Learning to Drive Software-Defined Solid-State Drives

Daixuan Li
University of Illinois
Urbana-Champaign
daixuan2@illinois.edu

Jinghan Sun
University of Illinois
Urbana-Champaign
js39@illinois.edu

Jian Huang
University of Illinois
Urbana-Champaign
jianh@illinois.edu

ABSTRACT

Thanks to the mature manufacturing techniques, flash-based solid-state drives (SSDs) are highly customizable for applications today, which brings opportunities to further improve their storage performance and resource utilization. However, the SSD efficiency is usually determined by many hardware parameters, making it hard for developers to manually tune them and determine the optimized SSD hardware configurations.

In this paper, we present an automated learning-based SSD hardware configuration framework, named AutoBlox, that utilizes both supervised and unsupervised machine learning (ML) techniques to drive the tuning of hardware configurations for SSDs. AutoBlox automatically extracts the unique access patterns of a new workload using its block I/O traces, maps the workload to previous workloads for utilizing the learned experiences, and recommends an optimized SSD configuration based on the validated storage performance. AutoBlox accelerates the development of new SSD devices by automating the hardware parameter configurations and reducing the manual efforts. We develop AutoBlox with simple yet effective learning algorithms that can run efficiently on multi-core CPUs. Given a target storage workload, our evaluation shows that AutoBlox can deliver an optimized SSD configuration that can improve the performance of the target workload by 1.30× on average, compared to commodity SSDs, while satisfying specified constraints such as SSD capacity, device interfaces, and power budget. And this configuration will maximize the performance improvement for both target workloads and non-target workloads.

CCS CONCEPTS

• **Hardware** → **External storage**; • **Computer systems organization** → **Architectures**; • **Computing methodologies** → *Machine learning algorithms.*

KEYWORDS

Learning-Based Storage, Solid State Drive, Software-Defined Hardware, Machine Learning for Systems

ACM Reference Format:
Daixuan Li, Jinghan Sun, and Jian Huang. 2023. Learning to Drive Software-Defined Solid-State Drives. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3613424.3614281

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3614281

1 INTRODUCTION

Flash-based solid-state drive (SSDs) have become the backbone of modern storage infrastructures in various computing platforms, as they offer orders-of-magnitude better performance than hard disk drives (HDDs), while their cost is approaching to that of HDDs [1, 21, 25, 38, 51]. Thanks to the development of manufacturing and shrinking process technology, the industry has been able to rapidly produce SSD devices with different hardware configurations.

Although SSD devices are becoming highly customizable to meet the ever-increasing demands on storage performance and capacity for new applications (i.e., software-defined SSDs) [25, 38], identifying optimized device configurations is on the critical path of SSD development. This is because the SSD hardware configurations are usually determined by the requirements from applications and customers [7, 31], and these configurations involve many components in the storage controller, such as flash chip specifications, chip layout, block/page sizes, device buffer sizes, and others.

In order to deliver optimized performance, storage vendors usually use typical application workloads as their benchmarks to help determine the device configurations. However, an SSD device usually has hundreds of hardware parameters, and these parameters normally have dependencies (i.e., the update of one parameter may affect others), making it hard for hardware engineers to tune the device configurations and identify the optimized ones quickly. This significantly hurts the productivity of new SSD device development.

There is an increasing demand for customized storage devices for applications. This is for two major reasons. First, computing platforms always wish to deploy the best-fit storage devices for their workloads. This is especially true for cloud platforms that require highly customized SSDs to support their cloud services [12, 14, 25, 38], such as Database-as-a-Service [56] and web services [3]. As applications evolve quickly, we need to revolutionize the configuration tuning procedure to shorten the lifecycle of producing new SSDs. Second, our study shows that storage workloads can be categorized with learning algorithms (see Figure 2), which shows that it is feasible to customize storage devices for a specific workload type, especially considering SSD manufacturing becomes mature today. However, there is still a long-standing gap between application demands and SSD productions.

In this paper, we develop an automated learning-based SSD hardware configuration framework named AutoBlox, which exploits both supervised and unsupervised machine learning (ML) techniques to drive the hardware configurations for new SSDs, with specified hardware constraints. Specifically, given a type of target storage workload, AutoBlox will recommend an SSD configuration that delivers optimized storage performance. Similarly, given

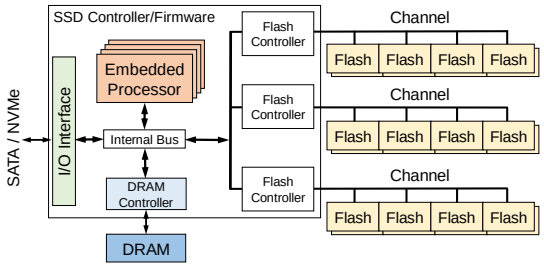

Figure 1: Internal architecture of flash-based SSDs.

a performance target for a type of target workload, AutoBlox can identify the optimized configurations with what-if analysis (see §4). It leverages linear regression to expose the device configurations that have the strongest correlation to storage performance. To present reasonable device configurations, we formulate different types of hardware parameters and performance metrics in the SSD, transfer them into vectors in a customized Bayesian optimization model, and utilize learning techniques to explore the optimization space and identify optimized options, with specified constraints such as SSD capacity, device interfaces (NVMe or SATA), flash memory types, and power budget of the storage device.

To reduce the time of learning an optimized SSD configuration while ensuring the learning accuracy, we develop pruning algorithms to identify the most important SSD parameters for different workloads. Our study leads to interesting insights. We turn them into learning rules that can aid AutoBlox to adjust the tuning procedure. For instance, (1) the sensitivity of device configurations to different optimization targets (e.g., latency and throughput) depends on the workload characteristics. AutoBlox quantifies the trade-offs with a unified optimization target to guide the tuning procedure. (2) Not all SSD parameters are equal. Some SSD parameters have non-linear correlations with storage performance, and these parameters vary for different workload types. (3) Not all parameters are sensitive to storage performance, and different sensitivities of different parameters can be utilized to improve the turning efficiency (see the detailed discussion in §3.3).

AutoBlox also maintains a configuration database called AutoDB to store the learned workloads and SSD configurations. For a new workload, AutoBlox will extract its features and compare them with the records in the AutoDB using clustering. If it identifies a similar workload in its AutoDB, it will recommend the corresponding SSD configuration directly, thus, we can utilize the previously learned experience. Otherwise, AutoBlox will learn a new SSD configuration for the workload, and add it into AutoDB for future references.

To evaluate AutoBlox, we implemented our proposed techniques using PyTorch [42], *scikit-learn* [6], and a production-level SSD simulator MQSim [55]. We perform experiments with various storage workloads. Our experimental results show that, out of a search space of billions of possible configurations, AutoBlox can learn a new optimized configuration in 670.89 seconds, and finalize it in 89 iterations on average. We also show that AutoBlox delivers an SSD configuration that can achieve 1.25–1.93× performance improvement for a target workload, and 1.09× improvement on average for non-target workloads with up to 16% energy saving, compared to the configurations specified in commodity SSDs. Overall, we make the following contributions:

- We present the first study of SSD hardware parameters and popular storage workloads with learning in mind, and demonstrate the feasibility of applying the learning-based approach for identifying optimized SSD specifications.
- We formulate the tuning problem of SSD device configurations using both supervised and unsupervised learning techniques, and develop an automated framework that can learn optimized SSD configurations for different workloads with defined constraints.
- We summarize a set of learning rules that can facilitate the hardware configurations and development of new SSDs, based on our study with AutoBlox.
- We examine the efficiency of AutoBlox and show its benefits for a variety of target data-intensive workloads, in comparison with different commodity SSD settings.

2 BACKGROUND AND MOTIVATION

2.1 SSD Architecture

We present the internal architecture of a typical SSD in Figure 1. An SSD consists of five major components: a set of flash memory packages, an SSD controller having embedded processors like ARM, off-chip DRAM (SSD DRAM), flash controllers, and the I/O interface that includes SATA and NVMe protocols [13, 20, 28, 46]. The flash packages are organized in a hierarchical manner. Each SSD has multiple channels where each channel can process read/write commands independently. Each channel is shared by multiple flash packages. Each package has multiple flash chips. Within each chip, there are multiple planes. Each plane includes multiple flash blocks, and each block has multiple flash pages. The page size varies in different SSDs. When a free flash page is written once, that page is no longer available for future writes until that page is erased. However, erase operation is expensive and performed at block granularity. As each flash block has limited endurance, it is important for blocks to age uniformly (i.e., wear leveling). Modern SSD controllers employ out-of-place write, GC, and wear leveling to overcome these shortcomings and maintain indirections for the address translation in their flash translation layer (FTL) [26, 58].

2.2 SSD Manufacturing Procedure

According to the interviews with SSD product managers [7] and our discussions with SSD vendors, finalizing SSD configurations is on the critical path in the SSD design. These configurations are usually determined by the requirements from applications and customers.

To finalize the SSD specifications, a simple approach is to test and profile application workloads with different hardware configurations. However, this is not scalable as we target different applications. With the confirmation from SSD vendors, there are more than a hundred tunable parameters in an SSD. Given an application, it is challenging for developers to explore all the combinations of device parameters. Likewise, given a new SSD specification, it requires significant manual effort to quantify the effectiveness of the selected parameters. In this work, we use the learning-based approach to automate the SSD hardware configurations. It helps both

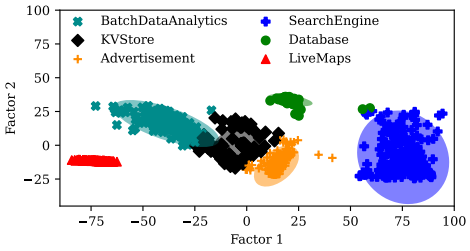

Figure 2: A clustering of popular storage workloads.

storage vendors and platform operators who want to efficiently identify optimized SSD configurations for their target workloads, accelerates their decision-making process for device configurations, and further benefits the manufacturing process of customized SSDs.

2.3 Software-Defined Solid-State Drive

With the increasing demands on storage performance from applications, storage systems are embracing software-defined hardware techniques [12, 14]. This allows upper-level applications to achieve maximum performance benefits and resource efficiency with customized storage devices. For instance, the recent development of software-defined flash [25, 38] enables platform operators to customize the number of flash channels and chips in an SSD, with the cooperation with SSD vendors.

The emerging demands of typical applications also motivate the need to customize hardware design for specific applications [4, 16, 48]. Applications such as Database-as-a-Service [56], web services [3], web search [25], and batch data analytics [15], can be highly classified. For instance, we use our learning-based workload characterization approach (see §3.1) to study the storage traces from a set of popular application workloads (see Table 2). Our experiments show that each workload type has its unique characteristics, and I/O traces from the same workload type have similarities in their data access pattern, as we used Principal Component Analysis [66] to map them into two dimensions in Figure 2. This shows that it is feasible to customize storage devices for a specific category of applications. Also, it is known that different workloads have different performance behaviors on the same SSDs, and the same workload will perform differently on different SSDs [29, 30, 39, 44, 65], which further motivates the need for customized storage devices. However, we lack a framework that can efficiently transfer application demands and characteristics into the hardware configurations of SSDs. In this work, we will use both supervised learning and unsupervised learning to develop AutoBlox framework.

Note that the software-defined SSD in this paper refers to the concept of allowing developers to customize SSD hardware based on the application needs. The OpenSSD [52] is different, as its purpose is to allow upper-level software to manage underlying flash chips. OpenSSD does not fit for our purpose, since its hardware configuration has been fixed and cannot be adjusted at runtime (e.g., its flash chip layout).

3 DESIGN AND IMPLEMENTATION

Our goal is to enable the automated tuning of SSD configurations for a specific workload with learning techniques.

- It should generate an optimized SSD configuration for a target workload, while this hardware configuration has minimal negative impact on other non-target workloads.
- It should identify an optimized SSD configuration quickly, without introducing much computation overhead.
- It should scale to support diverse target workloads as well as different device constraints.

We show the system architecture of AutoBlox in Figure 3, and discuss each component in detail as follows.

3.1 Learning-Based Workload Clustering

To categorize storage workloads, traditional methods usually use the read/write ratio and I/O patterns (e.g., sequential and random read/write). However, they cannot capture the whole picture of workload characteristics. Instead, we develop a learning-based clustering approach based on block I/O traces. This approach does not have system dependencies and does not require software semantics.

To facilitate the learning-based workload clustering, we first remove the information irrelevant to the workload characteristics in a block I/O trace. As the absolute value of a block address depends on the block allocation algorithm, we transfer the absolute block addresses of each workload to relative addresses (i.e., offsets) in a uniform block address space. We keep the I/O size and type (i.e., read/write) unmodified.

We then partition each I/O trace into small windows, each window represents the access pattern of a time period within the workload. According to our study of diverse workloads, we use 3,000 trace entries in each window by default. This is because using fewer entries will result in the loss of important data access patterns, while a larger number of entries will slow down the workload clustering procedure consisting of normalization, PCA and clustering.

The trace information used to conduct the workload characterization include the I/O timestamp, I/O size, block address, and operation types. We focus on their relative values by normalizing them with the values of the starting entry of each window. Since each window of I/O trace has multiple entries and each entry have multiple fields, we use Principal Component Analysis (PCA) [66] to transfer each window into 5 dimensions. This will simplify the learning of I/O workload patterns by transferring the complex characteristics of I/O traces into fewer dimensions. AutoBlox can learn traditional workload characteristics by default. These characteristics are quantified and differentiated by the PCA and k-means clustering on the block I/O trace. For example, for sequential/random access patterns, the vectors of adjacent logical page addresses will have monotonic increasing values and random values respectively. These different patterns are captured by PCA. Similarly, for I/O intensity, a more intensive workload will have larger I/O sizes and smaller time gaps between I/O requests within the same trace window. Our results show that selecting 5 dimensions capture 70.4% of the explainable variance of the dataset, and the maximum explainable variance of other dimensions is less than 1%.

After that, we use k-means to cluster these data points. We calculate the distance between the center of the examined data points and the center of an existing cluster. If the distance is below a configurable threshold (20 in AutoBlox), we claim that the new workload belongs to this cluster. This threshold corresponds to the

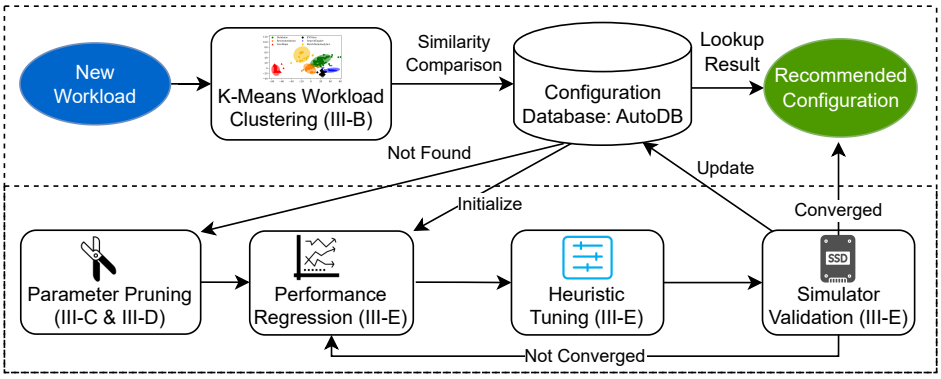

Figure 3: System overview of AutoBlox. It first learns new workload features with workload clustering (§3.1). If the new workload is similar to workloads in existing clusters in the configuration database AutoDB, AutoBlox will recommend an optimized configuration stored in the database. If not, AutoBlox will first conduct parameter pruning (§3.2 and §3.3) to identify performance-critical parameters of SSDs. Then perform three automated tunings: performance regression with Gaussian process, heuristic tuning, and efficiency validation (§3.4).

minimum distance between existing clusters, ensuring that a new cluster is only formed when the distance exceeds this threshold. If AutoBlox cannot identify a similar cluster, it will retrain the kmeans model with one more cluster. As shown in Figure 2, our learning-based workload clustering can successfully identify a cluster with the same or similar storage workloads.

To verify the effectiveness of the learning-based workload clustering, we collect multiple traces from different application categories. Each trace lasts 15-240 hours, covering multiple execution phases of an application. We divide each workload into training and validation datasets. We empirically set the number of cluster of k-means clustering algorithm to the number of workload categories, since a larger number of cluster could not differentiate the traces from different workload categories, and a smaller number of clusters may further separate traces from the same workload category. We observe that 95% of the validation data points fall into the same workload cluster on average. This shows that our approach is sufficient to identify an appropriate cluster for new workloads, and also can tolerate workload variations at different execution phases. It also offers the insight that it is feasible to develop an SSD that can deliver optimized performance for a category of workloads.

Note that it is rare that a new application belongs to one category but cannot be clustered into that category. For the rare case in which an application belongs to a category but it does not fall into the workload cluster, AutoBlox will treat it as a workload outlier. AutoBlox will still identify optimized SSD configurations for the majority of applications within that category, as it targets a category of applications. As AutoBlox receives a certain number (e.g., 20 by default) of such applications, AutoBlox will create a new category and learn optimized configurations for it.

3.2 Transfer SSD Tuning into ML Models

We now discuss how we can transfer the tuning problem of SSD configurations for a type of workload into an ML problem, such that we can automate the tuning procedure. To achieve this, we follow two steps: (1) develop an appropriate ML model for auto-tuning the SSD hardware configurations, and (2) formulate the SSD specifications and performance metrics into parameters of the ML model we will develop.

We develop a customized Bayesian optimization (BO) model for tuning SSD parameters. Based on the performance of known configurations, our model can predict the performance of similar unseen configurations, and deliver optimized configuration under certain constraints (e.g., storage capacity and power budget). As BO model can deliver similar performance compared to deep neural networks, but with low performance overhead and high explainability [11, 57], it is a natural fit for solving SSD tuning problem [67], especially for the cases that have search space constraints (e.g., capacity constraints). Similar to the predicted rewards used in DNN-based algorithms, BO quantifies the exploration trade-offs with predicated mean and variance values in the turning procedure, therefore, it can find near-optimal configurations within a certain number of iterations. It sometimes performs even faster than DNNs like deep Q-networks (DQN), as the DNNs usually need multiple iterations to train/retrain their reward functions [11].

In our customized BO learning model, we transfer SSD specifications and performance metrics into parameters of the BO model. This procedure is not easy. For the SSD specifications, we have to ensure the parameter formulation does not lose valid SSD hardware configurations, and the ML parameters can represent the characteristics of different device parameters as well as their correlations. For the performance metrics, we need to unify different performance trade-offs (e.g., latency and throughput) into one unified optimization target, therefore, the BO model can find the optimized solution with different trade-offs under defined constraints.

To model the SSD configurations and performance metrics, we formulate them into three major parts in the ML model: (1) the unified efficiency metrics used as the optimization targets for SSDs; (2) SSD configurations that can be vectorized as parameters; and (3) the constraints (e.g., the SSD capacity and power budget) that bound the optimization space. We discuss them as follows.

Efficiency metrics used in the ML model. As for storage efficiency, AutoBlox focuses on the storage latency and throughput under the SSD capacity and power budget constraints. To quantify whether an SSD configuration delivers the optimized performance

or not, we use reference performance as the baseline (i.e., the latency and throughput obtained from a commercial SSD's configurations), and the relative performance improvements as the evaluation metrics. Given a workload $\boldsymbol{W}$, we have a target configuration ***target***, and a reference configuration ***refer***, we have the following performance optimization goal:

$$\begin{aligned} &\boldsymbol{Performance}_{\boldsymbol{W}}\,(\boldsymbol{target}) = \\ &(1-\alpha) \times log\left(\frac{\boldsymbol{Latency}_{\boldsymbol{refer}}}{\boldsymbol{Latency}_{\boldsymbol{target}}}\right) + \alpha \times log\left(\frac{\boldsymbol{Throughput}_{\boldsymbol{target}}}{\boldsymbol{Throughput}_{\boldsymbol{refer}}}\right) \end{aligned} \quad (1)$$

where α is a tunable hyperparameter for balancing the latency and throughput improvement at proper scale. We set $\alpha = 0.5$ by default, based on our study of different coefficients (see §4.6).

Transfer SSD hardware specifications into ML parameters. To represent SSD hardware specifications in the ML models, we transfer them into four types of ML parameters and use different ways to set their values. They include *continuous*, *discrete*, *boolean*, and *categorical* parameters.

- Continuous parameter: They include SSD data cache size, cached mapping table size, over-provisioning ratio for GC, and others. To set the value of these type of parameters, we identify a range of possible values it could take in advance, and divide the range uniformly into N small pieces. Therefore, AutoBlox can take N endpoints as the possible values. We set the range to cover all common values in commodity SSDs to ensure the learned configurations are practical.
- Discrete parameter: its typical examples include the number of flash channels and PCIe bandwidth. We select the possible values and store them in a list. Their possible values also cover all the common values. Each discrete parameter follows different rules. For instance, the number of flash channels follows the possible values from commodity SSDs and the PCIe bandwidth follows the PCIe specifications defined in industry [40].
- Boolean parameter: We use the boolean parameters (0/1) to indicate whether a feature (e.g., statistic wear leveling, and greedy GC) will be enabled in the SSD or not.
- Categorical parameter: we convert it to the dummy variable. For example, there are 16 possible values for the plane allocation scheme, we create a list with a length of 16. When AutoBlox selects one scheme, it will set the value of the corresponding index to 1, and others to 0.

Configuration constraints. AutoBlox allows users to specify the configuration constraints. Typical examples include the SSD storage capacity, the interface (e.g., NVMe or SATA) supported by the SSD for interacting with the host machine, and the power budget for the SSD device. When AutoBlox sets different values for the device parameters while tuning of hardware configurations, it will simply abandon those configurations that violate the specified constraints at the efficiency validation stage. In the follow-up searching process, these configurations will be skipped. AutoBlox mainly works on the tunable parameters in SSD specifications. AutoBlox does not cover the lower-level circuit-relevant specifications that are strictly limited by the hardware.

3.3 Learning-based Parameter Pruning

After we transfer the SSD specifications and performance metrics into ML parameters, we can start to train the model. However, modern SSDs usually have hundreds of hardware parameters. Although ML models today can handle a large set of parameters, it is still desirable to develop efficient and lightweight models for reducing the learning time, and saving computation cycles [57]. For example, we develop a model covering 48 SSD device specifications or parameters; it takes 30.7 hours for the model to converge on a modern multi-core server (see the experimental setup in §4.1). We also find that not all SSD device parameters are strongly correlated to the storage performance, and these insensitive parameters may even hurt the learning efficiency. To this end, we develop a parameter pruning approach to identify the impactful device parameters.

We have to overcome two major challenges within the parameter pruning. First, we need an accurate method to measure the importance of a parameter. This is challenging because SSD parameters usually have dependencies. If we tune one parameter while keeping the values of other parameters fixed, it may violate the configuration constraints. For example, increasing the number of flash channels could violate the constraint of the SSD capacity. On the other hand, as we tune one parameter while updating the values of other device parameters accordingly for meeting the configuration constraints, we cannot accurately determine which parameters affect the storage performance significantly. Second, removing some of the SSD device parameters may hurt the overall accuracy of the learning model. And it is challenging to quantify how each parameter could affect the learning accuracy. To address these challenges, we conduct the parameter pruning procedure within two stages.

Coarse-grained parameter pruning. We first use a coarse-grained pruning method that adjusts the values of continuous and discrete numerical device parameters with a large stride length. In this stage, we only prune parameters that have almost no impact on the performance even if they break the configuration constraints. As shown in Figure 4, we increase the values of the 35 numerical parameters of SSDs from their baseline setting to 16×, and measure the storage performance with different workloads. We identify insensitive device parameters that do not affect the storage performance significantly (those flat lines in Figure 4).

We also find that these insensitive device parameters vary for different workload types, therefore, AutoBlox will conduct the coarse-grained parameter pruning for each type of workload and identify the corresponding insensitive parameters. In general, we identify 12 insensitive parameters, such as *Page_Metadata_Capacity*, *Static_Wearleveling_Threshold*, and *Suspend_Program_Time* (see Figure 4). And also, we do not observe any pair of insensitive SSD parameters have impact on the storage performance when tuning them simultaneously. This is because the hardware modules of SSDs are usually independent. As for these insensitive SSD parameters, they will not affect SSD performance whatever we tune them together or independently.

Fine-grained parameter pruning. After eliminating redundant parameters in coarse-grained pruning, we continue the pruning with a fine-grained approach. We employ the linear regression technique Ridge [41] to identify the linear correlations between the SSD

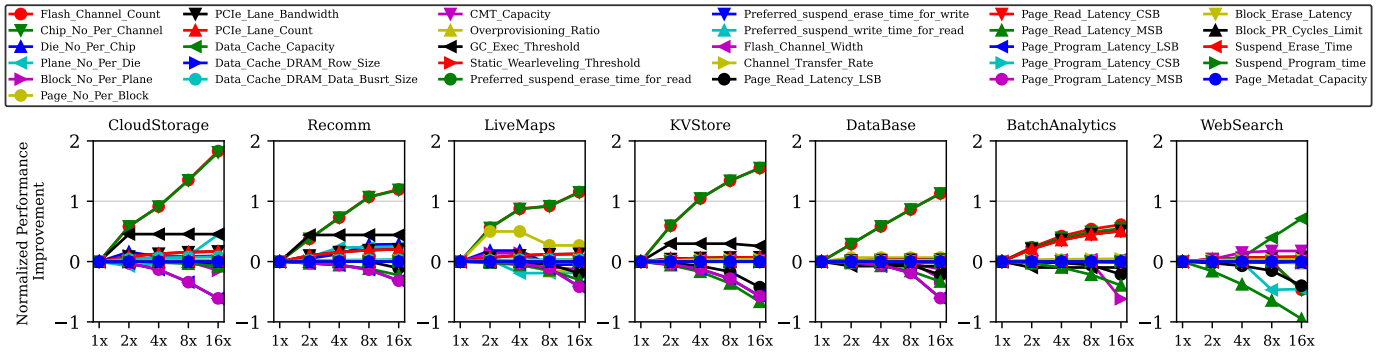

Figure 4: A study of coarse-grained parameter pruning for different storage workloads.

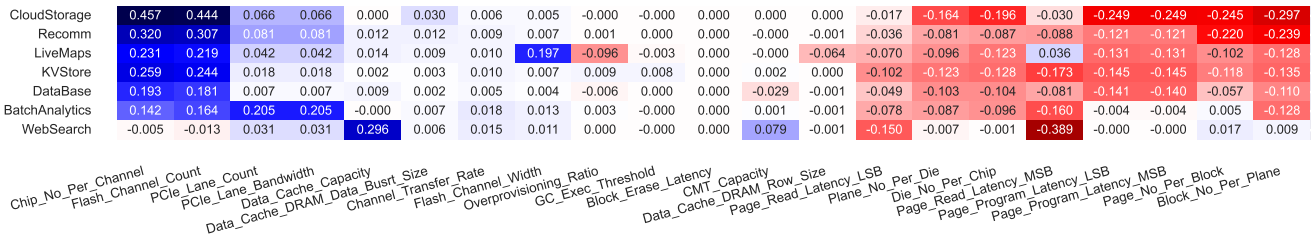

Figure 5: A study of fine-grained parameter pruning with linear regression. As we increase the values of device parameters with linear regression, some device parameters have a positive impact on the storage performance (blue colors), while some have a negative impact (red colors). We consider all these important ones in automated tuning.

parameters and performance. We set a regression space by maintaining the constraints of SSD capacity and power budget. We vary the values of device parameters and measure the regression coefficient for each parameter. A higher coefficient score of a parameter means it has a stronger correlation with the SSD performance. We abandon the parameter whose score is below a threshold (±0.001 by default) as shown in Figure 5. Thus, we can focus on parameter tuning for the important ones. These parameters have different correlations with the SSD performance. Specifically, the parameters that have linear correlation with performance include the number of channels and chips, and queue depth. The parameters that have non-linear correlations with SSD performance include the data cache size, and the number of dies/planes. Instead of limiting the search space for optimized configurations by tuning these parameters in a simple order, we use the regression coefficient of each parameter to guide the tuning order to improve the model efficiency.

In addition, we also order the parameters based on the absolute value of their coefficient scores, and use the orderings as learning rules to guide the turning order of device parameters (see §3.4). Our experimental results show that these learning rules can significantly improve the learning efficiency of AutoBlox (see the discussion in §4 and Figure 9).

Key observations. The learning-based parameter pruning not only helps us to eliminate the insensitive parameters but also offers interesting insights that would benefit SSD development. We observe that: (1) as we consider multiple optimization targets (e.g., latency, throughput, target workload, and non-target workloads) and constraints (e.g., SSD capacity and power) in the tuning of SSD configurations, the sensitivity of device configurations to different optimization targets depends on the workload characteristics. For example, latency-critical workloads like WebSearch are less sensitive to the flash chip layout, while the I/O-intensive workloads like KVStore and LiveMaps are extremely sensitive to the chip layout. This is because flash chip layout will significantly affect the SSD bandwidth. AutoBlox quantifies these trade-offs with a unified optimization target to guide the tuning procedure. (2) Not all SSD parameters are equal. Some SSD parameters have non-linear correlations with storage performance, and these parameters vary for different workload types. For example, the data cache size, and the number of dies and planes in an SSD. This observation reflects the complexity of SSD parameter tuning, motivating us to leverage learning-based techniques in AutoBlox. (3) Not all parameters are sensitive to storage performance, and some of them can be configured the same as commodity SSDs. By ordering the parameters based on their sensitivity to storage performance, AutoBlox can significantly reduce the learning time.

3.4 Automated Tuning of SSD Configurations

We now explain the details of the customized Bayesian optimization model for learning optimized SSD configurations. We present the system workflow of AutoBlox in Figure 6. Given a workload, AutoBlox will first use the configurations stored in the AutoDB as the initial configuration set, and leverage both Gaussian Process Regression (GPR) [45] and discrete Stochastic Gradient Descent (SGD) [35, 63] algorithms to explore optimized configurations. For each optimized configuration proposed by the model, AutoBlox will use a cycle-accurate SSD simulator to validate its performance

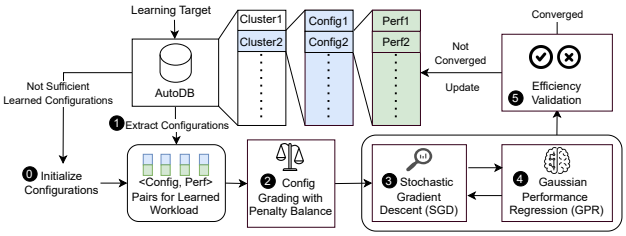

Figure 6: The automated learning workflow of AutoBlox.

and energy consumption until the model converges (i.e., an optimized SSD configuration is identified). We discuss each step of its workflow in detail.

Identify the initial configuration set for a new workload. For a new workload, AutoBlox will use the learning-based workload clustering as discussed in §3.1 to cluster the workload, and look up the learned configurations for the corresponding workload cluster in AutoDB (❶). AutoBlox will use these configurations and their delivered performance to initialize the ML model. However, if there are insufficient configurations in AutoDB (e.g., the AutoDB is empty), AutoBlox will use a configuration from existing commodity SSDs and its measured performance to initialize the model (❶).

Quantify the learned configurations with a unified grade. We will start the tuning procedure with the initial configurations. In order to check the effectiveness of these configurations, AutoBlox develops a grading mechanism (❷) to unify different performance metrics. To achieve the maximal optimizations for both data access latency and throughput, AutoBlox uses the Formula 1 as the learning goal. To ensure the learned configuration for a target workload does not hurt the performance of other workloads, AutoBlox introduces a new factor β named *penalty balance* in its grading. Therefore, we define the performance grade for a workload as follows: given a target workload $\boldsymbol{W}$, a target configuration $\boldsymbol{conf}$, and non-target workloads $\boldsymbol{W'}$,

$$\begin{aligned} \boldsymbol{Grade}_{\boldsymbol{W}}(\boldsymbol{conf}) &= (1-\beta) \times \boldsymbol{Performance}_{\boldsymbol{W}}(\boldsymbol{conf}) \\ &+ \beta \times \frac{\sum_{W' \in non-target} \boldsymbol{Performance}_{\boldsymbol{W'}}(\boldsymbol{conf})}{\boldsymbol{NumClusters} - 1} \end{aligned} \quad (2)$$

where $\boldsymbol{Performance}_{\boldsymbol{W}}(\boldsymbol{conf})$ follows the Formula 1, $\beta = 0.1$ by default based on our study (see Figure 12), and $\boldsymbol{NumClusters}$ denotes the number of different types of workloads available in the configuration database AutoDB. The second part in the equation represents the average performance of all the non-target workload clusters. We use the geometric mean to calculate the average performance within each cluster, and then average across all clusters. The denominator covers the number of clusters, which does not change during optimization for each workload type.

Search optimized configurations with SGD techniques. With the initial SSD configurations and their grades, AutoBlox will use SGD to search an optimized configuration (❸). AutoBlox first identifies the top three best configurations (i.e., the configurations whose grades rank at the top) from the learned configurations and randomly selects one as the search root. We randomly select the search root from the top three configurations to prevent SGD from converging into a suboptimal configuration. This is because checking fewer top configurations can accelerate the converging process and identify optimized configurations, while checking a larger number of configurations will significantly increase the convergence time, but still cannot identify a better configuration. In the gradient descent process, AutoBlox then expands the search space from the root by checking all the adjacent configurations under constraints (e.g., SSD capacity) in each search iteration. AutoBlox will tune the parameters following the ranking of their absolute coefficient scores for improved learning efficiency (see §3.3).

To ensure that we meet the specified constraints (e.g., SSD capacity and SSD DRAM capacity), AutoBlox will adjust the values of other parameters with linear regression. These parameters include seven SSD layout parameters (e.g., number of channels, number of chips per channel, and number of dies per chip) and two DRAM-related parameters (cached mapping table size and data cache size). AutoBlox will use GPR model to identify the configuration with the best-predicted performance grade (❹). If its performance grade is better than the search root, AutoBlox will set this configuration as the new root and continue the next search iteration.

The main challenge with the SGD procedure (❸) is to balance the learning accuracy and exploitation overhead. Since there is no guarantee that the initial configuration set will cover the entire search space, AutoBlox has to gradually expand its search space to ensure it can identify the optimized ones. However, this may cause search space explosion, and the prediction will be less accurate when the target configuration is far from the exploited search space. To address this issue, we introduce a heuristic exploration factor – the minimum Manhattan distance [64] between the configuration being exploited and the learned configurations. Once the minimum Manhattan distance reaches a threshold (5 by default), we will stop the SGD search and validate the current learned configuration. We also set a threshold for the number of search iterations (10 by default in AutoBlox) in the configuration exploration.

Predict the grades of explored configurations. As briefly discussed above, AutoBlox uses GPR [45] to predict the grades for new device configurations (❹). This is for four major reasons. First, GPR can provide nearly the same performance as deep neural networks, especially in the modeling of searching optimized configurations and making recommendations. Second, it offers excellent trade-offs between the exploration of new knowledge and learned knowledge [32, 53]. Third, GPR provides confidence intervals with low computation overhead by default [2], making AutoBlox computation affordable with multi-core systems. Finally, GPR can help AutoBlox estimate the efficiency of a configuration in a shorter time; therefore, we can avoid frequent expensive performance validations with SSD simulators (see the overheads in Table 6).

We build a new GPR model by specifying its mean function and covariance function. The mean function is trainable since the mean of the performance metrics is unknown before the learning in AutoBlox. We use the covariance functions to represent the correlation between two adjacent points in the model and adopt both radial basis function (RBF) kernel [61] and rational quadratic kernel [62] as the regression covariance. The parameter in the covariance functions (e.g., length scale) are also trainable, and they can learn how much performance similarity is there for two adjacent configurations. To tolerate the noise in simulations, we also add

a white kernel [49] for random noise simulation. GPR model is initialized each iteration with validated configurations in AutoDB. Then, given the input of a set of device configurations generated by SGD (❸), GPR will help AutoBlox select the best one based on its estimated grade.

Validate the efficiency of explored configurations. Once we obtain the selected device configuration from GPR, AutoBlox will conduct the efficiency validation using a cycle-accurate SSD simulator (❺). Before the validation, AutoBlox will warm up the SSD simulator by running diverse workload traces randomly. After that, AutoBlox will run the storage workloads in the AutoDB against the SSD simulator configured with the selected device configuration. AutoBlox will obtain the power consumption of the configured SSD. If it exceeds the specified power budget, it means the selected device configuration violates the power constraint, AutoBlox will drop this configuration and start a new learning iteration to learn another optimized configuration. Otherwise, AutoBlox will use the Formula 2 to calculate the performance grade. In the validation, AutoBlox maintains a set of optimized configurations whose grades rank at the top of all the learned configurations. It is worth noting that the grade considers the performance of target workloads and non-target workloads.

After a certain number of search iterations, if the overall grade of this configuration set is not significantly updated (within the bound [-1%, 1%]), the learning procedure will converge. Otherwise, AutoBlox will update the AutoDB with the newly learned configuration and start another search iteration until the learning converges.

The learning procedure of one new configuration will terminate by checking two conditions: (1) no configuration is better than the current root configurations in the search space; or (2) the search exceeds the threshold for the number of iterations (5 by default in AutoBlox). As discussed, AutoBlox will explore the search space from the root configuration by checking all the adjacent configurations until the learning procedure converges. According to our experiments (see §4.3), AutoBlox incurs 89 search iterations on average to identify an optimized configuration.

Optimization for efficiency validation. The efficiency validation could be expensive because running a workload with the SSD simulator will take a large amount of time (see Table 6). And we not only need to run the target workload but also execute the non-target workloads to ensure the selected configuration can deliver the best performance tradeoff between target workloads and non-target workloads. To accelerate the efficiency validation, we develop a pruning mechanism to avoid the execution of non-target workloads, based on the insight that: if the performance grade of the target workload (first half of Formula 2) running with the selected configuration is lower than the worst grade of existing learned configurations, we do not need to run non-target workloads, because the selected configuration usually has a negative impact on the performance of non-target workloads. In this case, we can quickly decide whether we need to run another learning iteration or not, rather than spending much time executing non-target workloads.

3.5 Implementation Details

AutoBlox Implementation. We implement AutoBlox with Python. AutoBlox supports storage traces collected with *blktrace* which is available on most computing systems. It uses PCA and k-means algorithms in the learning-based workload clustering. AutoBlox utilizes the scikit-learn library v0.24.2 [6] to develop the learning model that supports both SGD and GPR algorithms. AutoBlox uses the cycle-accurate SSD simulator MQSim to validate the efficiency of learned configurations. As validated in [55], the reported performance of MQSim matches with commodity SSDs.

AutoBlox extends MQSim by adding the power modeling and profiling for the efficiency validation of the learned configurations. AutoBlox measures the power consumption of three major components of the SSD controller at runtime: flash chips, SSD DRAM, and storage processor. We track the read/write/erase operations and the idle cycles for each flash chip. With the power model of flash chips discussed in [19], we can calculate the total energy consumption of flash chips. For the DRAM power consumption, we integrate the DRAMPower simulator into MQSim and use DRAM parameters specified in [18]. For the storage processor, we employ the ARM power modeling from Gem5 [17] and measure its power consumption as we issue storage operations.

We will open source AutoBlox. AutoBlox is also compatible with other SSD simulators, thus, SSD vendors can replace MQSim with their own simulators. AutoBlox implements AutoDB with the key-value store LevelDB, in which the key is the workload cluster ID, and the value includes the corresponding SSD configurations and their performance grades. The value is organized in JSON format. AutoBlox provides a simple interface *set_cons (capacity, interface, flash_type, power_budget)* for users to specify the constraints of SSD capacity, interface (i.e., NVMe or SATA), the flash type (i.e., SLC/MLC/TLC), and power budget, respectively.

AutoBlox Deployment. AutoBlox is deployed mainly for storage vendors and (datacenter) platform operators who want to efficiently identify optimized device configurations for their target workloads.

AutoBlox Training. For the clustering, we use 70% of storage workloads as the training set, and the rest of the workload trace as the testing set to validate the model. For the model of learning configurations, we divide the traces into 70%, 20% and 10% for training, testing, and validation, respectively. This is used for tuning hyperparameters (see α in Formula 1 and β in Formula 2).

3.6 Discussion and Future Work

Monetary cost constraint. AutoBlox does not currently have monetary cost constraints. This is for three major reasons. First, we cannot access the detailed manufacturing costs for each SSD component, as they are confidential. Second, the monetary cost is partially determined by economic factors that usually fluctuate according to market dynamics. Third, according to the market reports, the average cost of an SSD is \$0.09-\$0.2 per GB, which simply implies that we can estimate the SSD cost based on its capacity. AutoBlox has already had the SSD capacity constraint, which implicitly enforces the cost constraint. However, given a detailed cost model of SSD components, AutoBlox can incorporate them into its framework and guide the search process of optimized configurations. We wish to work on this as future work in collaboration with industry partners.

Whole-system performance. The goal of AutoBlox is to learn optimized SSD configurations. Just like SSD vendors nowadays, AutoBlox focused on device-level performance optimizations. With

Table 1: Performance of learned configurations for NVMe-based MLC SSDs (normalized to Intel 750 SSD). We list the target workloads in the first row. The first column shows the non-target workloads. The numbers show the speedup of the storage latency/throughput. AutoBlox delivers the best performance for target workloads (bold numbers). We also list the maximum performance of target workloads without considering the non-target workloads (ignore non-target).

Target workload →	Recomm	KVStore	Database	WebSearch	BatchAnalytics	CloudStorage	LiveMaps
Recomm	**1.28/1.07**	1.18/1.02	1.22/1.01	1.00/1.01	1.19/1.05	1.18/1.03	1.23/1.05
KVStore	1.21/1.19	**1.32/1.26**	1.21/1.18	0.97/0.98	1.25/1.19	1.23/1.19	1.20/1.16
Database	1.14/1.14	1.15/1.14	**1.45/1.43**	1.00/1.00	1.15/1.14	1.14/1.13	1.14/1.11
WebSearch	1.03/0.54	0.92/0.89	0.64/0.88	**1.25/1.00**	0.89/0.55	0.93/0.55	0.90/0.61
BatchAnalytics	1.28/1.00	1.21/1.00	1.15/1.00	1.12/1.01	**1.33/1.01**	1.24/1.00	1.21/1.00
CloudStorage	1.52/1.01	1.64/1.01	1.38/1.01	1.10/1.00	1.76/1.01	**1.93/1.01**	1.78/1.01
LiveMaps	1.38/1.24	1.11/1.07	1.26/1.14	0.95/0.98	1.33/1.19	1.40/1.21	**1.82/1.35**
Geometric mean (speedup of non-target workloads)	**1.26/1.00**	**1.10/1.02**	**1.11/1.03**	**1.02/1.03**	**1.24/1.00**	**1.19/1.01**	**1.14/1.00**
Maximum speedup of target workloads (ignore non-target)	**1.44/1.03**	**1.46/1.37**	**1.54/1.51**	**1.36/1.00**	**1.35/1.00**	**1.99/1.01**	**1.86/1.37**
Geometric mean of non-target workloads (ignore non-target)	1.05/1.06	1.22/0.97	1.11/1.01	0.91/0.90	1.22/1.00	1.14/0.99	1.11/0.98
Worst performance of non-target workloads (ignore non-target)	0.45/0.98	0.90/0.55	0.48/0.96	0.47/0.55	0.85/0.54	0.90/0.54	1.03/0.55

Table 2: Application workloads used in our evaluation.

Workload Category	Description
BatchAnalytics	MapReduce workloads running in data centers.
CloudStorage	Cloud storage workloads running in data centers.
KVStore	YCSB benchmarks are executed against RocksDB.
LiveMaps	LiveMaps workloads running on enterprise servers.
Database	TPCC executed against Windows SQL Server.
WebSearch	WebSearch services trace from UMassTraceRepository.
Recomm	Recommendations workloads running on servers.

Table 3: New storage workloads used in our evaluation.

Workload Category	Description
VDI	Storage workloads on virtual desktop infrastructure.
FIU	Traces from servers in Florida International University.
RadiusAuth	Traces from MicroSoft RADIUS authentication server.
LevelDB	A YCSB workload executed against LevelDB.
MySQL	TPCH benchmark executed on MySQL database.
HDFS	Storage workloads running in a distributed filesystem.

the learned configurations, it helps vendors produce new SSDs with improved performance for target workloads. To ensure improved end-to-end storage performance, end users can apply software optimizations such as multi-threaded I/O scheduling and increased page cache in OS, however, this is beyond the scope of this paper. As future work, we wish to explore learning techniques for whole-system performance optimizations.

4 EVALUATION

We show that: (1) AutoBlox can learn optimized SSD configurations for a given workload, and the learned configurations can deliver improved efficiency, compared with commodity SSD configurations (§4.2); (2) It can instantly learn an optimized configuration with low performance overheads (§4.3); (3) It works efficiently under different configuration constraints (§4.4); (4) It can conduct what-if analysis to identify optimized configurations for a given performance target (§4.5); and (5) AutoBlox itself is also tunable for satisfying various performance requirements from end users (§4.6).

4.1 Experimental Setup

In our evaluation, we use 7 different workload types as shown in Table 2. We also use 6 different new workloads (see Table 3) that have not been studied in AutoBlox to examine the generality of AutoBlox. These workloads cover various workloads that include key-value stores, databases, map services, advertisement recommendations, batch data analytics, enterprise file servers, and etc.

We run the AutoBlox on a server configured with 24 Intel Xeon CPU (E5-2687W v4) processors running at 3.0GHz, 96GB DRAM, and 4TB SSD. Since AutoBlox uses statistic learning models, it does not require GPUs. We use the configurations of Intel 750 SSD [8], Samsung 850 PRO SSD [9], and Z-SSD [10] as the baselines. AutoBlox uses these baselines to initialize the model. After that, AutoBlox compares the learned configurations with them to evaluate the learning efficiency. We believe they are reasonable baselines, since they mostly target data-intensive applications, and have been widely adopted in various computing platforms.

4.2 Efficiency of Learned Configurations

We first evaluate the efficiency of the learned configurations by AutoBlox. We use the Intel 750 SSD as the baseline. We set the configuration constraints as [SSD capacity = 512GB, interface = NVMe, flash type = MLC]. We warm up the SSDs by running random workloads, which will occupy at least 50% of the storage capacity. With its configuration in the SSD simulator, we run all the workloads in Table 2 to measure their performance. After that, we use the reference configuration and the measured performances to initialize AutoDB. Then, we feed the traces from different workload types into AutoBlox to learn new configurations.

We show the performance of learned configurations in Table 1. Compared to Intel 750 SSD, the learned SSD configurations reduce the storage latency by 1.25-1.93× for the target workload, while decreasing the storage latency by 1.15× on average for non-target workloads. The learned configurations can also improve the storage throughput by up to 1.43× for bandwidth-intensive applications such as the database and cloud storage. AutoBlox ensures that its learned configuration does not hurt the overall performance of other workloads. It cannot guarantee not to hurt non-target workloads. This is reasonable as AutoBlox focuses on the target workload while using coefficient factors to minimize the negative impact on non-target workloads.

We also show the best performance of target workload without considering the performance of non-target workloads, where we further improve the performance of target workloads by up to 16%. With software-defined SSDs, users may prefer maximizing the performance of their target workloads, without the need for considering non-target workloads. For example, cloud platforms that dedicatedly serve Database-as-a-Service would like to achieve the maximum performance for database workloads, without considering the performance of other workloads. These configurations slightly decrease the performance of non-target workloads. Our sensitivity analysis of β (see the Appendix) revealed that as we optimize performance for target workloads, we may not always hurt the performance of non-target workloads as some performance-critical

Table 4: Performance of learned configurations for NVMe-based MLC SSDs (normalized to Intel 750 SSD) for new (unseen) storage workloads. The numbers represent the speedup of the storage latency/throughput for different workloads. AutoBlox always delivers the best performance for the target workload (bold numbers).

Target workload →	LevelDB	MySQL	HDFS	VDI	FIU	RadiusAuth
LevelDB	**1.34/1.27**	1.05/1.04	1.00/1.00	1.02/1.03	1.24/1.20	1.01/1.00
MySQL	1.15/1.16	**1.45/1.43**	0.99/0.98	1.34/1.38	1.14/1.14	1.28/1.29
HDFS	1.41/1.00	1.23/1.00	**1.53/1.01**	1.12/1.00	1.36/1.04	1.10/1.09
VDI	1.27/1.16	1.30/1.17	1.05/1.03	**1.38/1.28**	1.13/1.08	1.17/1.10
FIU	1.13/1.00	0.99/1.00	1.16/1.00	1.08/1.09	**1.39/1.00**	1.08/1.06
RadiusAuth	1.03/1.00	1.43/1.00	1.08/1.00	1.32/1.00	1.23/1.00	**1.52/1.00**
Geometric mean (speedup of non-target workloads)	**1.19/1.06**	**1.19/1.04**	**1.05/1.00**	**1.17/1.09**	**1.22/1.08**	**1.12/1.10**

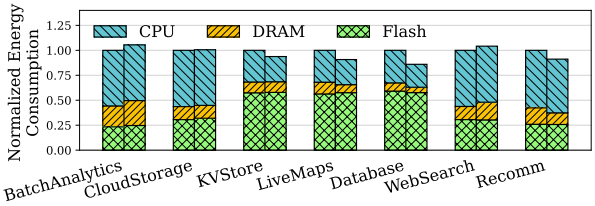

Figure 7: Energy consumption of learned SSD configurations, in comparison with the baseline SSD (Intel 750 SSD). Left: baseline SSD, Right: learned SSD.

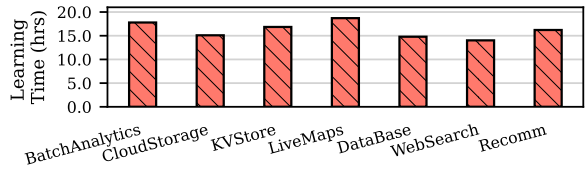

Figure 8: Learning time for different target workloads.

parameters are sensitive to both target and non-target workloads. We still need β, since it may still hurt the performance of some non-target workloads. For instance, in the worst case, a non-target workload can drop its performance by 56%. For some target workloads, we do not observe much improvement, this is because even though we enlarge the search space by removing the bound for non-target workloads, we do not identify a better configuration.

We also show the energy consumption of the learned SSD configurations in Figure 7. Compared to the baseline (Intel 750 SSD), the learned SSD configurations of AutoBlox can deliver up to 1.16× energy reduction, and at most 5% percent energy increase among all the workloads. The improvement of energy consumption is mainly because the learned configurations update the chip layout (e.g., the number of flash channels) accordingly and improve the data access bandwidth, which saves the CPU cycles in the SSD controller.

We list the critical parameters of the learned configurations in Table 5, in comparison with the reference configuration of Intel 750 SSD. For different target workloads, AutoBlox will learn different parameter values. We manually tuned these parameters and confirmed that AutoBlox selected correctly. Beyond the storage workloads in Table 2, we further examine the learning efficiency of AutoBlox with new workloads. As shown in Table 3, some of these workloads do not belong to any of the workload types (see Figure 2). The three workloads LevelDB, MySQL, and HDFS can be clustered into the studied workload types KVStore, Database, and CloudStorage respectively, but their traces are new. For the new traces, the average distance between the cluster center and existing clusters is 2.2× of the diameter of existing clusters. This

Table 5: The list of critical parameters in the learned configurations for different target workloads. RC: Recommendations, KV: KVStore, DB: Database, WS: WebSearch, BD: BatchAnalytics, CS: CloudStorage, LM: LiveMaps.

Parameters	Intel 750	RC	KV	DB	WS	BD	CS	LM
CMTCapacity	256MB	384MB	384MB	384MB	64MB	640MB	128MB	256MB
DataCacheSize	800MB	672MB	672MB	672MB	996MB	416MB	928MB	800MB
FlashChannelCount	12	32	10	10	32	32	32	16
ChipNoPerChannel	5	2	8	8	2	2	2	5
DieNoPerChip	8	2	1	2	8	1	1	4
PlaneNoPerDie	1	8	8	3	1	16	16	2
BlockNoPerPlane	512	512	512	512	512	512	512	256
PageNoPerBlock	512	256	384	512	512	256	256	768

Table 6: Overhead sources of AutoBlox.

Component	Execution Time (secs)
Extract workload features per 100K I/O requests	0.84
Workload Similarity Comparison	4.65
Workload Clustering	0.57
AutoDB Database Lookup	0.02
New configuration learning per iteration	2.75
Efficiency Validation	670.89

confirms that these new workloads are completely different from previous workloads, which helps us to further examine the model efficiency. We report the performance improvement of learned configurations of AutoBlox in Table 4. The learned configurations achieves 1.34–1.53× performance improvement for target workloads, and delivers 1.12× improvement on average for non-target workloads. This shows the generality of AutoBlox, AutoBlox can learn optimized configurations for unseen workloads that either fall into existing category or belong to a new category.

Although SSD vendors can have general optimizations for read or write-intensive workloads, the coarse-grained optimization approach cannot capture rich semantics of data access patterns, resulting in suboptimal SSD configurations. For example, BatchAnalytics (97.8% Read) and WebSearch (99.9% Read) are both read intensive workloads, AutoBlox shows that they can have different optimized configurations for improved performance (see Table 1 and Table 5).

4.3 Learning Efficiency of AutoBlox

We first examine the learning time of AutoBlox for different target workloads. As shown in Figure 8, AutoBlox can learn an optimized configuration in 14.02 - 18.71 hours. It incurs 89 search iterations on average to pinpoint an optimized configuration. We profile the execution time of its critical components on the multi-core server as described in §4.1, and show the results in Table 6. AutoBlox can finish each search iteration within 2.75 seconds. Its major performance overhead comes from the efficiency validation. AutoBlox only needs to validate the configuration selected by GPR in each search iteration, for reducing the validation overhead.

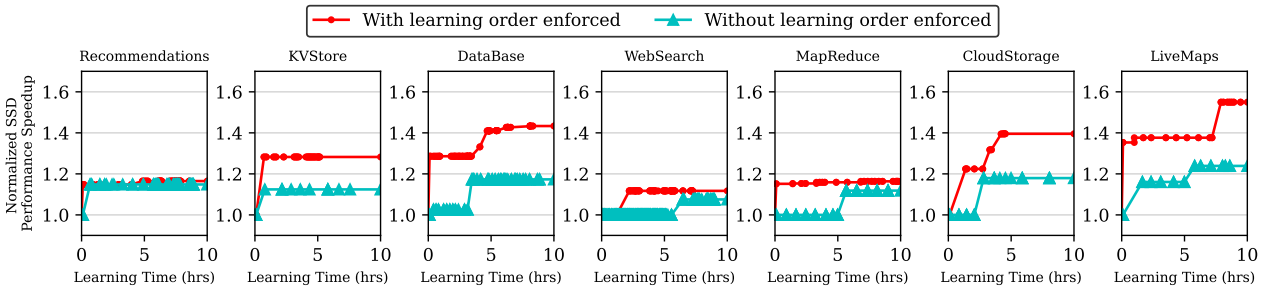

Figure 9: Reduced learning time of AutoBlox as we enforce the learning order of device parameters.

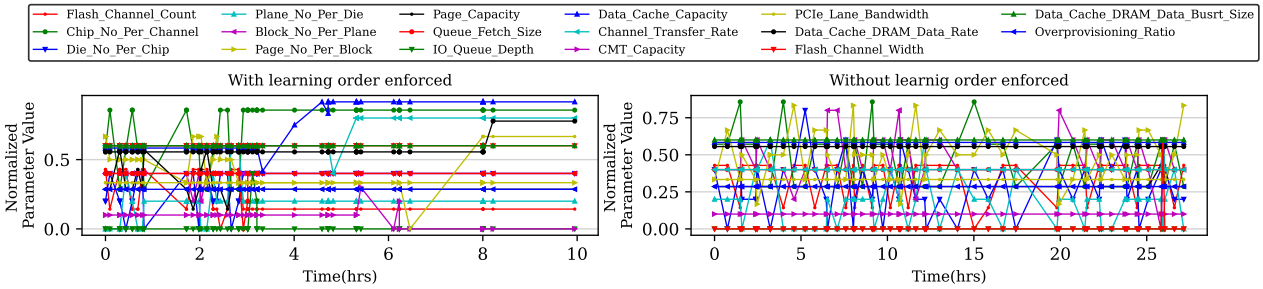

Figure 10: The learning procedure of AutoBlox for Database workload. Left/Right: with/without the learning order.

Table 7: The optimized configurations using what-if analysis for target workloads with given performance targets.

Parameters	Baseline	Boundaries	#Values	Latency Reduction by 3×		Throughput Improvement by 3×	
				VDI	WebSearch	Database	KVStore
DataCacheSize	800 MB	0 - 2048 MB	32	1056 MB	1248 MB	672 MB	800 MB
CMTSize	256 MB	0 - 2048 MB	32	512 MB	320 MB	384 MB	512 MB
ChannelWidth	8 bit	8 - 32 bit	3	8 bit	8 bit	32 bit	16 bit
ChannelTransRate	333 MT/s	67 - 1200 MT/s	13	333 MT/s	333 MT/s	800 MT/s	1200 MT/s
AvgPageReadLat	83 us	41 - 83 us	43	50 us	41 us	83 us	46 us
AvgPageWriteLat	1166 us	583 - 1166 us	584	700 us	1166 us	583 us	583 us
NumberofChannel	12	1 - 64	64	16	32	64	32
ChipPerChannel	5	1 - 64	64	5	2	2	4

Moreover, AutoBlox applied the learning order as studied in §3.3 to improve its learning efficiency. We profile the learning procedures for different target workloads, as shown in Figure 9. AutoBlox can always learn an optimized configuration that delivers better performance than the case without applying the learning order. The enforced learning order enables AutoBlox to focus on the optimization space search for the most relevant parameters, thus, its learning procedure can be converged in a shorter time (see Figure 10).

With this study, we confirm that (1) for each parameter, its correlation with the SSD performance is different, making it hard for developers to manually tune them; (2) not all parameters are equal, some parameters are insensitive to SSD performance. AutoBlox can help developers identify such parameters for different target workloads, and improve the productivity of SSD development.

4.4 Sensitivity to Configuration Constraints

We now evaluate how AutoBlox performs as we change the configuration constraints that include the flash types and device interface. To evaluate the sensitivity to flash types, we use Samsung Z-SSD, which is an NVMe SLC SSD, as the reference configuration. To evaluate the sensitivity to device interface, we use Samsung 850 PRO, which is a SATA MLC SSD, as the reference configuration.

We present the performance of the learned configurations for different workloads in Table 8 and Table 9, respectively. AutoBlox can learn optimized configurations for the target workloads, although the constraints have been changed. Table 8 shows that the configurations learned by AutoBlox can reduce the storage latency by up to 2.46×, and improve the storage throughput by up to 1.92×, compared to the Samsung Z-SSD. Table 9 demonstrates that our learned configurations can deliver up to 2.45× latency reduction and up to 1.58× throughput improvement for SATA SSDs for the target workload, in comparison with Samsung 850 PRO.

4.5 What-If Analysis with AutoBlox

We also show that AutoBlox can conduct what-if analysis to help developers to identify the optimized configurations for a given performance target for a target workload. These reported configurations can serve as the reference parameters as SSD vendors develop the next-generation SSDs. To support what-if analysis, we set more aggressive bounds for the hardware parameters (e.g., a higher upper bound for the number of channels) to explore a larger design space. These values may not be realistic today, but we foresee they would be possible with advanced manufacturing technologies, or they would be the target of future memory technologies.

As shown in Table 7, we use the configurations of Intel 750 SSD as the baseline, and set the boundaries for the tunable parameters in AutoBlox. We use two latency-sensitive workloads (VDI and WebSearch) and two throughput-intensive workloads (Database and KVStore) as the targets respectively. In the what-if analysis, we use the performance of Intel 750 SSD as the reference, and set the performance targets of latency-sensitive workloads as 3× latency reduction on average, and throughput-intensive workloads as 3× improvement on average, respectively. The configuration constraints are the same as indicated in §4.2. We list the optimized

Table 8: Performance of learned configurations for NVMe SLC SSDs (normalized to Samsung Z-SSD).

Target Workload →	Recomm	KVStore	Database	WebSearch	BatchAnalytics	CloudStorage	LiveMaps
Recomm	**2.38/1.04**	0.98/0.99	1.21/1.01	1.53/1.01	0.94/1.00	1.72/1.04	1.11/0.99
KVStore	1.83/1.38	**2.46/1.57**	2.08/1.47	1.13/1.12	1.33/1.17	1.23/1.13	1.46/1.24
Database	1.04/1.03	1.18/1.17	**1.84/1.71**	1.01/1.00	1.58/1.53	1.29/1.26	1.63/1.54
WebSearch	1.00/1.00	0.64/1.00	0.74/1.00	**1.08/1.92**	0.44/1.00	0.46/1.00	0.45/1.00
BatchAnalytics	1.06/1.00	1.15/1.17	0.78/1.07	1.24/1.07	**1.32/1.07**	0.99/1.07	1.24/1.07
CloudStorage	1.07/1.00	0.93/1.00	0.96/1.00	0.78/1.00	1.16/1.00	**1.22/1.00**	1.13/1.00
LiveMaps	0.72/0.94	1.01/1.00	1.36/1.04	0.49/0.86	1.53/1.04	1.33/1.03	**1.86/1.07**
Geometric mean (speedup of non-target workloads)	**1.08/1.05**	**1.00/1.04**	**1.11/1.09**	**1.00/1.01**	**1.12/1.14**	**1.09/1.09**	**1.10/1.12**

Table 9: Performance of learned configurations for SATA MLC SSDs (normalized to Samsung 850 PRO).

Target Workload →	Recomm	KVStore	Database	WebSearch	BatchAnalytics	CloudStorage	LiveMaps
Recomm	**1.23/1.07**	1.13/1.03	1.05/1.01	1.06/1.01	1.07/1.01	1.13/1.03	1.09/1.03
KVStore	2.13/1.60	**2.45/1.58**	0.76/0.67	1.15/0.74	0.87/0.51	2.03/1.70	1.65/0.90
Database	1.01/1.00	0.92/1.00	**1.01/1.18**	0.99/1.14	0.98/1.12	0.98/1.00	1.01/1.00
WebSearch	0.94/1.00	0.89/1.00	1.13/1.13	**1.13/1.14**	1.12/1.10	0.98/1.00	0.95/1.00
BatchAnalytics	0.82/1.00	1.03/1.00	1.16/1.00	1.15/1.00	**1.17/1.00**	1.03/1.00	0.99/1.00
CloudStorage	1.10/1.01	1.14/1.01	1.04/1.00	1.04/1.02	1.05/1.00	**1.40/1.01**	1.06/1.00
LiveMaps	1.42/1.31	1.62/1.43	1.14/1.16	1.03/1.04	1.40/1.32	1.59/1.46	**1.83/1.46**
Geometric mean (speedup of non-target workloads)	**1.18/1.13**	**1.10/1.07**	**1.02/1.00**	**1.07/1.00**	**1.00/1.00**	**1.18/1.15**	**1.11/1.00**

values of the most critical parameters in Table 7. AutoBlox can identify the optimized configurations for different target workloads within 121 iterations on average, out of a large exploration space of 4.11 trillion configuration combinations. Given a target workload and a target performance, it will be extremely difficult to manually identify the optimized configurations, if they wish to understand how the SSD device should be advanced to meet the performance goals. AutoBlox shows that SSD DRAM capacity, device read/write latency, and chip layout are all important for reducing storage latency. It also shows that the data transfer rates in each channel and chip layout are critical for improving storage throughput.

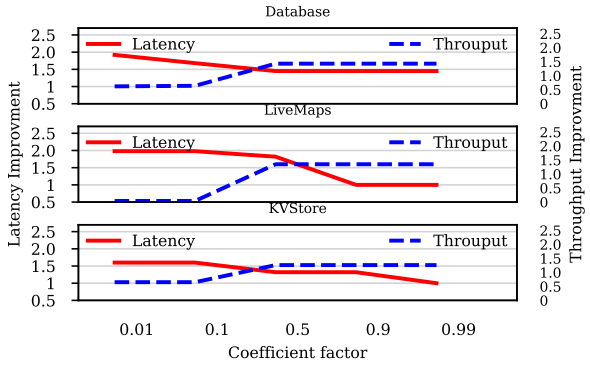

Figure 11: Performance impact of the coefficient factor for balancing the latency and throughput for a target workload.

4.6 Impact of the Balance Coefficient

As discussed in §3.2 and §3.4, AutoBlox uses the coefficient factor α (Formula 1) to balance the storage latency and throughput in the learning procedure, and defines the coefficient factor β (Formula 2) to balance the penalty (weight) between the target workload and non-target workloads. Both of them are tunable in AutoBlox, which allows end users to adjust them per their needs. In this part, we evaluate their impact on storage performance. We vary their values from 0.01 to 0.99, and measure the performance of the learned configurations for the three representative workloads Database, Key-Value store, and LiveMaps. As we examine each value of α and β, we reset the ML model and initialize the AutoDB.

With the coefficient factor α, our goal is to achieve the maximum improvement for both latency and throughput. In Figure 11, as we increase the value of α from 0.01 to 0.1, the latency of the target workload is dramatically reduced, however, its throughput is lower than the reference configuration. As we further increase its value to 0.5, we can achieve both improved latency and throughput for all the three target workloads. Thus, AutoBlox sets $\alpha = 0.5$ by default.

With the coefficient factor β, our goal is to achieve the maximum performance improvement for both target workload and non-target workloads. As we vary the value of β, we observe that there is a sweet spot ($\beta = 0.1$) that delivers maximum improvements for both target workload and non-target workloads (Figure 12). Thus, we set $\beta = 0.1$ by default.

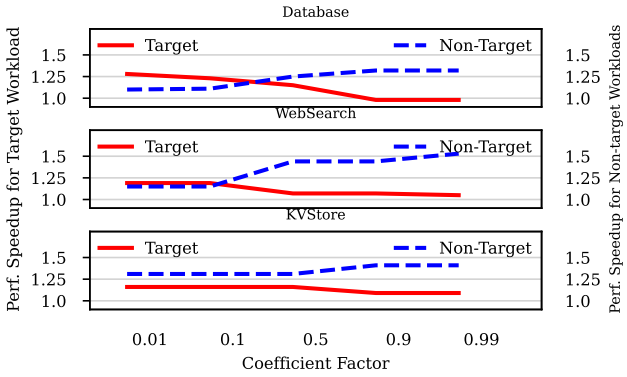

Figure 12: A study of the coefficient factor for balancing the performance between the target and non-target workloads.

5 RELATED WORK

SSD Performance Optimizations. SSDs have been widely used in modern storage systems to meet the performance and capacity requirements from applications [24, 26, 33, 36, 68]. Although many applications have unique data access patterns [5, 23, 69], they normally employ generic SSDs, causing suboptimal performance and resource inefficiency. Researchers proposed the software-defined flash to enable applications to develop their own storage stack [25,

34, 38, 47]. However, there is a longstanding gap between the application demands and device specifications. We develop AutoBlox with the goal of bridging this gap.

Machine Learning for Systems. Most recently, researchers leveraged ML techniques to solve system optimization problems, such as the task scheduling [43, 59, 70], performance optimizations [22, 37, 54, 71], and others. However, few studies conduct a systematic investigation of applying the learning techniques to develop SSDs. To the best of our knowledge, AutoBlox is the first work that utilizes the learning techniques to automate tuning of SSD specifications.

SSD Device Development. The industry has developed mature manufacturing techniques and fabrication process to produce new storage devices, such as Z-SSD [50], Optane SSD [27], ZNS SSDs [60]. In the era of Industry 4.0/5.0 powered by AI, storage devices should become highly customizable. In this work, we focus on building a learning-based framework aiming towards this goal.

6 CONCLUSION

We build a learning-based framework AutoBlox for enabling the automated tuning of SSD configurations. Given a storage workload, AutoBlox can efficiently learn an optimized SSD configuration under different configuration constraints, which significantly reduces manual efforts in SSD device development. Our experiments show that the learned SSD configurations can maximize the performance improvement for both target workload and non-target workloads.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments and feedback. This work was partially supported by the NSF CAREER Award CNS-2144796 and a gift fund from SK Hynix.

REFERENCES

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *Proceedings of the 2008 USENIX Annual Technical Conference (ATC'08)*. Boston, Massachusetts.

[2] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Billian, and Andrew Pavlo. 2021. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1241–1253.

[3] Amazon. 2021. AWS Web Services. https://aws.amazon.com/.

[4] Matt Asay. 2022. New Gartner report shows massive growth in the database market, fueled by cloud. https://www.techrepublic.com/article/new-gartner-report-shows-massive-growth-database-market-fueled-cloud/.

[5] Jean Luca Bez, Francieli Zanon Boito, Ramon Nou, Alberto Miranda, Toni Cortes, and Philippe O. A. Navaux. 2019. Detecting I/O Access Patterns of HPC Workloads at Runtime. In *Proceedings of the 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'19)*.

[6] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.

[7] Steve Burke. 2012. How SSDs Are Made: Phases of Solid-State Drive Development. https://www.gamersnexus.net/guides/956-how-ssds-are-made.

[8] CAMELab. 2018. Intel 750 SSD specifications. https://docs.simplessd.org/en/v2.0.12/_downloads/c1f977aafe4072ae4a21eae026d502ef/intel750_400gb.cfg.

[9] CAMELab. 2018. Samsung 850 Pro SSD specifications. https://docs.simplessd.org/en/v2.0.12/_downloads/f9180ac639ce600a54f6d2e982207edd/samsung_850pro_256gb.cfg.

[10] CAMELab. 2018. Samsung Z-SSD specifications. https://docs.simplessd.org/en/v2.0.12/_downloads/044b4e1a1bb1e3ba37b3709033f9bf63/samsung_zssd_800gb.cfg.

[11] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. 2018. Towards Better Understanding of Black-box Auto-Tuning: A Comparative Analysis for Storage Systems. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*. Boston, MA.

[12] Laura Caulfield. 2018. Project Denali to define flexible SSDs for cloud-scale applications. https://azure.microsoft.com/en-us/blog/project-denali-to-define-flexible-ssds-for-\cloud-scale-applications/.

[13] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*. IEEE.

[14] Open Compute. 2021. The Open Compute Project. https://www.opencompute.org/.

[15] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[16] FinancesOnline. 2022. 60 Notable Machine Learning Statistics: 2021/2022 Market Share and Data Analysis. https://financesonline.com/machine-learning-statistics/.

[17] Gem5. 2022. ARM Power Modelling. https://www.gem5.org/documentation/learning_gem5/part2/arm_power_modelling/.

[18] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. 2018. Amber*: Enabling precise full-system simulation with detailed modeling of all SSD resources. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 469–481.

[19] Laura M Grupp, Adrian M Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H Siegel, and Jack K Wolf. 2009. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 24–33.

[20] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. Washington, DC, USA.

[21] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. 2017. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. Shanghai, China.

[22] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. 2020. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*.

[23] Jun He, John Bent, Aaron Torres, Gary Grider, Garth Gibson, Carlos Maltzahn, and Xian-He Sun. 2013. I/O Acceleration with Pattern

Detection. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC'15)*. New York, New York, USA, 25–36.

[24] Yang Hu, Hang Liu, and H. Howie Huang. 2018. TriCore: Parallel Triangle Counting on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)* (Dallas, Texas).

[25] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. 2017. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. Santa Clara, CA.

[26] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. 2015. Unified Address Translation for Memory-mapped SSDs with FlashMap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. Portland, OR.

[27] Intel Corporation. 2018. Intel® Optane™ SSD DC P4801X Series. (2018).

[28] Luyi Kang, Yuqi Xie, Weiwei Jia, Xiaohao Wang, Jongryool Kim, Changhwan Youn, Myeong Joon Kang, Jin Lim, Bruce Jacob, and Jian Huang. 2021. IceClave: A Trusted Execution Environment for In-Storage Computing. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'21)*. Virtual Event.

[29] Jiho Kim, Seokwon Kang, Yongjun Park, and John Kim. 2022. Networked SSD: Flash Memory Interconnection Network for High-Bandwidth SSD. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 388–403.

[30] Joonsung Kim, Pyeongsu Park, Jaehyung Ahn, Jihun Kim, Jong Kim, and Jangwoo Kim. 2018. SSDcheck: Timely and accurate prediction of irregular behaviors in black-box SSDs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 455–468.

[31] Kingston. 2021. Design-in Solid State Drives for System Designers and Builders. https://www.kingston.com/unitedstates/us/embedded/design-in-ssd.

[32] Andreas Krause and Cheng Soon Ong. 2011. Contextual Gaussian Process Bandit Optimization. In *Proceedings of the 24th International Conference on Neural Information Processing Systems (NIPS'11)*. Granada, Spain.

[33] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proc. 10th USENIX OSDI*. Hollywood, CA.

[34] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. 2016. Application-Managed Flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. Santa Clara, CA.

[35] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. 2017. Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'17)*.

[36] Hang Liu and H. Howie Huang. 2019. SIMD-X: Programming and Processing of Graph Algorithms on GPUs. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*. Renton, WA.

[37] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-Based Memory Allocation for C++ Server Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. Lausanne, Switzerland.

[38] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-defined Flash for Web-scale Internet Storage Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. Salt Lake City, UT.

[39] Jisung Park, Myungsuk Kim, Myoungjun Chun, Lois Orosa, Jihong Kim, and Onur Mutlu. 2021. Reducing solid-state drive read latency by optimizing read-retry. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 702–716.

[40] PCISIG. 2022. PCIe Specification: The High-Speed Interconnect of Choice is Evolving. https://pcisig.com/specifications.

[41] Prashanth Ashok. [n. d.]. What is Ridge Regression? https://www.mygreatlearning.com/blog/what-is-ridge-regression/.

[42] PyTorch Team. 2021. PyTorch: From Research to Production. https://pytorch.org/.

[43] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*.

[44] Nadig Rakesh, Sadrosadati Mohammad, Mao Haiyu, Mansouri Ghiasi Nika, Tavakkol Arash, Park Jisung, Sarbazi-Azad Hamid, Gómez Luna Juan, and Mutlu Onur. 2023. Venice: Improving Solid-State Drive Parallelism at Low Cost via Conflict-Free Accesses. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ACM. https://doi.org/10.1145/3579371.3589071

[45] Carl Edward Rasmussen and Christopher K. I. Williams. 2006. *Gaussian Processes for Machine Learning*. The MIT Press.

[46] Benjamin Reidys, Peng Liu, and Jian Huang. 2022. RSSD: Defend Against Ransomware with Hardware-Isolated Network-Storage Codesign and Post-attack Analysis. In *In Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*. Lausanne, Switzerland.

[47] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi Noghabi, and Jian Huang. 2022. BlockFlex: Enabling Storage Harvesting with Software-Defined Flash in Modern Cloud Platforms. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA.

[48] Allied Market Research. 2022. Hadoop Market Statistics:2030. https://www.alliedmarketresearch.com/world-hadoop-market.

[49] scikit-learn. 2021. White Kernel. https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.kernels.WhiteKernel.html.

[50] Samsung Semiconductors. 2018. Ultra-Low Latency with Samsung Z-NAND SSD. *Technical Report* (2018).

[51] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. Broomfield, CO.

[52] Systems Software and Architecture Lab. 2022. OpenSSD. http://openssd-project.org/.

[53] Niranjan Srinivas, Andreas Krause, Sham Kakade, and Matthias Seeger. 2010. Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design. In *Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML'10)*. Haifa, Israel.

[54] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. 2023. LeaFTL: A Learning-based Flash Translation Layer for Solid-State Drives. In *In Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*. Vancouver, Canada.

[55] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. 2018. MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. Oakland, CA.

[56] TrustRadius. 2021. Database-as-a-Service. https://www.trustradius.com/database-as-a-service-dbaas.

[57] 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 (SIGMOD'17)*. Chicago, Illinois, USA.

[58] Xiaohao Wang, You Zhou, Chance C. Coats, and Jian Huang. 2019. Project Almanac: A Time-Traveling Solid-State Drive. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*. Dresden, Germany.

[59] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys'21)*.

[60] WDC. 2020. ZNS SSDs Just Got Real - Ultrastar DC ZN540 Now Sampling. https://blog.westerndigital.com/zns-ssd-ultrastar-dc-zn540-sampling/.

[61] Wikipedia. 2023. Radial basic function kernel. https://en.wikipedia.org/wiki/Radial_basis_function_kernel.

[62] Wikipedia. 2023. Rational Quadratic Covariance Function. https://en.wikipedia.org/wiki/Rational_quadratic_covariance_function.

[63] Wikipedia. 2023. Stochastic Gradient Descent. https://en.wikipedia.org/wiki/Stochastic_gradient_descent.

[64] Wikipedia. 2023. Taxicab Geometry: Manhattan Distance. https://en.wikipedia.org/wiki/Taxicab_geometry.

[65] Jiwon Woo, Minwoo Ahn, Gyusun Lee, and Jinkyu Jeong. 2021. {D2FQ}:{Device-Direct} Fair Queueing for {NVMe}{SSDs}. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 403–415.

[66] Zakaria Jaadi. [n. d.]. A Step-by-Step Explanation of Principal Component Analysis (PCA). https://builtin.com/data-science/step-step-explanation-principal-component-analysis.

[67] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. Restune: Resource oriented tuning boosted by meta-learning for cloud databases. In *Proceedings of the 2021 international conference on management of data*. 2102–2114.

[68] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST'15)*. Santa Clara, CA.

[69] Qing Zheng, Kai Ren, Garth Gibson, Bradley W. Settlemyer, and Gary Grider. 2015. DeltaFS: Exascale File Systems Scale Better without Dedicated Servers. In *Proceedings of the 10th Parallel Data Storage Workshop (PDSW'15)*. Austin, Texas, 1–6.

[70] Zhiheng Zhong, Minxian Xu, Maria Alejandra Rodriguez, Chengzhong Xu, and Rajkumar Buyya. 2021. Machine Learning-based Orchestration of Containers: A Taxonomy and Future Directions. *CoRR* abs/2106.12739 (2021).

[71] Giulio Zhou and Martin Maas. 2021. Learning on Distributed Traces for Data Center Storage Systems. In *Proceedings of the Machine Learning and Systems (MLSys'21)*. Austin, TX.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact of AutoBlox is a learning-based framework that optimizes the design of software-defined SSDs. Specifically, this artifact includes the source code of the SSD simulator, the training model, and the necessary scripts to reproduce the experiments in our paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** AutoBlox proposed and implemented a customized Bayesian optimization algorithm.
- **Data set:** AutoBlox uses block-level I/O traces for different workloads, such as YCSB, TPCC, AdspayLoad, MapReduce, LiveMaps-BackEnd, WebSearch, and CloudStorage. We provide the links for downloading these block I/O traces.
- **Hardware:** The simulator does not require any special hardware. But we prefer machines with larger memory capacities (>120GB) for accelerating the learning process.
- **Disk space required:** The trace files and simulator outputs need about 100GB storage space.
- **Expected execution time of the experiments:** AutoBlox takes approximately 21 hours to finish both pruning and training for each workload. The pruning and training of each workload can be performed in parallel across multiple machines.
- **Publicly available:** Yes. The DOI of this artifact is 10.5281/zenodo.8332820. You can also access the latest version at https://github.com/platformxlab/AutoBlox.

A.3 Description

A.3.1 How to access. The latest version of this artifact can be accessed via this link: https://github.com/platformxlab/AutoBlox. The artifact is available at https://zenodo.org/record/8332820.

A.3.2 Hardware dependencies. Our experiments do not require any special hardware. Since some of our experiments will cause large memory footprints, it will be helpful if you run the experiments with machines having larger memory capacities (e.g., 128 GB).

A.3.3 Software dependencies. The artifact uses MQSIM and DRAMPower simulator, they require Python3 and relevant packages. We specify all the software dependencies in the README.

A.3.4 Data sets. Our experiments need several block-level I/O traces. The links for downloading the traces are available in the Github README.

A.4 Installation

The setup of the artifact is as follows. First, please clone the repository and download the trace files with the following commands:

```
cd AutoBlox_Artifact/src
python3 download.py
cd ..
unzip autoblox_traces.zip
mv autoblox_traces/* .
rm -r autoblox_traces/
unzip xdb_base.zip
cd src/
bash setup_xdb.sh
cd ..
```

Then, install the dependencies with the command:

```
bash server_setup_instructions.sh
```

More detailed instructions can be found in the README file.

A.5 Evaluation and expected results

There are three steps towards reproducing the results of AutoBlox.
(1) Workload Clustering. AutoBlox clusters different workloads with block I/O traces (as shown in Figure 2). To verify the workload clustering results, please run the following commands:

```
cd src
python3 clustering_motivation.py
cd ../reproduced_dat
python3 clustering.py
```

(2) Parameter Pruning. AutoBlox accelerates the tuning procedure by pruning insensitive parameters and by using the pruning results to enforce the tuning order. To verify the coarse-grained pruning and fine-grained pruning results, please run the following commands:

```
cd ../src
export target="target_workload"
bash run_pruning.sh
```

In our experiments, the target workloads refer to the workloads in Table 2. After running the above comments, please run the following commands:

```
cd ../reproduced_dat
python3 coarsed_grained.py
python3 fine_grained.py
```

With the above commands, you should be able to generate the results as shown in Figure 4 and Figure 5. These experiments demonstrate that different workloads have different performance sensitivity to SSD hardware parameters, and it is hard to manually tune these parameters. More detailed discussions can be found in Section 3.
(3) Learning Procedure. To start the learning procedure, please run the following commands:

```
cd src
find_best_conf.py target_workload use_tuning_order
    xdb_directory
```

In the above command, target workload refers to the workloads in Table 2, the use_tuning_order has two options: True and False, and xdb_directory is the xdb database used in AutoBlox. Since the learning process will take multiple hours, we suggest the users to run the experiments using several machines in parallel.

To reproduce Table 1, please run the following command:

```
python3 get_recommended_configurations.py xdb_directory
```

We can reproduce Figure 9 and Figure 10 with the commands:

```
cd ../reproduced_dat
python3 learning_profile.py
python3 tuning_time.py
```

These figures show that, with enforced tuning order, AutoBlox can accelerate the learning procedure. Note that the generated figures could be slightly different from Figure 9 and 10, due to the randomness in the model update in AutoBlox. However, you will observe that the learning procedure with enforced tuning order is more efficient that that without the enforced tuning order.