 runs with different seeds.



## 5 EXPERIMENTAL EVALUATION

We examine the following research questions (RQs):

- **RQ1:** How good are GIL and GIL+ at extrapolating from configurations of low performance to high performance (low2high)?
- **RQ2:** How good are GIL and GIL+ at extrapolating from configurations of modest performance to high performance (mod2high)?
- **RQ3:** Does incorporating prior knowledge improve extrapolation performance compared to starting from scratch?
- **RQ4:** What can we interpret from visualization in case studies?

### 5.1 RQ1: How good are GIL and GIL+ at extrapolation from configurations of low performance to high performance?

We examine the extrapolation results from configurations of low performance to high performance in low2high setting. Figure 5 shows the relative performance (RP) results for each method. The strip label on the right for each row of bar charts represents the hardware platform for the experiments. The x-axis represents each workload, and the y-axis represents RP. Lower is better (closer to optimal). The last column, HarMean, shows the harmonic mean results over all workloads, which are also quantified in Table 5.

**Table 5: Harmonic mean results over all workloads of each hardware for low2high. The last column HarMean is the harmonic mean over three hardware.**

	Skylake	Haswell	Storage	HarMean
DEFAULT	43%	29%	25%	31%
RS	22%	11%	11%	13%
NN	22%	17%	13%	16%
DAC	36%	18%	19%	22%
BO	14%	1%	2%	2%
GIL	13%	7%	7%	8%
GIL+	10%	7%	5%	7%

For non-iterative learning methods, RS is generally better than NN and DAC, which is a bit counter-intuitive as we would think that linear models are not as powerful as black-box models. However, as discussed in §3.2.1, despite perhaps overall lower prediction accuracy, linear models are less likely to overfit and thus promote extrapolation. In contrast, black-box models such as neural networks and ensemble trees are more likely to overfit and make less accurate predictions for unseen data. For iterative learning methods, BO is slightly (5 percentage points) better than GIL and GIL+ because black-box models generally have higher prediction accuracy than linear models. Overall, iterative learning methods are better than non-iterative ones because they encourage extrapolation in the iterative process. *These results demonstrate that GIL and GIL+'s iterative learning can provide performance comparable to the best black-box methods and overcome faulty (or extremely low-performance) configurations.*

### 5.2 RQ2: How good are GIL and GIL+ at extrapolating from configurations of modest performance to high performance?

We examine the extrapolation results from configurations of modest performance to high performance in mod2high setting. Figure 6, 7, and 8 show the relative performance (RP) for each method on three target hardware platforms, respectively. In each figure, the strip label on the right for each row of bar charts represents the hardware platform where the starting configurations are from. The x-axis represents each workload, and the y-axis represents RP for the target hardware. Lower is better (closer to optimal). The last column, HarMean, shows the harmonic mean results over all workloads, which are also quantified in Table 6.

**Table 6: Harmonic mean results over all workloads of each hardware for mod2high. The last column HarMean is the harmonic mean over three hardware.**

	Skylake	Haswell	Storage	HarMean
DEFAULT	43%	29%	25%	31%
RS	17%	10%	9%	11%
NN	19%	13%	9%	13%
DAC	22%	14%	10%	14%
BO	7%	1%	2%	2%
GIL	2%	2%	6%	2%
GIL+	7%	6%	3%	4%

The results analyzed in the low2high setting almost hold for the mod2high setting. RS is better than NN and DAC, and BO is almost comparable to GIL and GIL+. Overall, BO, GIL and GIL+ are better than RS, NN, and DAC due to their iterative learning paradigm. *These results demonstrate that GIL and GIL+ are comparable to the best black-box learners. Different from prior methods, GIL and GIL+ can benefit from starting from user suggested configurations, which is evidence that incorporating human knowledge is beneficial—it gets GIL and GIL+ on par with the best black-box learner.*

### 5.3 RQ3: Does incorporating prior knowledge improve extrapolation performance compared to starting from scratch?

As further evidence of the benefits of extrapolating from prior knowledge, we compare starting from scratch with random sampling to incorporating prior knowledge for initial training. Figure 9 shows the aggregated improvements from incorporating prior knowledge over starting with random sampling across all workloads in the mod2high setting (higher is better). Overall, all learners benefit from incorporating prior knowledge with improvements from 2% to 15% in harmonic mean, where GIL and GIL+ both have 3% improvement. Starting from scratch achieves lower performance because random sampling over the whole space loses initial direction for extrapolation, while a tighter cluster of samples provides a clearer direction to pursue next, and reduces the chance of getting stuck in local optima. *These results demonstrate the efficacy of GIL and GIL+ by incorporating prior knowledge for initial training.*

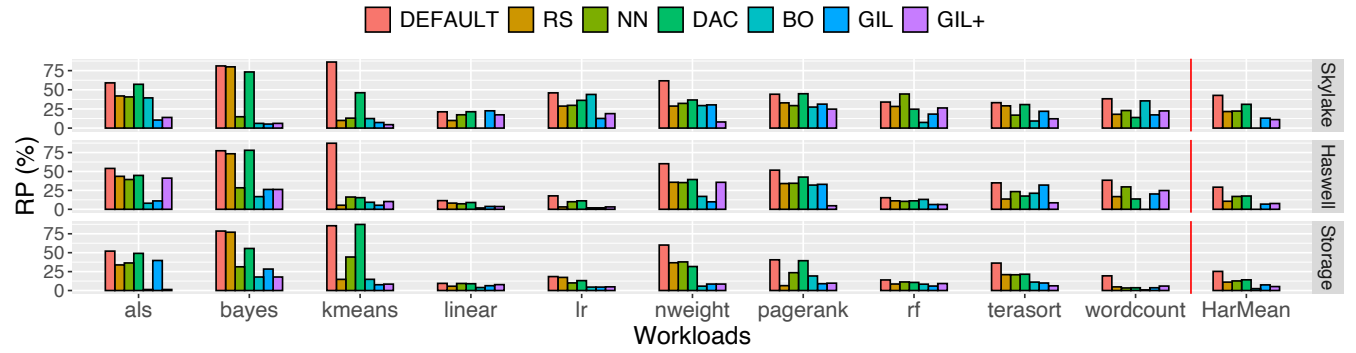


Figure 5: Relative performance (RP) of each method in low2high setting for different hardware (lower is better).

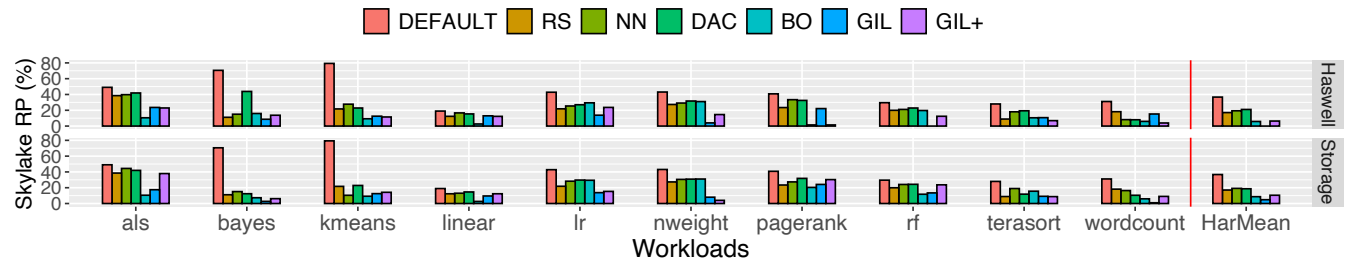


Figure 6: Relative performance (RP) of each method in mod2high setting when Skylake is the target hardware (lower is better).



Figure 7: Relative performance (RP) of each method in mod2high setting when Haswell is the target hardware (lower is better).

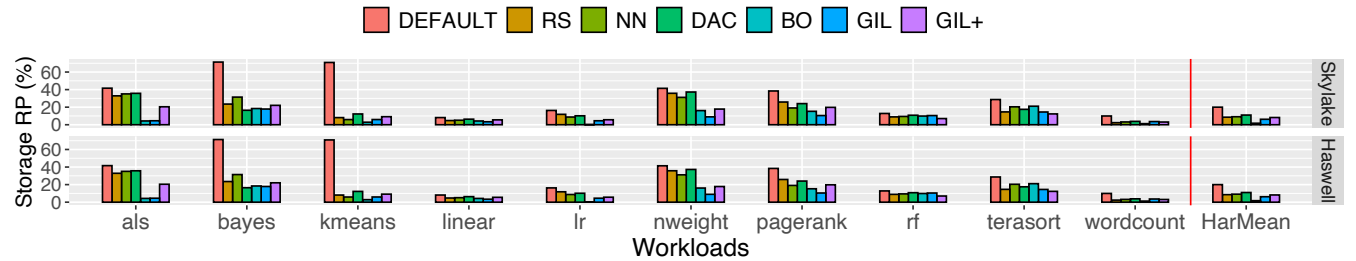


Figure 8: Relative performance (RP) of each method in mod2high setting when Storage is the target hardware (lower is better).

#### 5.4 RQ4: What can we interpret from visualization in case studies?

To demonstrate the scope of our visualization tools for interpretation, we use two case studies in different applications: cause interpretation and deployment prediction.

**5.4.1 Cause Interpretation.** We examine the GIL+'s ability to interpret the causes for low performance by visualizing the coefficients of the learned relationships mapping from low-level system metrics to performance, and mapping from application configurations to low-level system metrics. Figure 10 shows two workloads running on Skylake and how their coefficients change from low performance

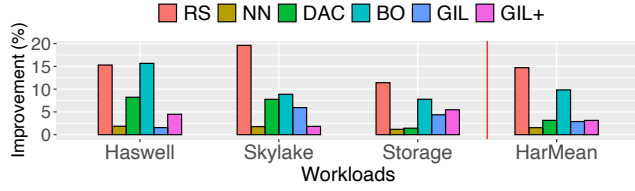


Figure 9: Improvements from incorporating prior knowledge over starting with random sampling (higher is better).

to high performance. The common thing observed in two radar charts is that both workloads need higher BMR and CMR for higher performance. This is counter-intuitive because higher BMR and CMR conventionally lead to lower performance. To interpret the radar charts correctly, we need to read every axis together. It is not that increasing BMR and CMR increases performance; rather increasing BMR and CMR together, while reducing CSR is key to achieving better performance.

Figure 11 shows the corresponding bar charts for the absolute learned coefficient differences from application configuration parameters on each low-level system metric. We can see that the ways to get similar system-level metrics changes are different for different workloads via different application-level configuration parameter changes. For linear, the influence magnitudes of BMR and CMR from configuration parameters do not change much. For 1r, the largest influence changes of BMR and CMR are from `spark.memory.fraction`. *These results demonstrate GIL+’s ability to interpret causes for low performance by discovering the relationships between application configurations, low-level system metrics, and performance.*

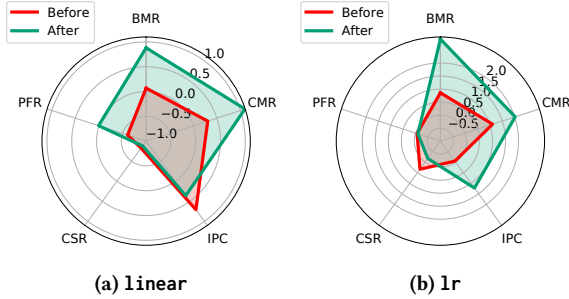
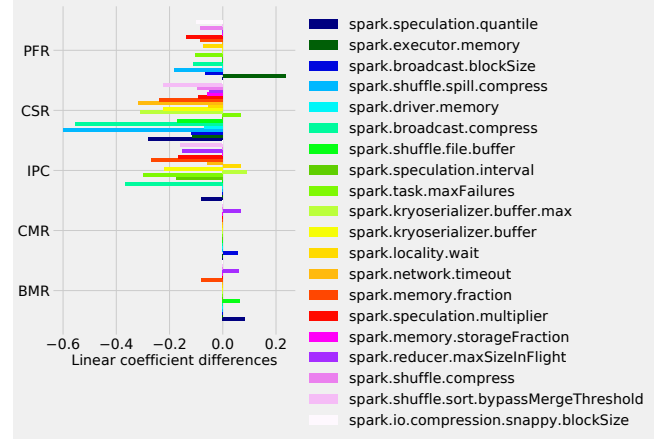
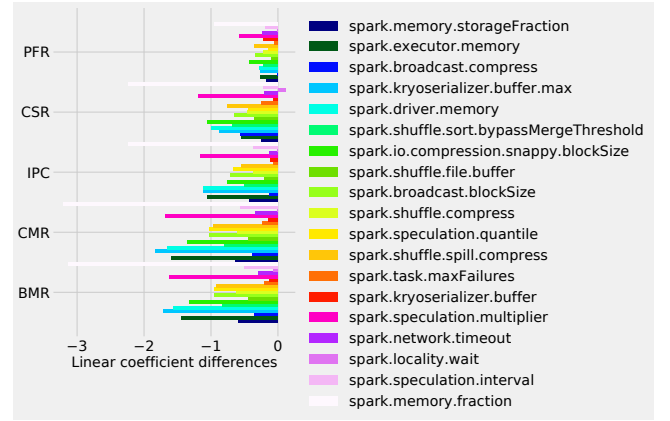


Figure 10: Visualizing learned relationships between low-level systems metrics and performance. The value on each axis shows how the corresponding low-level system metric influences performance when moving from low performance (Before) to high performance (After) on Skylake.

**5.4.2 Deployment Prediction.** We show that the learned coefficients of the model mapping from low-level system metrics to performance can predict the deployment benefits across hardware. Figure 12 shows two radar charts, where for each workload, configurations running fast on Haswell and their performance on Storage are used for initial training to obtain the *before* coefficients. These coefficients represent how fast configurations found on Haswell behave when they are on Storage. Then, we apply GIL+ to iteratively



(a) linear

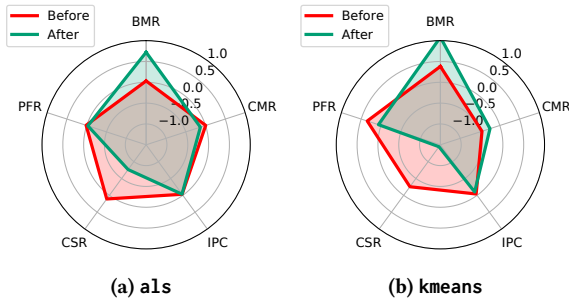


(b) 1r

Figure 11: Visualizing learned relationships between low-level system metrics and application-level configurations. The magnitude of each bar shows how much the corresponding application-level configuration parameter changes each low-level system metric when moving from low performance to high performance (with larger bars indicating larger effects). This enable users to relate the large effects from application-level configurations to the effects each low-level system metric has on performance in Figure 10.

extrapolate the configurations that can run fast on Storage until the sample budget is met. The *after* coefficients represent the best configurations found on Storage behave.

Figure 12a and 12b show that the two workloads `als` and `kmeans` both have sharp spikes away from CSR, meaning that they both benefit from decreased influence from CSR. Thus, we predict that both workloads have same best hardware deployment. This prediction is validated by the true data that the hardware choice for the best performance of both workloads is Storage over Haswell. This also tells users that there is no need to tune both workloads separately on both hardware to find which hardware will be a better deployment because it is likely that they will have the same best hardware deployment due to their similar low-level system



**Figure 12: Visualizing learned relationships between low-level systems metrics and performance. The value on each axis shows how the corresponding low-level system metric influences performance when moving from Haswell (Before) to Storage (After).**

behavior. These results show that our interpretation tool enables the users to interact with the learners and acquire the knowledge that generalizes across different hardware workloads.

## 6 RELATED WORK

**ML for configuration management.** ML techniques confront the complexity of software configuration management [13, 37, 40, 49]. ML is generally incorporated into configuration tuning via performance modeling [20, 20, 40]; i.e., the process of learning a function mapping application-level configuration parameters to quantifiable behavior (e.g., latency, throughput, energy consumption) [3, 8, 15, 30, 31, 42, 55]. For instance, ConEx uses evolutionary Markov Chain Monte Carlo sampling to find high-performance configurations for big data systems [23]. OtterTune uses Gaussian processes to tune database configurations [42]. OpenTuner uses an ensemble of genetic algorithms to tune configuration parameters for computer programs [2]. More recently, there has been a growing interest in tuning ML systems themselves using other black-box optimization techniques [11, 26, 46].

**Transfer learning.** Transfer learning is a common approach to incorporate prior knowledge, which gains knowledge from one problem and applies it to a different but related problem [34]. For configurable software systems, the knowledge can be transferred across different workloads and hardware environments to reduce data collection efforts [17]. For instance, a cost-aware transfer learning method uses Gaussian process to transfer knowledge from simulators to real systems [19]. L2S uses active learning to select samples for knowledge transfer to a new domain [18]. BEETLE uses a discovery step to identify the most relevant source from multiple sources of data [22]. However, these works are fundamentally different from our proposal in that they use black-box models, in which the knowledge transferred is not interpretable by users [36]. In contrast, GIL and GIL+ not only incorporate prior knowledge to improve application performance, but produce interpretable results to help expand that knowledge.

**Interpretability.** There has been an emerging line of work on interpreting software systems. Complex proposes a white-box performance model based on a data-flow analysis [44], which does not

incorporate prior knowledge or use learning. Valov et al use linear models to transfer knowledge across different hardware environments [41], but they directly use the model learned from the source on the target software applications, and thus do not extrapolate to improve results. Our proposed methods use linear models to provide users an interpretation of the interactions between application-level configurations, low-level system metrics, and performance [5]. Our proposed tools GIL and GIL+ combine the advantages of transfer learning and interpretability to improve configuration extrapolation results and give feedback to users.

## 7 CONCLUSION

This paper introduces two configuration extrapolation tools, GIL and GIL+, that incorporate prior knowledge and produce interpretable results. These tools also produce a graphical interpretation of how they arrived at the highest performance application configuration. Our results show that GIL and GIL+ achieve application performance comparable to the best black-box learner, and outperform those starting from scratch with random sampling for initial training, which demonstrates the benefits of incorporating prior knowledge into the learning process. Additionally, the graphical visualization tools enable users to interact with the learners and interpret relationships between application-level configurations, low-level system metrics, and performance. We hope this work can inspire software engineering researchers to consider prior knowledge and model interpretability when applying machine learning techniques to performance modeling.

## ACKNOWLEDGMENTS

We are grateful to Alex Renda who read the early draft of this work and provided extremely valuable feedback. We thank the anonymous reviewers for their helpful feedback to improve the final paper. Yi Ding’s work is supported by the National Science Foundation under Grant 2030859 to the Computing Research Association for the CIFellows Project. Ahsan Pervaiz and Henry Hoffmann’s work is supported by NSF (grants CCF-2028427, CNS-1956180, CCF-1837120, CNS-1764039), ARO (grant W911NF1920321), and a DOE Early Career Award (grant DESC0014195 0003). Michael Carbin’s work is supported in part by the National Science Foundation (NSF CCF-1918839). Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

## REFERENCES

- [1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81. <https://doi.org/10.1145/3212695>
- [2] 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. <https://doi.org/10.1145/2628071.2628092>
- [3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 49–65. <https://doi.org/10.1145/2997641>



- [4] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794. <https://doi.org/10.1145/2939672.2939785>
- [5] David Culler, Jaswinder Pal Singh, and Anoop Gupta. 1999. *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing.
- [6] Tom Dietterich. 1995. Overfitting and undercomputing in machine learning. *ACM computing surveys (CSUR)* 27, 3 (1995), 326–327. <https://doi.org/10.1145/212094.212114>
- [7] Yi Ding, Risi Kondor, and Jonathan Eskreis-Winkler. 2017. Multiresolution Kernel Approximation for Gaussian Process Regression. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) (*NIPS'17*). Curran Associates Inc., Red Hook, NY, USA, 3743–3751. <https://doi.org/10.5555/3294996.3295131>
- [8] Yi Ding, Nikita Mishra, and Henry Hoffmann. 2019. Generative and multi-phase learning for computer systems optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*. 39–52. <https://doi.org/10.1145/3307650.3326633>
- [9] Finale Doshi-Velez and Been Kim. 2017. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608* (2017).
- [10] Julian J Faraway. 2014. *Linear models with R*. CRC press. <https://doi.org/10.4324/9780203507278>
- [11] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. 2017. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 1487–1495. <https://doi.org/10.1145/3097983.3098043>
- [12] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*. Vol. 1. MIT press Cambridge. <https://doi.org/10.5555/3086952>
- [13] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 301–311. <https://doi.org/10.1109/ASE.2013.6693089>
- [14] Shengsheng Huang, Jie Huang, Yan Liu, Lan Yi, and Jinquan Dai. 2010. HibenCh: A representative and comprehensive hadoop benchmark suite. In *Proc. ICDE Workshops*. 41–51.
- [15] Connor Imes, Steven A. Hofmeyr, and Henry Hoffmann. 2018. Energy-efficient Application Resource Scheduling using Machine Learning Classifiers. In *Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018*. ACM, 45:1–45:11. <https://doi.org/10.1145/3225058.3225088>
- [16] Engin İpek, Sally A McKee, Rich Caruana, Bronis R de Supinski, and Martin Schulz. 2006. Efficiently exploring architectural design spaces via predictive modeling. *ACM SIGOPS Operating Systems Review* 40, 5 (2006), 195–206. <https://doi.org/10.1145/1168917.1168882>
- [17] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 497–508. <https://doi.org/10.1109/ASE.2017.8115661>
- [18] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 71–82. <https://doi.org/10.1145/3236024.3236074>
- [19] Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. 2017. Transfer learning for improving model predictions in highly configurable software. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 31–41. <https://doi.org/10.1109/SEAMS.2017.11>
- [20] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. 2020. The interplay of sampling and machine learning for software performance prediction. *IEEE Software* 37, 4 (2020), 58–66. <https://doi.org/10.1109/MS.2020.2987024>
- [21] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association. <https://doi.org/10.1145/3355738.3355750>
- [22] Rahul Krishna, Vivek Nair, Pooyan Jamshidi, and Tim Menzies. 2020. Whence to Learn? Transferring Knowledge in Configurable Systems using BEETLE. *IEEE Transactions on Software Engineering* (2020). <https://doi.org/10.1109/TSE.2020.2983927>
- [23] Rahul Krishna, Chong Tang, Kevin Sullivan, and Baishakhi Ray. 2020. ConEx: Efficient Exploration of Big-Data System Configurations for Better Performance. *IEEE Transactions on Software Engineering* (2020). <https://doi.org/10.1109/TSE.2020.3007560>
- [24] Benjamin C Lee and David M Brooks. 2006. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ACM SIGOPS operating systems review* 40, 5 (2006), 185–194. <https://doi.org/10.1145/1168917.1168881>
- [25] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. 2020. Statically inferring performance properties of software configurations. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 10:1–10:16. <https://doi.org/10.1145/3342195.3387520>
- [26] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research* 18, 1 (2017), 6765–6816. <https://doi.org/10.5555/3122009.3242042>
- [27] Weiyang Liu, Bo Dai, Ahmad Humayun, Charlene Tay, Chen Yu, Linda B Smith, James M Reh, and Le Song. 2017. Iterative machine teaching. In *International Conference on Machine Learning*. PMLR, 2149–2158. <https://doi.org/10.5555/3305890.3305903>
- [28] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) (*ICSE '16*). Association for Computing Machinery, New York, NY, USA, 643–654. <https://doi.org/10.1145/2884781.2884793>
- [29] Atefeh Mehrabi, Aninda Manocha, Benjamin C Lee, and Daniel J Sorin. 2020. Bayesian Optimization for Efficient Accelerator Synthesis. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 1 (2020), 1–25.
- [30] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. 2018. CALOREE: Learning Control for Predictable Latency and Low Energy. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) (*ASPLOS'18*). New York, NY, USA, 184–198. <https://doi.org/10.1145/3173162.3173184>
- [31] Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. 2015. A Probabilistic Graphical Model-Based Approach for Minimizing Energy Under Performance Constraints. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) (*ASPLOS'15*). New York, NY, USA, 267–281. <https://doi.org/10.1145/2775054.2694373>
- [32] Christoph Molnar. 2020. *Interpretable machine learning*. Lulu. com.
- [33] L. Nardi, A. Souza, D. Koeplinger, and K. Olukotun. 2019. HyperMapper: a Practical Design Space Exploration Framework. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE Computer Society, Los Alamitos, CA, USA, 425–426. <https://doi.org/10.1109/MASCOTS.2019.00053>
- [34] Simo Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2009), 1345–1359. <https://doi.org/10.1109/TKDE.2009.191>
- [35] Carl Edward Rasmussen. 2003. Gaussian processes in machine learning. In *Summer school on machine learning*. Springer, 63–71.
- [36] Cynthia Rudin. 2019. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence* 1, 5 (2019), 206–215. <https://doi.org/10.1038/s42256-019-0048-x>
- [37] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 342–352. <https://doi.org/10.1109/ASE.2015.45>
- [38] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. 2015. Taking the human out of the loop: A review of Bayesian optimization. *Proc. IEEE* 104, 1 (2015), 148–175. <https://doi.org/10.1109/JPROC.2015.2494218>
- [39] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 284–294. <https://doi.org/10.1145/2786805.2786845>
- [40] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting performance via automated feature-interaction detection. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 167–177. <https://doi.org/10.1109/ICSE.2012.6227196>
- [41] Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. 2017. Transferring performance prediction models across different hardware platforms. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 39–50. <https://doi.org/10.1145/3030207.3030216>
- [42] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1009–1024. <https://doi.org/10.1145/3035918.3064029>
- [43] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. 2012. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 213–224. <https://doi.org/10.1145/2366231.2337184>

- [44] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems. *Proceedings of the 43rd International Conference on Software Engineering (ICSE 21)* (2021). <https://doi.org/10.1109/ICSE43902.2021.00100>
- [45] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. 2010. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*. 193–204. <https://doi.org/10.1145/1807128.1807161>
- [46] Chengcheng Wan, Muhammad Husni Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, and Shan Lu. 2020. ALERT: Accurate Learning for Energy and Timeliness. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 353–369. <https://www.usenix.org/conference/atc20/presentation/wan>
- [47] Zhiyuan Wan, Xin Xia, David Lo, and Gail C Murphy. 2019. How does machine learning change software development practices? *IEEE Transactions on Software Engineering* (2019). <https://doi.org/10.1109/TSE.2019.2937083>
- [48] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. 2018. Understanding and auto-adjusting performance-sensitive configurations. *ACM SIGPLAN Notices* 53, 2 (2018), 154–168. <https://doi.org/10.1145/3173162.3173206>
- [49] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 307–319. <https://doi.org/10.1145/2786805.2786852>
- [50] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 619–634. <https://doi.org/10.5555/3026877.3026925>
- [51] Tianyin Xu, Vineet Pandey, and Scott Klemmer. 2016. An HCI view of configuration problems. *arXiv preprint arXiv:1601.01747* (2016).
- [52] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How do fixes become bugs?. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 26–36. <https://doi.org/10.1145/2025113.2025121>
- [53] Zhibin Yu, Zhendong Bei, and Xuehai Qian. 2018. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 564–577. <https://doi.org/10.1145/3173162.3173187>
- [54] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95. <https://doi.org/10.5555/1863103.1863113>
- [55] Huazhe Zhang and Henry Hoffmann. 2019. PoDD: power-capping dependent distributed applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, Michela Taufer, Pavan Balaji, and Antonio J. Peña (Eds.). ACM, 28:1–28:23. <https://doi.org/10.1145/3295500.3356174>