Shiro: Efficient and Accurate In-Storage Data Lifetime Separation for NAND Flash SSDs

Penghao Sun, Shengan Zheng, Litong You, Wanru Zhang, Ruoyan Ma,
Jie Yang, Feng Zhu, Shu Li, Linpeng Huang

***Abstract*—The log-structured nature of NAND flash storage necessitates garbage collection in SSDs. Garbage collection (GC) is a major source of runtime write amplification (WA), leading to faster device wear out and interference with host I/Os. The key to mitigating this problem is separating data by lifetime so that data in the same flash block are invalidated within temporal proximity. For higher lifetime prediction accuracy and adaptibility, prior works proposed using machine learning algorithms for data separation. However, existing learning-based solutions perform data lifetime prediction at the host side, leading to several drawbacks. First, host-side prediction does not have knowledge of the internal data movement inside the SSD during GC, and thus fails to leverage the opportunity to further separate GC writes, resulting in suboptimal WA reduction in the long term. Second, performing prediction at the host significantly prolongs the I/O critical path and consumes host resources that could otherwise be used for serving user applications.**

We present Shiro, a holistic FTL design that performs in-storage data separation for both user writes and GC writes for maximal long-term WA reduction. For user writes, Shiro uses a sequence model to accurately predict data lifetime by learning lifetime distribution from long historical access patterns. For GC writes, Shiro incorporates a reinforcement learning-assisted page migration strategy that takes direct feedback from long-term WA to further improve data separation efficacy. To address the challenges posed by performing fine-grained and real-time machine learning decisions inside the resource-constrained SSD, we propose a suite of enabling techniques to keep computation and storage overhead low. Extensive evaluation of Shiro on real-world traces shows that Shiro can deliver 29%-68% lower WA compared with conventional FTL and state-of-the-art in-storage data separation schemes. Furthermore, thanks to lower data migration overhead during GC, Shiro achieves significantly higher steady-state I/O performance.

***Index Terms*—solid state drive, data separation, write amplification, garbage collection, machine learning.**

I. Introduction

NAND flash-based solid state drives (SSDs) have seen wide deployment in consumer electronics and servers to serve as secondary storage. To provide higher capacity within small form factors, modern NAND flash technology is advancing towards higher storage density, with reduced endurance.. Notably, the transition from SLC to QLC flash increased density by fourfold, but the number of sustainable program/erase (P/E) cycles dropped drastically [1]. On the other hand, write amplification (WA) is an inherent challenge in SSDs due to the characteristics of flash storage, leading to excessive P/E cycle consumption and faster wear-out. A recent study highlighted that WA in real-world production environments can exceed 100, meaning the amount of data written to flash can be over $100\times$ the data written by users [2]. As a result, reducing WA in SSDs remains an important area of study.

Garbage collection (GC) is one of the primary contributors to WA in SSDs. Due to the 'erase-before-write' limitation, a flash translation layer (FTL) is implemented inside SSDs on top of raw flash, converting all write operations into append operations, thus making the SSD conceptually similar to a log-structured storage [3]. When the FTL exhausts its supply of free pages (unit of read and write; typically 4-32KB) for new writes, GC is triggered. This process involves selecting one or more blocks (unit of erase; typically hundreds of pages) as victims. The FTL copies the valid pages from victim blocks elsewhere before erasing the blocks to free up space. WA arises during this data copying process.

The key to minimizing WA from GC is separating data by lifetime (*data separation* hereafter) [4], [5]. By grouping pages with similar lifetimes into dedicated blocks, it becomes more likely that the pages within the same block will be invalidated around the same time. This increases the likelihood that the FTL can find victim blocks with fewer valid pages, effectively reducing WA.

Due to lack of future knowledge, we need to *predict* the lifetime of incoming data by inferring at what time a piece of data will be overwritten by the host. Most previous studies in this area rely on rule-based approaches built on simple heuristics [5]–[9], which lack adaptability and accuracy. Recent successes of the application of machine learning (ML) in system software [10]–[12] promise more adaptive and accurate data separation using ML models [13], [14]. However, existing solutions typically perform both model training and inference on the host, leading to several limitations.

First, due to the black-box nature of SSDs, the host does not have knowledge about the internal GC data placement. Therefore, it is only possible to perform data separation for *user writes*. Since prediction is not 100% accurate, mispredicted pages remain in the wrong lifetime group, hurting long-term WA. Although the SSD has the opportunity to further separate such pages to more suitable lifetime groups during *GC writes*,

Penghao Sun, Wanru Zhang, Ruoyan Ma and Linpeng Huang are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. Shengan Zheng is with the MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University. Litong You is with the School of Computer Science and Technology, Hangzhou Dianzi University. Jie Yang, Feng Zhu and Shu Li are with Alibaba Cloud. Shengan Zheng and Linpeng Huang are corresponding authors.

a host-side scheme fails to leverage such opportunity. Second, whereas training can be performed offline, prediction is tightly coupled with application I/Os because the dispatch of a write request must be postponed until the model has finished prediction. This puts model prediction on the I/O critical path and significantly degrades I/O latency as ML prediction is generally much slower than simple rule-based heuristics.

An in-storage learning-based approach that performs data lifetime prediction inside the SSD grants the data separation scheme first-hand knowledge of data placement and ensures the overhead does not consume extra host resources. However, introducing learning-based data separation into the resource-constrained SSD poses significant challenges. For *user writes*, the effectiveness of data separation is maximized when the lifetime of every page can be accurately predicted. Such fine-grained predictions imposes strict requirements on the efficiency of the ML model, particularly given that the write latency of modern SSDs can be as low as a few microseconds [15], [16]. While modern SSD controllers adopt multi-core architectures [17], they still lag behind in processing power compared with host-side hardware. For *GC writes*, the information available for deciding the page's migration destination for the purpose of lower future WA is extremely limited. Unlike user writes, where the host I/O pattern can be relied upon for accurate decision making, it is challenging to directly link the destination of page migration to its impact on future WA. Designing a GC data separation policy for lower long-term WA is therefore non-trivial. Furthermore, metadata management for the data separation scheme must avoid excessive consumption of on-device RAM, a limited resource due to its higher cost compared to flash memory.

We present Shiro, a holistic FTL design that performs in-storage data separation for both user writes and GC writes for maximal long-term WA reduction. For **user writes**, Shiro uses supervised machine learning model to perform fine-grained data separation. The model in Shiro is a lightweight sequence model that can extract information from prolonged historical access patterns of a page by learning from a time series of carefully selected features. For every written page, the model is able to accurately predict whether it is short-living or long-living. Shiro dynamically adjusts the classification criterion at runtime with an adaptive data labeling algorithm. For **GC writes**, Shiro incorporates a reinforcement learning-assisted page migration strategy that takes direct feedback from long-term WA for data separation. Specifically, the reinforcement learning agent dynamically steers valid pages in the blocks being GC'ed to multiple destinations and is given a higher reward for each decision of the migration destination if the observed WA of future GC operations is lower, thus evolving toward lower long-term WA. On the input side, the agent characterizes the valid pages based on a rich set of runtime statistics.

To address the runtime overhead incurred by the data separation scheme, Shiro introduces a set of optimization techniques. For computational costs, Shiro masks ML computation by leveraging the NVMe I/O process to remove prediction from the I/O critical path. This is achieved by parallelizing prediction with the data payload transmission and other internal tasks. Additionally, Shiro caches intermediate model prediction results (the hidden state) for each page. This allows Shiro to enjoy the full might of the sequence model without expensive iterative computation over long feature sequences. For storage costs, the on-board memory consumption by ML metadata, such as feature extraction information and the cached hidden state, is tackled with an efficient flash data layout. The proposed flash layout enables Shiro to store all ML metadata in flash and easily retrieve them in batches to RAM, building an on-demand cache for quick access. Together, the proposed techniques make real-time and fine-grained data separation inside the SSD possible with low computation and storage overhead. To the best of our knowledge, Shiro is the first FTL design that successfully deploys complex ML algorithms for in-storage data separation.

We prototype Shiro on a hardware-based SSD evaluation platform and evaluate it against state-of-the-art in-storage data separation schemes on real-world traces to showcase Shiro's superiority in reducing WA. We demonstrate that our proposed optimization techniques successfully mask the overhead of ML computation with almost no overhead perceived by the host. Furthermore, trace I/O evaluation confirms that lower WA can transform into substantial improvements in steady-state I/O performance. This work is based on our previous work published at DAC 2023 [18].

In summary, this paper makes the following contributions:

(1) We present Shiro, an FTL design that performs real-time and fine-grained data separation inside the storage device for both user writes and GC writes. Shiro uses a sequence model that can learn from prolonged history of I/O patterns to accurately separate short-living and long-living data written by the user, and a reinforcement learning-based data migration strategy to further separate GC'ed pages while taking direct feedback from long-term WA.

(2) We introduce a suite of enabling techniques to address the runtime overhead of the data separation scheme. Computation overhead is addressed by parallelizing prediction with payload transmission and a caching mechanism for intermediate results to avoid costly iterative ML computation. Storage overhead is tackled with an efficient flash data layout that enables the storage of all ML metadata in flash, which can be easily fetched to RAM in batches for fast serving at runtime.

(3) We prototype the proposed design on a hardware-based evaluation platform and evaluate it against state-of-the-art data separation schemes to demonstrate its effectiveness in reducing WA and low runtime overhead. We also show that lower WA effectively translates into substantial gains in steady-state I/O performance.

II. Background

A. SSD Preliminaries

Flash-based SSDs are built on arrays of NAND flash chips connected to an embedded controller through multiple channels. Each flash chip is organized into multiple levels of operational units, arranged in decreasing granularity as dies, planes, blocks, and pages. Flash dies operate independently and support multi-plane operations within a die. Read and

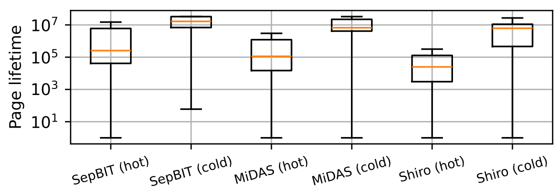


Fig. 1. Page lifetime distribution under different separation schemes.

write operations occur at the page level, and pages in a block must be written sequentially but can be read in any order. Once all pages in a block are written, the entire block must be erased before rewriting. To abstract these constraints from the host, the controller employs a flash translation layer (FTL) that converts all writes into appends. The FTL uses an L2P (logical-to-physical) table, typically stored at page granularity in the on-device RAM, to map logical page numbers (LPNs) to physical page numbers (PPNs) for optimal performance. Modern SSDs also utilize ”superblocks” as the basic management unit [19]. A superblock comprises all blocks with the same die offset. To leverage inter-die parallelism, the FTL allocates new pages in a round-robin manner from ”open” superblocks. Once fully allocated, these superblocks are marked as ”closed”, making them read-only and pending for GC.

B. Tackling GC Write Amplification

When free pages deplete, GC kicks in to reclaim obsolete pages. During GC, pages in the victim (super)blocks with valid data must be copied elsewhere. This copying process is a major source of WA in SSDs. Increased WA from GC accelerates device wear-out and interferes with host I/O operations [20], [21]. Addressing this issue has become a focal point for researchers in recent years. The key to reducing WA from GC is data separation, i.e., separating user-written data with different lifetimes [4]. When pages with similar lifetimes are grouped within the same blocks, they are more likely to become invalidated in shorter time windows. As a result, the FTL has a higher chance of finding a victim block with a low valid page count, thus leading to lower WA. In fact, if the lifetime of a host-written page is known in advance, an optimal data placement strategy could achieve zero WA by organizing pages in the order by which they are invalidated [5]. However, this method is impractical due to lack of future knowledge. Researchers thus developed methods to predict the lifetime of user data for effective data separation [5]–[9], [13], [14], [22]. Below, we briefly summarize existing data separation approaches.

Rule-based separation. Existing rule-based approaches include both host-side solutions [8], [9], [22] and device-side solutions [5]–[7]. Host-side solutions enjoy the benefit of richer information from user applications and the OS kernel. Specifically, AutoStream [22] monitors the lifetime of LBAs and directs those with similar lifetimes to dedicated I/O streams. FStream [8] leverages file system-level knowledge to separate metadata, journal, and file data writes. PCStream [9] further brings in application-level information by clustering writes according to their I/O call sites by referring to program stack signatures. Although information from higher layers in host-side software can potentially improve data separation efficacy, the application, OS, and device interface (e.g., multi-streamed SSDs [23] or ZNS SSDs [24]) must be adapted so that the separation result can be communicated to the device. Furthermore, since host-side solutions can only separate user writes, they miss the opportunity to further separate GC writes, which we show to be very effective in minimizing WA.

Device-side approaches are forced to use the limited knowledge regarding host I/O patterns inside the device, but have the advantage of full control over both user and GC data placement. Among them, 2R [7], SepBIT [5] and MiDAS [6] are the current state of the art. 2R [7] demonstrated that simply separating GC writes from user writes can deliver significant reduction in WA. SepBIT [5] uses the previously observed lifetime of a flash as the estimate of its future lifetime to separate user writes to hot/cold groups and GC writes to multiple levels of groups. MiDAS [6] also adopts SepBIT’s user data separation method, but applies a stricter admission policy for the hot group to avoid false positives. For GC writes, MiDAS builds a theoretical model to predict the WA under a given group configuration based on page lifetime distribution, and uses this model to find the optimal configuration.

Learning-based separation. Although rule-based prediction can quickly yield prediction result, it suffers from limited accuracy. Figure 1 shows the lifetime distribution of user-written pages classified as hot or cold by SepBIT, MiDAS and the ML model in this work. Here, a page’s lifetime is measured by the number of pages written between two consecutive writes to the page. The dataset is a write-intensive block trace from Alibaba Cloud [25] (trace #144; see Section V for detail). From the results we see that the lifetimes of pages classified as “hot” by SepBIT and MiDAS span across a very wide range. This is likely to lead to higher number of valid pages in the hot group during GC, resulting in higher WA (we experimentally verify this in Section V-D). In contrast, using ML model can more accurately identify hot pages with shorter lifetime, ensuring lower WA.

Prior works have proposed using ML models for data separation [13], [14]. However, they all follow a host-side design, suffering from weaknesses discussed earlier. Thus, our goal in this work is to build an in-storage, learning-based data separation scheme for maximum WA reduction.

III. Shiro Design

A. Shiro overview

Figure 2 shows the overall design of Shiro. Shiro uses supervised machine learning to accurately and adaptively predict the lifetime of pages written by the user to perform data separation. During GC, it further separates valid pages into multiple levels of superblocks with a reinforcement learning-assisted page migration strategy for lower long-term WA. As depicted in Figure 2, the main components of Shiro include:

1) The *Page Classifier* model. When processing a write command from the host, Shiro uses the Page Classifier to make a binary prediction for each logical page covered by the write command to determine whether it is short-living (overwritten in the near future). At the input side, the Page Classifier takes a time series of the page’s historical access statistics as features to learn from prolonged historical access patterns.

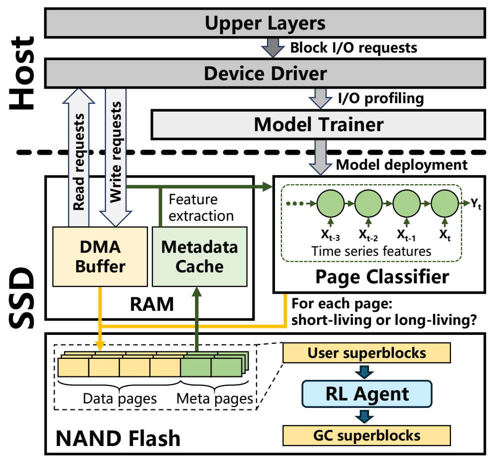


Fig. 2. Overview of Shiro's architecture.

2) The *Model Trainer*. Since there is no strict real-time requirement for model training, Shiro trains the Page Classifier model at the host to take advantage of host computation resources. Specifically, the Model Trainer profiles user I/Os from the device driver to collect training data used to train the Page Classifier. The Model Trainer deploys the trained Page Classifier by transferring model parameters to the SSD. As the system runs, the Model Trainer continues training data collection and periodically trains and deploys new model parameters to adapt to changing workload patterns.

4) The *RL Agent*. During GC, the reinforcement learning agent dynamically steers valid pages from the victim superblock to multiple destinations (GC superblocks; see below). We use a rich set of runtime page information as the environment space to characterize each page to be migrated so that the agent can perform meticulous decision making for every page. The agent is trained continuously with long-term write amplification as the reward (lower as better) to separate GC-written data and reduce long-term WA.

3) NAND flash management. Shiro manages NAND flash in the unit of superblocks. Data separation is performed by separating pages into dedicated superblocks. User-written pages are directed to *user superblocks* based on the prediction result of the Page Classifier. There are thus 3 types of user superblocks: short living, long living, or unseen (no model prediction result due to lack of input features). GC-written pages are separated into multiple levels of *GC superblocks* based on the RL Agent's decision. The number of levels of GC superblocks is configurable, and we use 5 in the current implementation. Within a superblock, user data are stored in data pages, while meta pages at the tail hold ML metadata.

4) Metadata cache and DMA buffer in RAM. In RAM, Shiro maintains a small on-demand cache of ML metadata for fast metadata retrieval. The RAM also serves as a temporary buffer for user data transferred from the host via DMA. The ML metadata cache takes up 23MB RAM per 1TB of flash storage assuming 16KB flash pages (Section III-D), and the DMA buffer has a fixed size of 2 superpages.

In the remainder of this section, we will describe in detail how the Page Classifier learns and predicts page lifetime for user-written data separation (Section III-B), the reinforcement learning-assisted page migration strategy for GC-written data separation (Section III-C), and techniques proposed to enable real-time learning-based decision making inside the resource-constrained device (Section III-D).

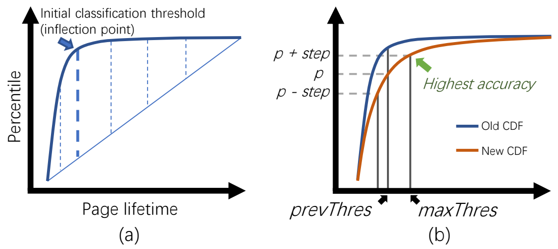


Fig. 3. Skewed distribution of page lifetime and threshold adjustment process.

B. Separating User-Written Data

Shiro uses supervised machine learning to predict the lifetime of data written by the user. To train the machine learning model, we need to label pages according to their lifetime and define a set of descriptive features to collect training data. In this section, we describe how Shiro adaptively labels pages as short-living and long-living, followed by the features and the structure of the model used to learn the labeled training data.

Adaptive data labeling. On the output side, the Page Classifier is designed to make a binary prediction for each logical page touched by a write command. The prediction result indicates whether the page's lifetime is lower than a threshold value, which is adaptively set at runtime (described below). In Shiro, we define the lifetime of a logical page as the number of logical pages written between two writes to this particular page. This is equivalent to using the global page write counter as a virtual clock. The effectiveness of this binary classification approach stems from the observation that page lifetimes in real-world workloads usually follow a skewed distribution [25], [26], as in Figure 3(a). This allows us to set aside a group of "short-living" pages whose lifetimes are significantly shorter than the rest to take advantage of data separation. Further, compared with multi-class classification and regression approaches, a binary classification model usually requires a smaller model capacity to achieve high accuracy and is thus more friendly to the resource-constrained SSD.

Although prior works proposed using multi-class classification [14] or regression [13] for fine-grained prediction, a larger model is required to achieve satisfactory accuracy. For example, the regression model used in ML-DT [13] takes over 100μs to complete a prediction on a server-grade CPU. In comparison, a binary classification by the Page Classifier only takes a few μs on the device controller (Section IV), making it more suitable for resource-constrained environments.

Shiro dynamically adjusts the classification threshold for training data labeling at runtime by sampling page lifetime. Specifically, we define a write *window* as 5% of the total size of the SSD written by the host. After each write window, the adjustment process is triggered. To collect training data for the Page Classifier, the Model Trainer profiles I/O requests at the device driver level. Any write request targeting a page that has been written before in the same window contributes a lifetime sample. Thus, at the end of a window, Model Trainer will have a set of sampled page lifetimes. Model Trainer then adjusts the classification threshold according to Algorithm 1.

Algorithm 1 The classification threshold adjustment algorithm

```
/* globals: initialized once */
step ← 5                                  /* threshold adjustment step length */
function PickThreshold($lifetimes, features, prevThres$)
    if $prevThres = -1$ then
        /* for the first window, just use the inflection point */
        return $GetInflectionPoint(lifetimes)$
    $p \leftarrow PercentileOfValue(lifetimes, prevThres)$
    $maxAccu, maxThres \leftarrow 0$
    for $dir$ in $[-1, 0, 1]$ do
        $t \leftarrow ValueAtPercentile(lifetimes, p + dir \times step)$
        /* label and resample to a small, balanced training set */
        $trainFeat, trainLabel \leftarrow LabelAndResample($
                $features, lifetimes, t)$
        /* train a lightweight model and evaluate accuracy */
        $accu \leftarrow TrainEvalLightModel(trainFeat, trainLabel)$
        if $accu > maxAccu$ then
            $maxAccu \leftarrow accu, maxThres \leftarrow t$
    /* adjust step */
    if no adjustment in previous and current window then
        $step++$                          /* avoid getting trapped in local optimal */
    else if adjusted in previous, no adjustment in current then
        $step--$                          /* try a finer adjustment step */
    else if different adjustment direction in previous and current then
        $step--$                          /* fluctuation, decrease step */
    else if same adjustment direction in previous and current then
        $step++$                          /* try to reach optimal faster */
    $step \leftarrow min(abs(step), 10)$
    return $maxThres$
```

The goal of the classification threshold adjustment algorithm is to dynamically adjust the threshold value toward the direction that can improve prediction accuracy. For the first window after system initialization, Shiro picks a threshold directly by first sorting the lifetime samples to acquire a set of (L_i, i) coordinates, where L_i is the ith sample in the sorted sample array (N samples in total). The corresponding sample of the coordinate that has the maximum distance (dashed lines in Figure 3(a)) from the line that connects $(L_1, 1)$ and (L_N, N) is selected as the initial threshold. Visually, this is the inflection point of the lifetime CDF curve (Figure 3(a)). The intuition behind this method is that the straight line represents a uniform distribution, and the point at which the distance between it and the real CDF starts to shrink can represent entrance to a “long tail”. For other windows, as in Figure 3(b), Shiro starts by locating the percentile position of the previous window's threshold value ($prevThres$) in the current window, say p. Shiro then attempts to adjust the classification threshold toward the direction that can improve prediction accuracy. This is done by testing three candidate thresholds at the $(p - step)$th, pth and $(p + step)$th percentile, where $step$ is the adjustment step in the current training round. Shiro labels the training data collected in the current window using the three candidate values to acquire three training sets. The sets are then used to train three lightweight logistic regression models. All training sets are resampled to have a balanced class distribution. The candidate value that produces a logistic regression model with the highest accuracy, $maxThres$, is picked as the new classification threshold. Finally, Shiro refines the adjustment process by lengthening or shortening the adjustment step:

- If no adjustment to the threshold value has been made in two consecutive windows, the algorithm increments the search step to avoid getting trapped in a local optimal.

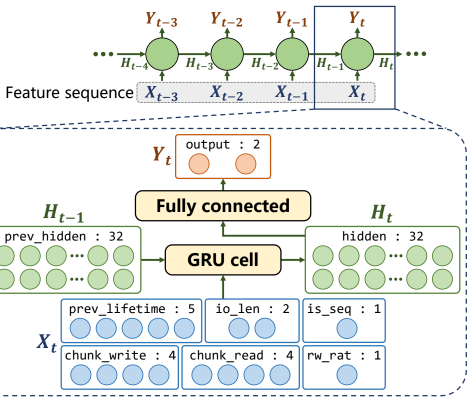


Fig. 4. Structure of the Page Classifier model.

- If the threshold value is adjusted in the previous window but not in the current one, it means we have possibly stepped over the optimal target. The algorithm thus decrements the step value.
- If the threshold value is adjusted in different directions in the previous window and the current one, lower the search step value to avoid hovering around a potential optimal value.
- If the same adjustment directions were applied to the threshold value, we increase the search step value as an attempt to let the algorithm reach the optimal value faster.

In Section V, we show that the Page Classifier can maintain high prediction accuracy throughout long user write sequences in real-world traces.

Feature extraction and model structure. After exploring a wide variety of machine learning models and input features, we finalized the Page Classifier to a lightweight sequence model using a time series of historical access statistics of the predicted page as features (Figure 4). During our design iterations, the most recently observed lifetime of a page (`prev_lifetime`) was found to be the most descriptive feature, capable of achieving ~70% accuracy. The accuracy can be improved by adding information about the current I/O request, including request size (`io_len`) and whether the request is sequential (`is_seq`). Here, a write request is deemed sequential if, together with the current request, the recent 32 requests cover 128KB or more consecutive data in sequential order. Locality and workload profile-related features are also helpful. Such features are captured by recording the number of recent read/write requests targeting the larger chunk to which the currently written page belongs (`chunk_write` and `chunk_read`) and the global read/write ratio (`rw_rat`). Finally, over 90% accuracy can be obtained by including all historical information using a time series of the aforementioned features.

As depicted in Figure 4, Page Classifier uses a gated recurrent unit (GRU) [27], a sequence model capable of processing time series data, to learn the labeled input features. We perform a grid search through different model structure and hyperparameter settings to land one that provides a balanced accuracy-computation tradeoff. In the finalized design, the GRU model uses a single-layer hidden state with 32 neurons. The hidden state of the last GRU cell is pushed through a fully connected layer to produce 2 output neurons. Finally, $argmax$

TABLE I
STATES USED IN REINFORCEMENT LEARNING-ASSISTED PAGE MIGRATION

Name	Number of states	Note
Current lifetime	25	Binning: [1, 2), [2, 4), [4, 8), [8, 16)......
Superblock valid page proportion	25	Binning: [0, 0.04), [0.04, 0.08), [0.08, 0.12)......
Superblock type	8	3 types of user superblocks and 5 levels of GC superblocks
Model prediction result	3	Short-living, long-living or unseen
Previous action	6	The page's previous GC destination with a special state indicating that the page has never been GC'ed

is applied to get the prediction result. For efficient processing, Shiro breaks numerical inputs into hexadecimal digits and each digit is used as an input neuron. The number of digits used for each feature is chosen so that most cases can be handled without overflow, while not wasting input budget by leaving large numbers of neurons as 0. The model is trained with the cross entropy loss function and the Adam optimizer using the training data collected in the window. After the first window, the model is trained until convergence. This can be viewed as bootstrapping the system using offline training sets. In later windows, the model is trained online for one epoch after each window using the training data collected in the window.

C. Separating GC-Written Data

To further separate GC-written data for lower runtime WA, Shiro uses reinforcement learning to steer valid pages to different destinations while taking feedback from long-term WA. In this section, we first briefly describe the basic GC policy used by Shiro, followed by the reinforcement learning-assisted page migration strategy.

GC triggering. Shiro performs garbage collection at superblock granularity. Both background and foreground garbage collection are performed, triggered by different free space watermarks. During system idle time, Shiro initiates a background garbage collection if the proportion of free superblocks is below 25%. When handling a write request, foreground garbage collection is triggered if the proportion of free superblocks is below 5%.

Victim selection. During garbage collection, Shiro uses the *Adjusted Greedy* policy given below to compute a score for each candidate superblock and picks the one with the highest score as victim:

$$score = \begin{cases} \frac{I}{1+V\frac{T}{C}} & \text{superblocks with short-living pages} \\ I & \text{all other superblocks} \end{cases} \qquad (1)$$

where I and V are the proportion of invalid and valid pages in the superblock, T is the classification threshold, and C is the elapsed time (number of pages written) since the superblock was closed.

The rationale of the Adjusted Greedy policy is similar to the Cost-Benefit policy [3] in that it gives lower priority to hot pages during GC. The more important purpose of it is to remedy wrong predictions. Specifically, the denominator in Equation 1 applies a discount to the final score of short-living (hot) pages. When there are more valid pages, the discount factor should be higher since the hidden cost of migrated pages soon turning invalid is greater, hence the term V. The term $\frac{T}{C}$ (T at the numerator for normalization) is added so that for two superblocks with the same number of invalid pages, the one that was closed earlier has a lower discount factor. This is because the model may make mistakes, and pages that are left valid for longer are more likely to have been mistakenly predicted as short-living. Such "false" short-living pages should be favored over "true" short-living pages during GC to remedy wrong predictions.

Reinforcement learning-assisted page migration. For further separation of GC writes, Shiro maintains multiple levels of GC superblocks for pages with different lifetime patterns, and each level corresponds to a possible destination of valid pages migrated during GC. As a result, each valid page to be migrated in a GC operation has multiple possible destinations. The goal when deciding the correct migration destination is to minimize write amplification in the long term. Although a rich set of runtime information can be collected to for each page (e.g., the page's current lifetime, the Page Classifier's prediction result), it is non-trivial to establish a causal relationship with future WA. To this end, Shiro uses reinforcement learning to steer valid data to the destination that is most likely to yield the lowest write amplification in the future. Reinforcement learning is well suited for this task since by training the agent continuously with a trial and error-like approach, the agent can dynamically map a given *environment* to an optimal *action* that maximizes future *reward* [28], [29]. In the specific task we have at hand, the page-specific information represents the environment, the migration destination corresponds to the action, and future reward can be set as lower write amplification.

Specifically, Shiro uses the Q-learning algorithm [30], which models external environment with discrete *states*, and tracks the expected reward of all actions under each combination of states using the Q-table. The key to effective RL-based decision making for page migration is thus the construction of an expressive state space to characterize each page and a reward function that can guide the agent toward the lower WA over the long term. We empirically select a rich set of runtime information to build the state space (Table I). The most important metric in the state space is the current lifetime of the page. This is used so that pages with different observed lifetimes are mapped to different states, allowing the RL agent to take different actions for pages falling into different lifetime ranges. The feature of the page's belonging superblock is also helpful because such information allows the RL agent to distinguish pages from superblocks with different runtime characteristics. Finally, we also add the Page Classifier's prediction result and the RL agent's previous action as potential hints for future actions.

For the reward function, we use the average proportion of invalid pages in the victim superblocks of the subsequent 200 GC operations. As a result, a higher reward corresponds to lower WA from GC, thus capturing the impact of an action

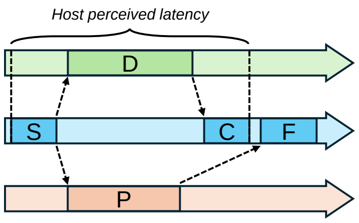


Fig. 5. Off-critical path prediction. S: command submission, D: DMA transmission, C: command completion, P: prediction, F: flash write back; arrow indicates dependency.

on long-term WA. Implementation-wise, we use the ϵ-greedy policy with a 1% exploration probability. Since the Q-table is small in size (439KB in total), we store the Q-table entirely in the on-device RAM for faster access. To kickstart the reinforcement agent, we initialize the Q-table to represent a simple hierarchical page migration strategy to avoid high WA when the agent has not been fully trained. Specifically, we initialize the reward values such that pages from user superblocks will be migrated to the GC superblocks in the lowest level, and that pages from GC superblocks will be migrated to a higher level. This is based on the heuristic that pages GC'ed for more times are more likely to be long-living, and grouping them to dedicated superblocks can avoid repeated data copies of (approximately) read-only data.

D. Enabling learning-based decision making inside the SSD

The majority of runtime overhead, both computation and storage, stems from the Page Classifier. To integrate the Page Classifier into the resource-constrained SSD, we propose a suite of enabling techniques. For computation overhead, Shiro removes prediction from the critical path and caches intermediate computation results (the hidden state) to reduce prediction complexity. For storage overhead, Shiro maintains ML metadata (the cached hidden state and feature extraction information) in flash and only keeps a small on-demand metadata cache in RAM.

Reducing computation cost. In NVMe, the de-facto standard for high-performance SSDs, the write command (64B) and data payload (usually in 4KB blocks) are transferred separately, and the latter is more time-consuming due to larger size. Specifically, the host initiates a write request by submitting an NVMe write command to the submission queue ("S" in Figure 5). When the SSD receives the command, it transfers the data payload from host memory to the device through the DMA engine ("D" in Figure 5). Afterwards, the device pushes an entry to the completion queue, marking the completion of the request ("C" in Figure 5). Writeback of the transferred payload to flash can be performed asynchronously ("F" in Figure 5).

This I/O process allows two optimizations that can mask prediction overhead from the host's perception: (1) *Parallelized prediction*. Shiro takes advantage of the multi-core architecture in modern SSD controllers by offloading Page Classifier to a dedicated core. This design allows prediction, command processing, and payload transfer to run in parallel, as all input features are available immediately upon receiving a write command. Consequently, the prediction process does

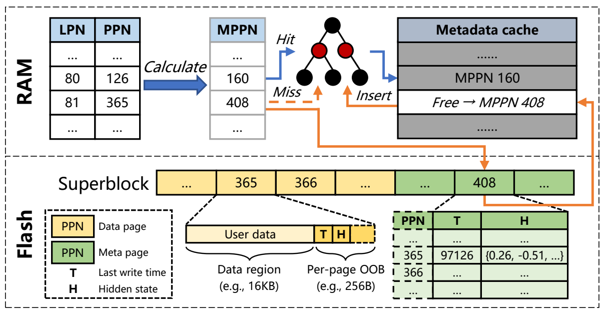


Fig. 6. Flash data layout and RAM metadata cache management.

not delay the handling of subsequent I/O commands and other internal FTL tasks. Moreover, the overhead of prediction for the current request is effectively masked by the payload transfer latency (denoted as "P" in Figure 5). This ensures that the latency experienced by the host remains low, maintaining high responsiveness. (2) *Decoupled command completion*. Since the result of prediction is only required when a page is about to be evicted to flash, Shiro decouples NVMe I/O command completion from model prediction. A write command can be marked as successfully completed once the payload reaches the DMA buffer in RAM, irrespective of whether the prediction process is finished. When the page is eventually flushed to flash, the prediction result is retrieved asynchronously. As demonstrated in Section V, this off-critical-path prediction technique effectively masks nearly all prediction overhead in our prototype implementation, ensuring minimal impact on system performance.

It is also necessary to address the high computation overhead of the sequence model due to long feature sequences. As shown in Figure 4, the model iteratively performs computation for the (very long) feature sequence of a page to draw a single prediction. Such computation overhead is unacceptable for real-time prediction. However, we observe that the feature sequences of a given page at two consecutive writes only differ in the last time step. Based on this observation, for every page, Shiro caches the hidden state of the last GRU cell (H_t) after each prediction. Upon prediction for a page, Shiro retrieves the cached hidden state. Together with the new input features as described in Section III-B, the model can yield the final result in a single step. The computation complexity of prediction is thus reduced from $O(N)$ to $O(1)$, where N is the length of the feature sequence. This allows us to take full advantage of the sequence model without costly iterative computation.

Reducing storage cost. In addition to the cached hidden state (32B for 8-bit quantized model) for each page, Shiro also needs to record per-page write timestamp (4B) for lifetime calculation. This leads to 36B of ML metadata for each page, which is usually unacceptable in commercial products. To reduce RAM consumption, Shiro stores these metadata in flash and maintains an on-demand metadata cache in RAM.

As shown in Figure 6, pages at the tail of a superblock are meta pages and store the ML metadata for each data page in the superblock. A meta page contains metadata entries for consecutive data pages in the superblock. Each time the model performs prediction for a logical page (LPN), its metadata are retrieved by first looking up the PPN of its corresponding data page in the L2P table. The address of the desired meta page (MPPN) can be calculated using the offset of the data page in the superblock. The MPPN is then used to search the metadata cache in RAM, which is indexed by a red-black tree. If the MPPN is found in the metadata cache, the requested metadata can be fetched directly from RAM, without incurring flash I/O. In the event of a cache miss, the meta page is read from flash and then inserted to the metadata cache. Since consecutive pages in the superblock are also allocated consecutively, the retrieved metadata entries have inherent locality. If the metadata cache is full, a slot is emptied following the LRU policy. The size of the metadata cache is set to 1% of the number of meta pages in the SSD. This cuts metadata RAM consumption to 0.36 bytes per page, which we believe is an acceptable cost. Besides the meta pages, each data page also keeps a copy of its metadata in the per-page OOB area so that during GC there is no need to read the meta pages for metadata migration.

IV. Shiro Implementation

We build a Shiro prototype on the Daisy+ OpenSSD (Daisy+ hereafter) [31], an SSD evaluation platform based on real hardware. Daisy+ is the latest iteration of the OpenSSD project, which has powered many studies on SSDs in recent years [7], [32]–[35]. Daisy+ is equipped with a Xilinx Zynq UltraScale+ SoC, which consists of a hard-wired quad-core ARM Cortex-A53 processor and a programmable FPGA. The ARM processor has access to 2GB LPDDR4 DRAM, serving as scratchpad memory for the firmware. Another 64GB (2$\times$ 32GB) of DDR4 DRAM in DIMM form factor is used as the storage backend, where host-written data reside. The host is connected to Daisy+ via NVMe over PCIe 3.0 $\times$8.

We first extend the firmware of Daisy+ to integrate NAND flash emulation since Daisy+ uses DRAM as the storage media. Besides storage media differences, the stock firmware that came with Daisy+ also lacks integral FTL functionalities. Out of the box, the firmware uses a direct mapping between the memory address space of the on-board DIMMs and the block address space exposed to the host. Host I/Os in the form of NVMe commands directly access data in the corresponding DIMM address ranges through PCIe DMA. There is therefore no “flash translation” involved. We thus port FEMU’s page-level FTL [36] for its flash emulation capability. The stock firmware only uses one ARM processor core to process NVMe commands. We bring up a dedicated core to run the FTL logic. The resulting architecture thus has a separated data plane and control plane: Host I/O data are directed to the DDR4 DIMMs, as in the stock firmware, but the completion of NVMe commands is now handled by the FTL, which simulates NAND flash behavior with configurable flash chip layout and read/program/erase latencies.

This architecture allows us to evaluate SSDs with different logical sizes, regardless of physical hardware limitation. Specifically, we match the emulated flash layout with the size of the device used by the traces (Table III). The logically emulated flash size need not be the same as the physical size of the DRAM backend. When conducting a trace replay, we cap addresses that fall out of the DRAM address space. The real LBA is embedded in a reserved field of the NVMe command. The FTL can then extract the real LBA from the command and correctly calculate the incurred flash operation latency. Compared to a pure emulator-based approach, our implementation can more accurately capture the performance implications of host-device communications since data transfers still happen via the hardware PCIe interface.

Upon the NAND-enabled Daisy+, we further add Shiro components. We use another dedicated core to run the Page Classifier. The GRU model is implemented in C from scratch. Since GRU mainly involves basic tensor computations such as matrix/vector multiplications, we take advantage of SIMD instructions (the ARM NEON extension [37]) to speed up model inference on the ARM processor. Note that modern ARM controllers that target high-performance storage applications (such as Cortex-R82 [38]) also pack SIMD extensions. To further boost inference performance, we quantize model parameters to 8-bit integers. The loss of accuracy from quantization is within 1%. After quantization, a NEON instruction can process up to 16 neurons at a time with its 128-bit vector registers [37], significantly boosting the model’s responsiveness. The inference process of GRU also involves heavy algebraic computations, most notably the sigmoid and tanh functions. We prepare a look-up table for each algebraic function at runtime so that computation can be finished with a simple table look up. The look-up tables are negligible in size since the quantized neurons are only 8-bit long. With the above optimizations, the overhead of a single prediction is brought down to 3.4μs, corresponding to a 4.48GB/s throughput given 16KB flash pages, regardless of the SSD’s capacity. For our evaluation platform, this is more than fast enough since the maximum host-to-device bandwidth is measured at around 3.7GB/s (without NAND flash emulation). For future faster SSDs (e.g., PCIe 5.0 SSDs), prediction can be accelerated by using techniques such as multi-thread inference or a specialized hardware accelerator. At the host side, we implement the Model Trainer using PyTorch. Across the host and the device, the implementation of Shiro totalled 8189 lines of C and Python code.

V. Evaluation

In this section, we present the experiment results of Shiro and baseline data separation schemes. Our evaluation seeks to answer the following questions:

- **Q1.** How well can Shiro reduce WA from GC compared with state-of-the-art in-storage data separation schemes?
- **Q2.** Can the Page Classifier accurately predict page lifetime and adapt to changing application patterns in real-world workloads?
- **Q3.** Can further separation of GC writes benefit WA reduction compared with only separating user writes?
- **Q4.** Is the runtime overhead of performing ML model training and prediction acceptable?
- **Q5.** Does the reduction in GC WA translate into tangible gains in I/O bandwidth and latency?

TABLE II
EVALUATION SYSTEM CONFIGURATIONS.

Platform	Item	Configuration
Host	CPU	2× Intel Xeon Gold 6240
	Memory	384GB DDR4
	OS	Ubuntu 22.04 LTS (Linux 6.8.0)
Device	CPU	4× Cortex-A53 @ 1.5GHz
	Memory	2GB LPDDR4
	Host interface	NVMe over PCIe 3.0 ×8
	Storage backend	2× 32GB DDR4 DIMM
	Flash latency	40μs read, 200μs program, 2ms erase

TABLE III
FLASH CHIP LAYOUT FOR CONFIGURED CAPACITIES.

Flash layout	Configured logical capacity			
	40GB	50GB	100GB	500GB
Pages per block	256	256	512	1024
Blocks per die	171	214	214	535
Dies per channel	8	8	8	8
Number of channels	8	8	8	8

A. Evaluation Setup

Dataset. We conduct our experiments using an open-source dataset of block traces from Alibaba Cloud [25]. This dataset includes block-level traces collected over a one-month period from 1,000 drives randomly sampled from a production cluster. To accurately assess WA caused by garbage collection, it is essential for the workload's total write size to be sufficiently large. Therefore, we select drives that went through more than 20 drive writes (i.e., 20 times the drive's capacity) during the trace period and use the first 20 drive writes in the traces for evaluation. A total of 20 drives out of the 1,000 met this criterion. Table IV shows the workload characteristics of the tested traces.

Methodology. To answer Q1, we implement MiDAS [6], SepBIT [5] and 2R [7] on our NAND-enabled Daisy+ and evaluate Shiro against them. We also evaluate the case where no data separation is performed (Base). For baselines that did not specify a victim selection policy, Cost-Benefit [3] is used. We run the traces on our evaluation platform and report the resulting WA. To answer Q2, we preprocess the traces to annotate the real lifetime of each page. This allows us to evaluate the accuracy of the Page Classifier at runtime. To answer Q3, we perform an ablation study of Shiro and baseline systems to investigate the contribution of user and GC data separation to the final performance results. To answer Q4 and Q5, we run microbenchmark as well as the Alibaba Cloud traces on Daisy+ running Shiro and baseline FTLs. I/O latency and bandwidth during the process are reported.

Testbed. Full configurations of our testbed are listed in Table II. As discussed in Section IV, our implementation on Daisy+ with separated data and control planes allows us to configure an emulated SSD capacity different from that physically available. Therefore, for each trace in the dataset, we set the logical capacity of the SSD to match that specified by the dataset. We couple each tested logical capacity with a flash layout that provides 7% over provisioning. Detailed flash chip layout for all evaluated logical capacities is given in Table III. The number of dies and channels are kept the same for all configurations to provide the same level of I/O concurrency. We use 16KB page for all configurations. The size of the DMA buffer is set to 2MB (2 superpages) to align with typical commercial SSDs [39].

TABLE IV
I/O CHARACTERISTICS AND TESTED TRACES.

Trace ID	Drive size (GB)	Write size (GB)	Read size (GB)	Avg. write size (KB)	Avg. read size (KB)
52	500	20378	97	72.74	32.97
58	500	33110	222	92.38	34.04
107	500	26329	123	81.24	29.80
141	500	27762	147	83.96	35.01
144	500	33436	397	90.90	59.04
178	500	23182	111	76.96	32.12
225	500	36210	252	47.67	34.58
177	100	8280	244	25.19	14.98
202	100	2104	0	276.09	4.00
316	100	4160	0	342.54	4.00
721	100	5832	1061	156.65	205.41
748	100	4674	108	303.05	58.46
38	50	13975	0	13.91	20.47
126	50	1332	0	63.58	15.72
132	50	1912	0	151.87	382.67
223	40	1562	18	131.01	23.77
228	40	842	0	8.45	12.02
277	40	12310	3	21.73	152.97
326	40	1551	4	160.56	26.00
679	40	9769	0	14.16	19.58

B. Write Amplification

Figure 7 illustrates the write amplification factor (WAF) of the tested traces under various data separation schemes. The WAF is computed as $(F - U)/U$, where F and U are the sizes of data written to flash and that written by the host, respectively. On average, Shiro reduces the overall WA by 67.6% compared with Base and 17.1%-58.0% compared with rule-based schemes. In Base, where no data separation is implemented, pages with different lifetimes are mixed within the same superblocks. During garbage collection, long-living pages contain valid data and must be relocated, resulting in high WA. 2R performs data separation by separating GC writes from user writes, based on the heuristic that valid pages during GC are likely to have longer lifetimes. SepBIT and MiDAS goes further by classifying user-written and GC-written pages into multiple levels according to their previous lifetimes [5] or the modelled lifetime distribution [6]. However, their rule-based heuristics offer limited accuracy, restricting their effectiveness in reducing WA. In contrast, Shiro offers better accuracy and adaptiveness using learning-based decision making, allowing it to achieve the lowest WA.

We therefore have the answer to **Q1**: On real-world traces, Shiro can significantly reduce WA compared with conventional FTLs and state-of-the-art in-storage data separation schemes.

C. Page Classifier Performance

Table V summarizes the inference performance metrics of Page Classifier across the tested traces. The model achieves an accuracy of 81.4%–98.7%, with an average of 90.9%. The F1 score (the harmonic average of precision and recall) ranges from 21.3% to 98.3%, averaging 86.7%. Trace #38 is the only case with an F1 score below 70%, but this trace inherently exhibits low WA, minimizing its impact. The model's strength lies in its ability to leverage prolonged historical patterns to predict page lifetime. When the feature sequence length is truncated to a single entry, prediction accuracy drops by as

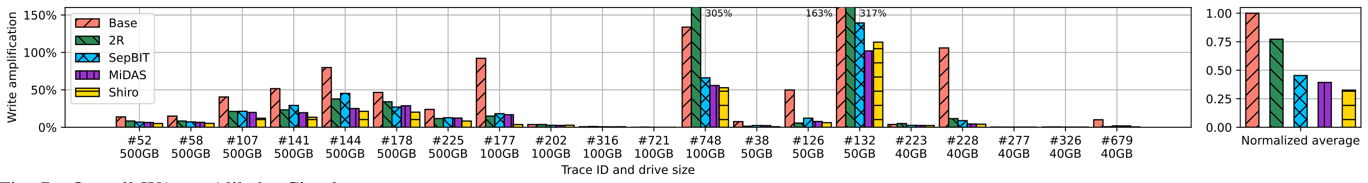


Fig. 7. Overall WA on Alibaba Cloud traces.

TABLE V
PAGE CLASSIFIER PERFORMANCE ON ALIBABA CLOUD TRACES

Trace ID	#52	#58	#107	#141	#144	#178	#225	#177	#202	#316	#721	#748	#38	#126	#132	#223	#228	#277	#326	#679	*Average*
Accuracy	0.894	0.826	0.852	0.898	0.844	0.879	0.814	0.972	0.969	0.957	0.937	0.832	0.874	0.863	0.907	0.951	0.979	0.971	0.987	0.968	*0.909*
Precision	0.933	0.837	0.869	0.937	0.872	0.909	0.823	0.824	0.980	0.984	0.772	0.897	0.213	0.792	0.931	0.967	0.892	0.969	0.672	0.606	*0.834*
Recall	0.938	0.932	0.944	0.940	0.925	0.942	0.934	0.944	0.988	0.972	0.897	0.907	0.664	0.675	0.969	0.979	0.972	0.987	0.965	0.947	*0.921*
F1	0.935	0.882	0.905	0.938	0.898	0.925	0.875	0.880	0.984	0.978	0.830	0.902	0.323	0.729	0.950	0.973	0.930	0.978	0.792	0.739	*0.867*

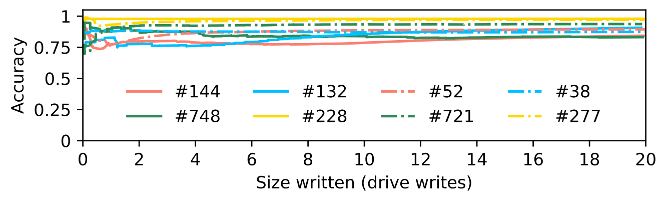


Fig. 8. Page Classifier prediction accuracy over time (cumulative).

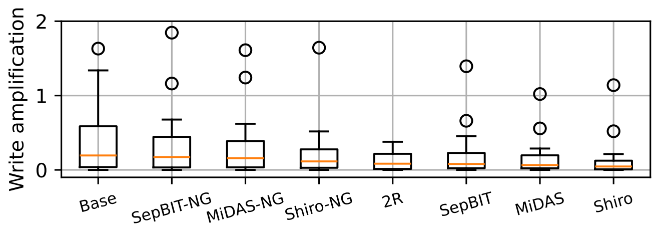


Fig. 9. WA on Alibaba Cloud traces with GC/user data separation disabled.

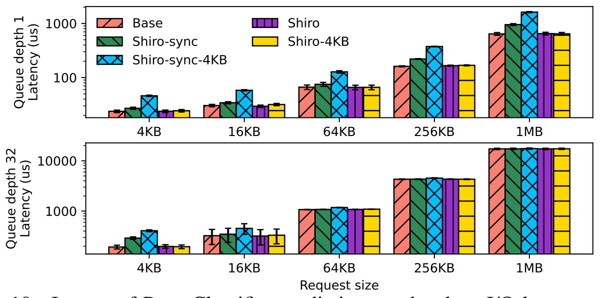


Fig. 10. Impact of Page Classifier prediction overhead on I/O latency.

much as 9.2% (4.0% on average), underscoring the importance of historical context. This predictive accuracy is a critical factor contributing to Shiro's effectiveness in separating data and reducing WA.

In Figure 8, we show the evolution of the Page Classifier's prediction accuracy over time for 8 of the tested traces (for each drive size, we show the trace with the highest/lowest WA). Although in some traces the Page Classifier starts off with relatively low prediction accuracy (73.1% for trace #721), online runtime training and adaptive data labeling allow the Page Classifier to continuously adapt to workload patterns. As a result, the Page Classifier can achieve high prediction accuracy as the drive enters steady state in most cases.

As such, we are able to answer **Q2**: <u>The Page Classifier can accurately predict the lifetime of user-written data in real-world workloads.</u>

D. *Effectiveness of Separating GC Writes*

In this experiment, we analyze the contribution to the reduction of WA from user data separation and GC data separation. Figure 9 shows the WA of the Alibaba Cloud traces (each box plot includes all 20 traces), and systems with the suffix "-NG" have their respective GC data separation removed. 2R does not have an "-NG" counterpart because it does not perform further separation for GC data. From the results we see that by separating user writes, SepBIT-NG, MiDAS-NG and Shiro-NG provide lower WA compared with Base, with Shiro-NG being the best performer. This shows that the Page Classifier can achieve better data separation accuracy compared with rule-based approaches. When GC writes are further separated in their full implementations, all evaluated systems exhibit further performance improvements, confirming the benefit of GC data separation on top of user data separation.

Therefore, we conclude for **Q3**: <u>Combining supervised learning for user data separation and reinforcement learning for GC data separation can deliver significant reduction in WA.</u>

E. *Machine Learning Overhead*

Training and prediction overhead. Runtime overhead from ML mainly comes from the Page Classifier, since reinforcement learning does not require much computation. In terms of training, we use the host server's CPU to train the Page Classifier at runtime. As discussed in Section III, the model is trained for one epoch in each write window, which is 5% of the drive size. Given a 500GB SSD with 16KB pages, this corresponds to a maximum of 1.7M samples. Each time step in a sample is 24B, and we cap the length of a sample series to 20. The maximum size of a training set is therefore aroud 800MB. In practice, the average size is 120MB for 500GB drives. The host also needs to track the lifetime of logical flash pages, which takes up 125MB per 500GB flash (for high-performance enterprise SSDs with 4KB pages, the memory overhead increases by 4×). To speed up training, we sample 10% of the collected data to train the model (loss of accuracy is negligible). On our testbed, training the model for one epoch takes less than one second, which is negligible compared to the duration of a write window in real-world workloads (minutes to hours).

To analyze the overhead of performing real-time prediction inside the SSD, we use fio to issue random write requests in different sizes and measure the latency of the requests. Only write is tested here because reads do not trigger prediction. We use a 500GB drive and direct the write requests to a

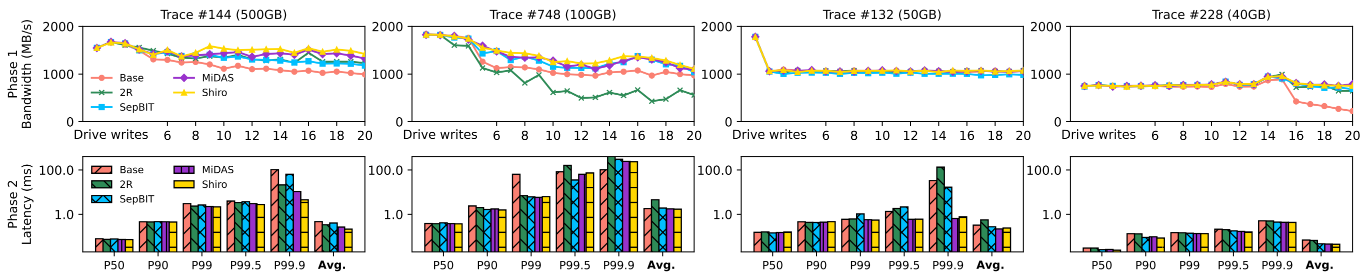


Fig. 11. I/O bandwidth and latency on high-WA traces.

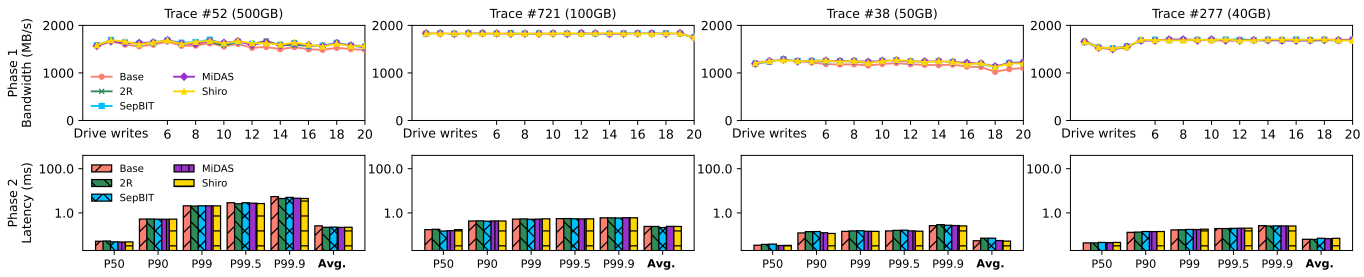


Fig. 12. I/O bandwidth and latency on low-WA traces.

fixed 10GB region prefilled with valid data so that all write requests can trigger prediction. To test the effectiveness of the proposed off-critical path prediction techniques, we test the scenario where prediction is placed on the critical path by using 1 core to handle both NVMe command processing and model inference (Shiro-sync). We also evaluate the case where 4KB logical pages are used by the FTL (suffixed with “4KB”).

Figure 10 presents the results (error bars indicate standard deviation; Y axis uses logarithmic scaling). In Shiro-sync, prediction affects I/O latency significantly under a single thread, with a 22.9% increase on average. When 4KB pages are used, the gap widens to 104%, since each write request triggers more predictions. The extra latency is proportional to the size of the request because the model performs prediction for all logical pages covered by the request. With 32 threads, Shiro-sync and Shiro-sync-4KB also exhibit higher latency when the size of the request cannot saturate device bandwidth. When we take prediction off the critical path (Shiro), average latency returns to normal and is almost the same as the stock FTL (Base), regardless of request size, number of I/O threads and the logical page size in use. This confirms the effectiveness of off-critical path prediction techniques. The standard deviation of I/O latency is slightly higher in Shiro than in Base because of occasional synchronization between the two cores and more cache line misses due to sharing.

Metadata overhead. As discussed in Section III, the RAM overhead of the metadata cache is 0.36B per page. If a page’s ML metadata is not found in the cache, a flash read operation is required to fetch it. When replaying the traces in Figure 7, only 0.1%–1.8% of metadata retrievals triggerred flash reads. This is due to the grouping and batch fetching of metadata in flash and, which benefits from temporal and spatial locality. In the worst case scenario (uniform random writes with a working set spanning the entire SSD), metadata fetching leads to a 19% decrease in write throughput. However, we did not encounter such workloads in any of the tested traces. Even if host I/Os follow the worst-case pattern, it is easy to avoid the performance drop by, for example, monitoring the hit rate of the metadata cache and disabling data separation when unusual I/O patterns are detected.

At this point, we can answer **Q4**: <u>ML model training and prediction do not pose significant runtime overhead.</u>

F. Impact on I/O Performance

To analyze the impact of WA reduction on I/O bandwidth and latency, we replay the Alibaba Cloud traces on Daisy+ with Shiro and baseline schemes. The results are presented in Figure 11 and Figure 12. Due to space limit, we show the results of the traces with the highest/lowest WA for each tested drive size. A trace with high WA allows us to analyze the performance gains from lower GC overhead, whereas a trace with low WA reveals the impact of ML overhead on performance. We break the trace replay into 2 phases. In phase 1, we ignore the timestamps of the I/O requests and stress load the trace data (except the last hour) into the SSD. This allows us to analyze the bandwidth of different data separation schemes under real-world I/O patterns. In phase 2, we follow the timestamps in the trace file and replay the last-hour trace data. This gives us meaningful I/O latency distributions of real-world applications.

As shown in Figure 11, when the trace exhibits high WA, Shiro can deliver tangible gains in I/O bandwidth and latency by lowering GC overhead. In phase 1, Shiro has slightly lower bandwidth than Base and rule-based data separation schemes in earlier drive writes. However, after the 5th drive write, WA reduction starts to take effect, and the bandwidth of Shiro surpasses that of the baseline systems. During the last drive write, Shiro can provide up to 43.2% and 20.5% higher bandwidth compared with the stock firmware and rule-based schemes. In phase 2, Shiro and the stock FTL have almost the same latency distribution (within 5% discrepancy) at low percentiles. At high percentiles, thanks to lower GC overhead, lower GC overhead significantly reduces tail latencies in Shiro, contributing to up to 51.2% reduction in average I/O latency.

When the trace does not have WA in the first place, data separation does not have a material impact on I/O performance

(Figure 12). Although some slight improvements are observed at high-percentile latency, the overall average latency and bandwidth latency remain similar. Nevertheless, this again shows that the added components in Shiro does not lead to visible runtime overhead.

Hence, the answer to **Q5** is clear: Accurate data separation can provide substantial improvement in steady-state I/O bandwidth and latency by reducing WA from GC.

VI. Related Work

Optimizing GC. Shiro uses ML to perform data separation inside the SSD to reduce WA from GC. With fewer valid pages, GC itself is accelerated. Other data separation schemes, including rule-based [5], [7]–[9] and learning-based approaches [13], [14], have been discussed in earlier sections. Researchers have also taken on other fronts to tackle GC and WA, and works under such topics are complementary to Shiro. First, SSDs in production environment often adopt data redundancy for reliability. There are therefore opportunities to ”unblock” host I/Os during GC, such as using erasure codes internally [40] or in an SSD array [21]. Furthermore, prioritizing host I/Os over GC is effective in improving I/O performance. Kang *et al.* [41] proposed breaking a full GC to partial GCs and using RL to determine the optimal number of partial GCs under heavy workloads. PEN [42] further introduces partial erase at the hardware level. Finally, data deduplication [43]–[46] can also improve GC since less data written to flash is equivalent to a higher OP ratio.

Prolonging device lifespan. NAND flash is known to suffer from degraded performance as more data are written to it, eventually leading to device wearout [1]. Shiro attacks this problem by reducing WA from GC writes to delay wearout due to excessive writes. Other studies that target the lifespan issue of SSDs focus on hardware-level causes of wearout, including mitigating read disturbance [47], [48], data retention handling [49]–[51] and advanced wear levelling strategy [19], [52], [53]. Since GC is an inevitable operation for NAND flash due to its log-structured nature, Shiro can work in tandem such solutions to provide optimal device lifespan.

Exploiting in-storage computing resources. Shiro uses the SSD's computing resources for data separation to reduce WA. As SSD controller becomes more powerful, recent studies have also proposed running user application logic directly in storage. Such a paradigm is referred to as computational storage (CS) [54]. Prior works on CS system include domain-specific architectures such as information retrieval [32], neural network training [34], [55] and data analysis in HTAP [56], and generic CS frameworks offering programming models based on block [57], [58] or file interfaces [35], [59], [60]. CS improves the performance of offloaded tasks by shortening the I/O path, which is particularly effective in I/O-intensive applications. In comparison, our evaluation confirms that Shiro can provide higher I/O performance by reducing GC overhead, thus also benefiting host applications.

VII. Conclusion

In this paper, we introduce Shiro, a novel approach that employs machine learning techniques for data lifetime separation inside the storage device, targeting both user writes and GC writes. To implement the proposed scheme efficiently, we present a set of optimization techniques that minimize computation and storage overhead, making it feasible for SSD integration. Experimental comparisons with state-of-the-art data separation methods demonstrate that Shiro achieves significantly lower write amplification. Additionally, trace I/O evaluations confirm that Shiro is a practical solution, offering substantial improvements in application I/O performance.

Acknowledgment

This work is supported by National Key Research and Development Program of China (Grant No. 2023YFB4502902), National Natural Science Foundation of China (NSFC) (Grant No. 62332012, 62227809, 62302290), the Fundamental Research Funds for the Central Universities, and Alibaba Group through the Alibaba Innovative Research (AIR) program.

References

[1] A. Tai, A. Kryczka, S. O. Kanaujia, K. Jamieson, M. J. Freedman, and A. Cidon, “Who's afraid of uncorrectable bit errors? online recovery of flash errors with distributed redundancy,” in 2019 USENIX Annual Technical Conference (USENIX ATC 19), 2019, pp. 977–992.

[2] S. Maneas, K. Mahdaviani, T. Emami, and B. Schroeder, “Operational characteristics of ssds in enterprise storage systems: A large-scale field study,” in 20th USENIX Conference on File and Storage Technologies (FAST 22), 2022, pp. 165–180.

[3] M. Rosenblum and J. K. Ousterhout, “The design and implementation of a log-structured file system,” ACM Transactions on Computer Systems (TOCS), vol. 10, no. 1, pp. 26–52, 1992.

[4] J. He, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “The unwritten contract of solid state drives,” in Proceedings of the twelfth European conference on computer systems, 2017, pp. 127–144.

[5] Q. Wang, J. Li, P. P. Lee, T. Ouyang, C. Shi, and L. Huang, “Separating data via block invalidation time inference for write amplification reduction in log-structured storage,” in 20th USENIX Conference on File and Storage Technologies (FAST 22), 2022, pp. 429–444.

[6] S. Oh, J. Kim, S. Han, J. Kim, S. Lee, and S. H. Noh, “{MIDAS}: Minimizing write amplification in {Log-Structured} systems through adaptive group number and size configuration,” in 22nd USENIX Conference on File and Storage Technologies (FAST 24), 2024, pp. 259–275.

[7] M. Kang, S. Choi, G. Oh, and S.-W. Lee, “2r: Efficiently isolating cold pages in flash storages,” Proceedings of the VLDB Endowment, vol. 13, no. 12, pp. 2004–2017, 2020.

[8] E. Rho, K. Joshi, S.-U. Shin, N. J. Shetty, J. Hwang, S. Cho, D. D. Lee, and J. Jeong, “Fstream: Managing flash streams in the file system,” in 16th USENIX Conference on File and Storage Technologies (FAST 18), 2018, pp. 257–264.

[9] T. Kim, D. Hong, S. S. Hahn, M. Chun, S. Lee, J. Hwang, J. Lee, and J. Kim, “Fully automatic stream management for multi-streamedssds using program contexts,” in 17th USENIX Conference on File and Storage Technologies (FAST 19), 2019, pp. 295–308.

[10] M. Maas, D. G. Andersen, M. Isard, M. M. Javanmard, K. S. McKinley, and C. Raffel, “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, 2020, pp. 541–556.

[11] C. Li, M. Wu, Y. Liu, K. Zhou, J. Zhang, and Y. Sun, “Ss-lru: a smart segmented lru caching,” in Proceedings of the 59th ACM/IEEE Design Automation Conference, 2022, pp. 397–402.

[12] J. Zhang, X. Li, X. Zhou, M. Yuan, Z. Cheng, K. Huang, and Y. Li, “L-qoco: learning to optimize cache capacity overloading in storage systems,” in Proceedings of the 59th ACM/IEEE Design Automation Conference, 2022, pp. 379–384.

[13] C. Chakraborttii and H. Litz, “Reducing write amplification in flash by death-time prediction of logical block addresses,” in Proceedings of the 14th ACM International Conference on Systems and Storage, 2021, pp. 1–12.

[14] P. Yang, N. Xue, Y. Zhang, Y. Zhou, L. Sun, W. Chen, Z. Chen, W. Xia, J. Li, and K. Kwon, "Reducing garbage collection overhead in ssd based on workload prediction," in 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19), 2019.

[15] J. Zhang, M. Kwon, D. Gouk, S. Koh, C. Lee, M. Alian, M. Chun, M. T. Kandemir, N. S. Kim, J. Kim et al., "Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds," in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), 2018, pp. 477–492.

[16] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal, "Rearchitecting linux storage stack for μs latency and high throughput," in 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), 2021, pp. 113–128.

[17] "Marvell ssd controllers," https://www.marvell.com/products/ssd-controllers.html.

[18] P. Sun, L. You, S. Zheng, W. Zhang, R. Ma, J. Yang, G. Wang, F. Zhu, S. Li, and L. Huang, "Learning-based data separation for write amplification reduction in solid state drives," in 2023 60th ACM/IEEE Design Automation Conference (DAC). IEEE, 2023, pp. 1–6.

[19] S. Wang, F. Wu, C. Yang, J. Zhou, C. Xie, and J. Wan, "Was: Wear aware superblock management for prolonging ssd lifetime," in Proceedings of the 56th Annual Design Automation Conference 2019, 2019, pp. 1–6.

[20] M. Hao, L. Toksoz, N. Li, E. E. Halim, H. Hoffmann, and H. S. Gunawi, "Linnos: Predictability on unpredictable flash storage with a light neural network," in 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), 2020, pp. 173–190.

[21] H. Li, M. L. Putra, R. Shi, X. Lin, G. R. Ganger, and H. S. Gunawi, "Ioda: A host/device co-design for strong predictability contract on modern flash storage," in Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, 2021, pp. 263–279.

[22] J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan, "Autostream: Automatic stream management for multi-streamed ssds," in Proceedings of the 10th ACM International Systems and Storage Conference, 2017, pp. 1–11.

[23] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14), 2014.

[24] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. Le Moal, G. R. Ganger, and G. Amvrosiadis, "Zns: Avoiding the block interface tax for flash-based ssds," in 2021 USENIX Annual Technical Conference (USENIX ATC 21), 2021, pp. 689–703.

[25] J. Li, Q. Wang, P. P. Lee, and C. Shi, "An in-depth analysis of cloud block storage workloads in large-scale production," in 2020 IEEE International Symposium on Workload Characterization (IISWC). IEEE, 2020, pp. 37–47.

[26] G. Yadgar, M. Gabel, S. Jaffer, and B. Schroeder, "Ssd-based workload characteristics and their performance implications," ACM Transactions on Storage (TOS), vol. 17, no. 1, pp. 1–26, 2021.

[27] K. Cho, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," arXiv preprint arXiv:1406.1078, 2014.

[28] W. Kang and S. Yoo, "Dynamic management of key states for reinforcement learning-assisted garbage collection to reduce long tail latency in ssd," in Proceedings of the 55th Annual Design Automation Conference, 2018, pp. 1–6.

[29] Q. Wei, Y. Li, Z. Jia, M. Zhao, Z. Shen, and B. Li, "Reinforcement learning-assisted management for convertible ssds," in 2023 60th ACM/IEEE Design Automation Conference (DAC). IEEE, 2023, pp. 1–6.

[30] R. S. Sutton and A. G. Barto, Reinforcement learning: An introduction. MIT press, 2018.

[31] "Daisy+ openssd," https://www.crz-tech.com/crz/article/DaisyPlus/.

[32] S. Liang, Y. Wang, Y. Lu, Z. Yang, H. Li, and X. Li, "Cognitive ssd: A deep learning engine for in-storage data retrieval," in 2019 USENIX Annual Technical Conference (USENIX ATC 19), 2019, pp. 395–410.

[33] M. Wilkening, U. Gupta, S. Hsia, C. Trippel, C.-J. Wu, D. Brooks, and G.-Y. Wei, "Recssd: near data processing for solid state drive based recommendation inference," in Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021, pp. 717–729.

[34] Y. Lee, J. Chung, and M. Rhu, "Smartsage: training large-scale graph neural networks using in-storage processing architectures," in Proceedings of the 49th Annual International Symposium on Computer Architecture, 2022, pp. 932–945.

[35] Z. Yang, Y. Lu, X. Liao, Y. Chen, J. Li, S. He, and J. Shu, "λ-io: A unified io stack for computational storage," in 21st USENIX Conference on File and Storage Technologies (FAST 23), 2023, pp. 347–362.

[36] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The case of femu: Cheap, accurate, scalable and extensible flash emulator," in 16th USENIX Conference on File and Storage Technologies (FAST 18), 2018, pp. 83–90.

[37] "Neon," https://developer.arm.com/Architectures/Neon.

[38] "Cortex-r82," https://developer.arm.com/Processors/Cortex-R82.

[39] S.-H. Kim, J. Shim, E. Lee, S. Jeong, I. Kang, and J.-S. Kim, "Nvmevirt: A versatile software-defined virtual nvme device," in 21st USENIX Conference on File and Storage Technologies (FAST 23), 2023, pp. 379–394.

[40] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds," ACM Transactions on Storage (TOS), vol. 13, no. 3, pp. 1–26, 2017.

[41] W. Kang, D. Shin, and S. Yoo, "Reinforcement learning-assisted garbage collection to mitigate long-tail latency in ssd," ACM Transactions on Embedded Computing Systems (TECS), vol. 16, no. 5s, pp. 1–20, 2017.

[42] C.-Y. Liu, J. Kotra, M. Jung, and M. Kandemir, "Pen: Design and evaluation of partial-erase for 3d nand-based high density ssds," in 16th USENIX Conference on File and Storage Technologies (FAST 18), 2018, pp. 67–82.

[43] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing nand flash-based ssds," in 9th USENIX Conference on File and Storage Technologies (FAST 11), 2011.

[44] F. Chen, T. Luo, and X. Zhang, "Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in 9th USENIX Conference on File and Storage Technologies (FAST 11), 2011.

[45] F. Ni, X. Wu, W. Li, L. Wang, and S. Jiang, "Leveraging ssd's flexible address mapping to accelerate data copy operations," in 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). IEEE, 2019, pp. 1051–1059.

[46] Y. Zhou, Q. Wu, F. Wu, H. Jiang, J. Zhou, and C. Xie, "Remap-ssd: Safely and efficiently exploiting ssd address remapping to eliminate duplicate writes," in 19th USENIX Conference on File and Storage Technologies (FAST 21), 2021, pp. 187–202.

[47] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Error patterns in mlc nand flash memory: Measurement, characterization, and analysis," in 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2012, pp. 521–526.

[48] Y. Cai, Y. Luo, S. Ghose, and O. Mutlu, "Read disturb errors in mlc nand flash memory: Characterization, mitigation, and recovery," in 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE, 2015, pp. 438–449.

[49] Y. Luo, Y. Cai, S. Ghose, J. Choi, and O. Mutlu, "Warm: Improving nand flash memory lifetime with write-hotness aware retention management," in 2015 31st Symposium on Mass Storage Systems and Technologies (MSST). IEEE, 2015, pp. 1–14.

[50] M.-C. Yang, C.-F. Wu, S.-H. Chen, Y.-L. Lin, C.-W. Chang, and Y.-H. Chang, "On minimizing internal data migrations of flash devices via lifetime-retention harmonization," IEEE Transactions on Computers, vol. 70, no. 3, pp. 428–439, 2020.

[51] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu, "Improving 3d nand flash memory lifetime by tolerating early retention loss and process variation," Proceedings of the ACM on Measurement and Analysis of Computing Systems, vol. 2, no. 3, pp. 1–48, 2018.

[52] J. Li, X. Xu, X. Peng, and J. Liao, "Pattern-based write scheduling and read balance-oriented wear-leveling for solid state drivers," in 2019 35th Symposium on Mass Storage Systems and Technologies (MSST). IEEE, 2019, pp. 126–133.

[53] Z. Jiao, J. Bhimani, and B. S. Kim, "Wear leveling in ssds considered harmful," in Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems, 2022, pp. 72–78.

[54] J. Do, S. Sengupta, and S. Swanson, "Programmable solid-state storage in future cloud datacenters," Communications of the ACM, vol. 62, no. 6, pp. 54–62, 2019.

[55] S. Kim, Y. Jin, G. Sohn, J. Bae, T. J. Ham, and J. W. Lee, "Behemoth: a flash-centric training accelerator for extreme-scale dnns," in 19th USENIX Conference on File and Storage Technologies (FAST 21), 2021, pp. 371–385.

[56] K. Lee, I. Jo, J. Ahn, H. Lee, H. Lee, W. Sul, and H. Jung, "Deploying computational storage for htap dbmss takes more than just computation offloading," Proceedings of the VLDB Endowment, vol. 16, no. 6, pp. 1480–1493, 2023.

[57] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A user-programmablessd," in 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), 2014, pp. 67–80.
[58] G. Koo, K. K. Matam, T. I, H. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: trading communication with computing near storage," in Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, 2017, pp. 219–231.
[59] Z. Ruan, T. He, and J. Cong, "Insider: Designing in-storage computing system for emerging high-performance drive," in 2019 USENIX Annual Technical Conference (USENIX ATC 19), 2019, pp. 379–394.
[60] R. Schmid, M. Plauth, L. Wenzel, F. Eberhardt, and A. Polze, "Accessible near-storage computing with fpgas," in Proceedings of the Fifteenth European Conference on Computer Systems, 2020, pp. 1–12.

Penghao Sun received his M.S. degree from Shanghai Jiao Tong University in 2022, and is now a Ph.D. candidate at Shanghai Jiao Tong University. His research interests include memory and storage systems.

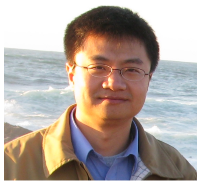
Shengan Zheng received the B.S. and Ph.D. degrees from Shanghai Jiao Tong University in 2014 and 2019, respectively. He is currently an assistant professor at Shanghai Jiao Tong University. His research interests include memory systems, storage systems and distributed systems.

Litong You received the B.Eng. degree in computer science from Sichuan University, China, in 2017, and the Ph.D. degree in computer science from Shanghai Jiao Tong University. He is currently an Associate Research Professor with the School of Computer Science and Technology, Hangzhou Dianzi University, China. His research interests include storage systems, embedded system, and Industrial Internet of Things.

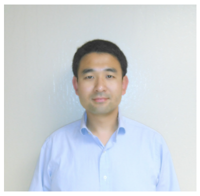
Wanru Zhang received her M.S. from Shanghai Jiao Tong University in 2024, where she studied and researched SSD-based storage systems.

Ruoyan Ma is currently pursuing his master's degree at Shanghai Jiao Tong University. His research interests include machine learning system and near data processing.

Jie Yang got his master's degree from the College of Control Science and Engineering at Zhejiang University in 2006, and his bachelor's degree from the Department of Mechanical Engineering at Zhejiang University in 2004. He has worked at Broadcom and Micron, and is currently working at Alibaba Cloud.



Feng Zhu got his Ph.D. degree from the Department of Electrical and Computer Engineering at University of Texas at Austin in 2008, his master's degree from the Department of Electrical Engineering at University of Notre Dame in 2003, and his bachelor's degree from the Department of Electrical Engineering at Tsinghua University in 2001. He is currently working at Alibaba Cloud.



Shu Li got his Ph.D. degree from the Department of Electrical Engineering in Rensselaer Polytechnic Institute in 2009, and his master's and bachelor's degree from the Department of Electrical Engineering at Tsinghua University int 2005 and 2003, respectively. He has worked at Marvell Semiconductor and Broadcom (formerly LSI), and is currently working at Alibaba Cloud.

Linpeng Huang (Senior Member, IEEE) received his M.S. and Ph.D. degrees in computer science from Shanghai Jiao Tong University in 1989 and 1992, respectively. He is a professor of computer science in the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research interests include distributed systems and service oriented computing.