Learning-based Data Separation for Write Amplification Reduction in Solid State Drives

Penghao Sun[1], Litong You[1], Shengan Zheng[2] ✉, Wanru Zhang[1], Ruoyan Ma[1],
Jie Yang[3], Guanzhong Wang[3], Feng Zhu[3], Shu Li[3], Linpeng Huang[1] ✉
[1]*Shanghai Jiao Tong University* [2]*MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University* [3]*Alibaba Group*
✉{shengan, lphuang}@sjtu.edu.cn

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247795

***Abstract*—Garbage collection in SSDs causes write amplification. The key to mitigating this problem is separating data by lifetime. Prior works proposed using machine learning to accurately predict data lifetime but prediction is performed at the host side, burdening the host storage stack. We present PHFTL, a practical, holistic FTL design with device-side learning-based data separation. The machine learning model in PHFTL accurately and adaptively predicts the lifetime of every written page. A suite of enabling techniques are introduced to keep computation and storage overhead low. Extensive evaluation of PHFTL demonstrates superiority over state-of-the-art and feasibility on real hardware.**

***Index Terms*—solid state drive, flash translation layer, write amplification, data separation, machine learning**

I. Introduction

NAND flash-based solid state drives (SSDs) are widely deployed in data centers to back secondary storage. To squeeze more capacity out of compact form factors, current NAND flash technology is heading towards higher density with lower endurance: From SLC flash to QLC flash, storage density increased by 4 times, but the number of program/erase cycles (P/E cycles) the media can sustain dropped from tens of thousands to only a few thousands [1]. On the other hand, write amplification (WA) is inevitable in SSDs due to the nature of flash storage, which consumes extra P/E cycles and accelerates device wear out. With a recent study reporting that WA at the field can soar up to over 100 (i.e., flash write size is over 100× the actual user write size) [2], controlling WA remains an urgent research topic.

Garbage collection (GC) is among the major sources that contribute to WA in SSDs. Due to the "erase before write" restriction, a flash translation layer (FTL) is placed between the host and bare flash to turn all writes into appends. When the FTL runs out of free pages (the smaller read/write units) to accommodate new writes, GC is triggered and one or more blocks (the larger erase units) are selected as victims. The victim blocks are erased to release free pages after all pages with valid data have been migrated, leading to WA.

The key to reducing WA from GC is separating data by lifetime, i.e., data separation [3], [4]. When pages with similar lifetimes are grouped into dedicated blocks, it is more likely that pages in the same block are invalidated within temporal proximity. The FTL thus has a greater chance of securing a victim block with a low valid page count during GC, thereby lowering WA.

To perform data separation, we need to predict when in the future a page written by the host will be overwritten. Most prior works in this respect are rule-based approaches built on simple heuristics [4]–[7] and fall short of adaptability and accuracy. Recent successes of the application of machine learning (ML) in system software [8]–[10] promise more adaptive and accurate data separation using ML models [11], [12]. However, existing solutions put both model training and prediction at the host. Whereas training can be performed offline, prediction is tightly coupled with application I/Os and is thus latency-critical. Consequently, integrating computation-intensive ML models into the host I/O path for prediction may lead to interference with other storage stack tasks [13]. It also adds to the already burdensome "storage tax", where up to 20% CPU cycles are spent on the storage stack in data centers [14].

A device-side approach that performs prediction inside the SSD confines overhead to the device itself and avoids burdening the host storage stack. However, this poses significant challenges. The effectiveness of data separation is maximized when the lifetime of every page can be accurately predicted. But such fine-grained prediction puts rigorous demand on the responsiveness of the model, especially considering that the write latency of modern SSDs can be as low as a few microseconds [15]. Although SSD controllers have evolved to adopt multi-core architectures [16], they still cannot compete with host-side CPUs or GPUs. Furthermore, metadata bookkeeping for ML should not incur overly high consumption of the on-device RAM, a scarce resource due to its high cost compared to flash.

We present *Prediction-based High-performance FTL* (PHFTL), a practical, holistic FTL design that employs ML for real-time, fine-grained data separation inside the SSD. The ML model in PHFTL is a lightweight sequence model. It extracts information from long historical access patterns of a page using a time series of lifetime and request-related features. For every written page, the model is able to accurately predict whether it is short-living or long-living. The classification criterion is dynamically adjusted at runtime via an adaptive labeling scheme. To address computation overhead, PHFTL masks ML computation by taking prediction off the critical path. This is achieved by interleaving prediction with other internal tasks and decoupling command completion from prediction. Additionally, PHFTL caches intermediate computation results (the hidden state) for each page to take full advantage of a sequence model without costly recursive computation. For storage overhead, the RAM consumption that arises from ML metadata, including feature extraction information and the cached hidden state, is resolved with an efficient flash data layout. The layout design allows PHFTL to keep all ML metadata in flash and easily fetch them to RAM in batches to build an on-demand cache for fast retrieval. Together, the proposed techniques enable real-time prediction inside the SSD while keeping computation and storage overhead low. To the best of our knowledge, PHFTL is the first FTL design that successfully incorporates complex ML algorithm for data separation at the device side.

We compare PHFTL with state-of-the-art data separation schemes using real-world traces to demonstrate PHFTL's superiority in reducing WA. To showcase its feasibility, we prototype PHFTL on Cosmos+ OpenSSD [17], an SSD evaluation platform based on real

Penghao Sun and Litong You contributed equally to this work. This work is supported by the National Key Research and Development Program of China (No. 2022YFB4500303), the National Natural Science Foundation of China (No. 62227809), the Fundamental Research Funds for the Central Universities, the Shanghai Municipal Science and Technology Major Project (No. 2021SHZDZX0102), Natural Science Foundation of Shanghai (No. 22ZR1435400). This work is also supported by Alibaba Group through the Alibaba Innovative Research (AIR) program.

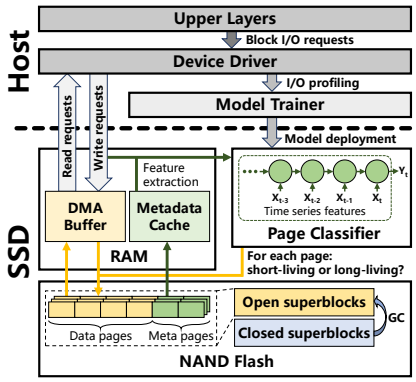

Fig. 1. PHFTL architecture

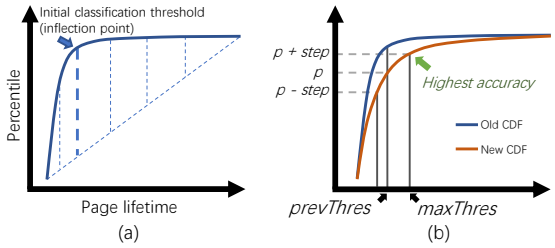

Fig. 2. Skewed distribution of page lifetime and threshold adjustment process

hardware. Microbenchmark shows that ML incurs almost no runtime overhead. Furthermore, trace evaluation confirms that lower WA transforms into substantial gains in steady-state I/O performance.

In summary, this paper makes the following contributions:

(1) We present a holistic FTL design that incorporates device-side learning-based data separation using an ML model that can accurately and adaptively identify short-living and long-living pages in real time.

(2) We address ML computation overhead by removing prediction from the critical path and caching intermediate computation results.

(3) We address ML storage overhead with an efficient flash data layout that allows all ML metadata to be kept in flash and easily retrieved to RAM in batches for caching to exploit locality.

(4) We prototype the proposed design and evaluate it against state-of-the-art schemes to demonstrate its effectiveness in reducing WA and its feasibility on real hardware.

II. Background and Related Work

A. SSD Preliminaries

Flash-based SSDs are built on arrays of NAND flash dies connected to an embedded controller via multiple channels. Each die can be accessed independently and is composed of a series of blocks, which again contain multiple pages. Reads and writes are performed at page granularity. Pages within a block must be written sequentially, while written pages can be read in any order. When all pages in a block have been written, the block must be erased as a whole before the pages in it can be written again. To hide this idiosyncrasy from the host and preserve the standard block device interface, the controller runs a flash translation layer (FTL) to turn all writes into appends. The FTL tracks the mapping from logical page numbers (LPNs) to physical page numbers (PPNs) using the L2P table, usually maintained at page granularity in the on-device RAM for maximum performance. In modern SSDs, another common technique is to use *superblocks* (formed by all blocks with the same die offset) as the basic management unit [18]. The FTL allocates new pages from an *open* superblock in a round-robin fashion across dies to exploit inter-die parallelism. Full open superblocks are *closed* and read-only, awaiting GC.

B. Tackling GC Write Amplification

When the FTL runs out of free pages, GC is triggered to recycle obsolete pages. During GC, pages in the victim (super)blocks that contain valid data must be copied elsewhere, which incurs WA. Higher WA from GC leads to faster device wear out and interference with host I/Os. Tackling this problem has attracted much attention from researchers. The key to reducing WA from GC is data separation, i.e., separating user-written data with different lifetimes [3]. Assuming the lifetime of a page written by the host is known, there exists an optimal data placement scheme that can ensure 0 WA by organizing pages in their invalidation order [4]. However, this is impractical due to lack of future knowledge. Researchers thus proposed methods to estimate the lifetime of user-written data for data separation [4]–[7], [11], [12]. Most prior works are rule-based approaches based on simple heuristics [4]–[7], such as grouping user-written pages according to their update frequency or write syscall context. They have negligible runtime overhead, but can only offer limited adaptiveness and accuracy. Learning-based data separation uses ML models to extract patterns from a wide variety of features for higher lifetime prediction accuracy [11], [12]. However, existing solutions perform both training and prediction at the host, and the latter burdens the host storage stack due to its latency-critical nature. In this work, our goal is to confine ML overhead to the device itself by enabling real-time lifetime prediction inside the SSD.

III. Design

A. PHFTL overview

Figure 1 shows the overall design of PHFTL. PHFTL uses supervised machine learning to accurately and adaptively predict the lifetime of pages written by the user to perform data separation. As depicted in Figure 1, the main components of PHFTL include:

1) The *Page Classifier* model. When processing a write command from the host, PHFTL uses the Page Classifier to make a binary prediction for each logical page touched 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.

2) The *Model Trainer*. Since there is no strict real-time requirement for model training, PHFTL trains the 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.

3) NAND flash management. PHFTL manages NAND flash in the unit of superblocks. Data separation is performed by separating pages into dedicated superblocks. User-written pages are directed to different superblocks based on the prediction result of the Page Classifier (i.e., whether they are short-living or long-living). GC-written pages are separated based on the number of times they have been selected as GC victims. For example, pages that have been GC'ed once and pages that have been GC'ed twice go to different superblocks, etc. (Pages GC'ed five times or more go to the same superblock.) This has the benefit that read-only pages will eventually be grouped into dedicated superblocks. 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, PHFTL maintains a small on-demand cache of ML metadata for fast metadata

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 ← PercentileOfValue(lifetimes, prevThres)
    maxAccu, maxThres ← 0
    for dir in [−1, 0, 1] do
        t ← ValueAtPercentile(lifetimes, p + dir × step)
        /* label and resample to a small, balanced training set */
        trainFeat, trainLabel ← LabelAndResample(
                features, lifetimes, t)
        /* train a lightweight model and evaluate accuracy */
        accu ← TrainEvalLightModel(trainFeat, trainLabel)
        if accu > maxAccu then
            maxAccu ← accu, maxThres ← 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 ← min(abs(step), 10)
    return maxThres
```

retrieval. The RAM also serves as a temporary buffer for user data transferred from the host via DMA.

In the remainder of this section, we will describe in detail how the Page Classifier learns and predicts page lifetime (Section III-B), techniques proposed to enable real-time prediction at the device side (Section III-C), and the GC policy (Section III-D).

B. Learning-based Data Separation

Since PHFTL uses supervised machine learning to predict page lifetime, the first question that arises is how to label pages according to their lifetime. In this section, we describe how PHFTL 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 PHFTL, the lifetime of a logical page is defined 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 [19], [20], as in Figure 2(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 requires a smaller model capacity to achieve high accuracy and is thus more friendly to the resource-constrained SSD.

PHFTL dynamically adjusts the classification threshold for training data labeling at runtime by sampling page lifetime. The adjustment process is triggered after each write *window*, during which the host writes 5% of the total size of the SSD. In each window, any write request targeting a page that has been written before in the same window contributes a lifetime sample. At the end of a window,

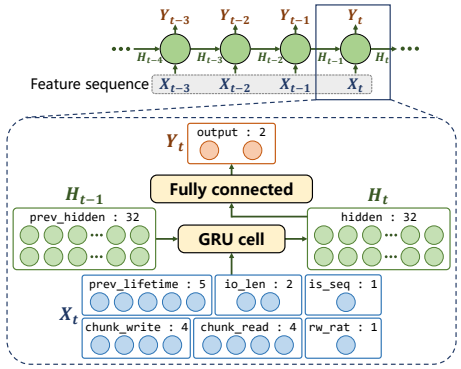

Fig. 3. Structure of the Page Classifier model

PHFTL will have a set of sampled page lifetimes. PHFTL then adjusts the classification threshold according to Algorithm 1.

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, PHFTL 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 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 2(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 2(b), PHFTL starts by locating the percentile position of the previous window's threshold value ($prevThres$) in the current window, say p. PHFTL 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. PHFTL labels the training data collected in the current window using the three candidate values to train three lightweight logistic regression models. All training sets are resampled to have a balanced class distribution. The candidate value that delivers the highest accuracy, $maxThres$, is picked as the new classification threshold. Finally, PHFTL refines the adjustment process by lengthening or shortening the adjustment step.

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 3). During our design iterations, the most recently observed lifetime of a page (`prev_lifetime`) was found to be the most useful feature, capable of achieving ∼70% accuracy. The accuracy can be improved by adding information about the current write request, including request size (`io_len`) and whether the request is sequential (`is_seq`). Locality and workload profile-related features are also helpful, and 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 3, Page Classifier uses a gated recurrent unit (GRU), a sequence model capable of processing time series data, to learn the labeled input features. The GRU model has a single-layer

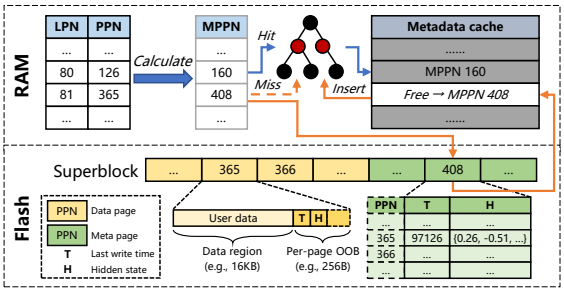

Fig. 4. Flash data layout and RAM metadata cache management

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, $argmax$ is applied to get the prediction result. For efficient processing, PHFTL breaks numerical inputs into hexadecimal digits and each digit represents a neuron. The number of digits used for each feature is chosen so that most cases can be handled without overflow. The model is trained for one epoch at the end of each window with the cross entropy loss function and the Adam optimizer using the training data collected in the window.

C. Enabling Prediction inside the SSD

To integrate the Page Classifier into the resource-constrained SSD, we propose a suite of enabling techniques. For computation overhead, PHFTL removes prediction from the critical path and caches intermediate computation results (the hidden state) to reduce prediction complexity. For storage overhead, PHFTL 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. This allows two optimizations that can mask prediction overhead from the host's perception: (1) *Interleaved prediction*. PHFTL leverages the multi-core architecture of modern SSD controllers and offloads Page Classifier to a dedicated core. Since all input features are available as soon as a write command is received, prediction, command processing and payload transfer can proceed in parallel. This ensures that prediction does not block other internal tasks. More importantly, prediction overhead for the current request is hidden by the payload transfer latency. (2) *Decoupled command completion*. The prediction result is only needed when a page is to be written to flash. We thus decouple command completion and model prediction by allowing a write command to return successfully once the payload has reached the DMA buffer in RAM, regardless of whether prediction has finished. When the page is to be flushed to flash, the prediction result is collected asynchronously. In Section V, we show that off-critical path prediction techniques can mask nearly all the prediction overhead.

Further optimization opportunity lies in the computation complexity of the sequence model itself. As shown in Figure 3, the model recursively 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, 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, PHFTL caches the hidden state of the last GRU cell (H_t) after each prediction. Upon prediction for a page, PHFTL 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.

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

As shown in Figure 4, 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. 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 intrinsic locality. If the metadata cache is full, a slot is emptied first 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. 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.

D. Garbage Collection

In PHFTL, GC is performed in the unit of superblocks. After each write request, if the proportion of free superblocks is lower than 5%, a victim superblock will be selected for GC. During GC, the superblock with the highest score computed using the *Adjusted Greedy* policy is selected as the victim:

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

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 [21] 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.

IV. Implementation

We prototype PHFTL with two implementations on different evaluation platforms: PHFTL-emu and PHFTL-hw. PHFTL-emu is implemented on FEMU [22], a QEMU-based SSD emulator. The flexibility of FEMU allows us to align SSD configurations (page size,

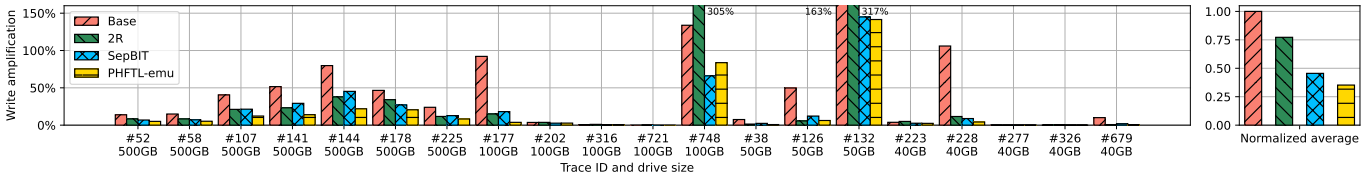

Fig. 5. Overall write amplification on Alibaba Cloud traces

TABLE I
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*

number of channels, etc.) to different traces with ease. However, since FEMU emulates the FTL with a dedicated thread on the host CPU, PHFTL-emu cannot capture the ML computation overhead on a weak SSD controller. We therefore only use PHFTL-emu to evaluate the capability of PHFTL in reducing WA.

PHFTL-hw is implemented on Cosmos+ OpenSSD (the OpenSSD) [17], an SSD evaluation platform based on real hardware. The OpenSSD features a Xilinx Zynq 7000 SoC, 1GB DRAM and ∼500GB NAND flash. It communicates with the host using standard NVMe. The SoC consists of a hard-wired dual-core ARM Cortex-A9 processor and a programmable FPGA. The ARM processor serves as the controller and runs the FTL, where we add PHFTL components. We use one core to run the Page Classifier and the other to handle all other tasks. We accelerate ML computation using SIMD instructions [23], which are also available in the newest high-end storage controllers [24]. All model parameters are quantized to 8-bit integers at a loss of accuracy in less than 1%. After careful tuning and optimization, the overhead of a single prediction is brought down to around 9μs, which is enough to enable real-time prediction. Nevertheless, even faster prediction can be achieved by using a more powerful embedded processor or a specialized hardware accelerator.

The development of PHFTL-emu and PHFTL-hw took ∼1500 LoC and ∼6000 LoC, respectively.

V. EVALUATION

This section presents evaluation results of PHFTL and baseline systems. When conducting the evaluation, we seek to answer the following research questions:

- **RQ1.** How does PHFTL perform in terms of write amplification reduction compared with state-of-the-art schemes?
- **RQ2.** Can the Page Classifier accurately predict page lifetime in real-world workloads?
- **RQ3.** Is the overhead of performing machine learning computation at runtime acceptable?
- **RQ4.** How well does the reduction in GC write amplification translate into tangible gains in latency and bandwidth?

A. Evaluation Setup

Dataset. We use the block trace dataset from Alibaba Cloud [19] to perform the experiments. The dataset contains 1-month block traces of 1000 drives randomly sampled from a production cluster. Because testing WA from GC requires the write size to be sufficiently large, we select all drives that sustained more than 20 drive writes (i.e., 20× the drive size) during the trace period and use the first 20 drive writes in the traces for evaluation. 20 drives out of 1000 meet this requirement, and their sizes range from 40GB to 500GB.

Methodology. We evaluate the two prototype implementations of PHFTL (Section IV) for different purposes. To answer RQ1, we implement SepBIT [4] and 2R [5], two state-of-the-art device-side data separation schemes, on FEMU and evaluate PHFTL-emu against them. We also evaluate FEMU's original FTL (Base), which performs no data separation. For baselines that did not specify a victim selection policy, Cost-Benefit [21] is used. We run the traces on FEMU and report the resulting WA. To answer RQ2, 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 RQ3 and RQ4, we run microbenchmark as well as the Alibaba Cloud traces on the OpenSSD running PHFTL-hw and the stock FTL. I/O latency and bandwidth during the process are reported.

Configurations. We use a server with two Intel Xeon Gold 6240M CPUs and 1.5TB memory (Intel Optane Persistent Memory in memory mode; DRAM size is 192GB) as the host system. The page size of all emulated SSDs and the OpenSSD is 16KB. Over-provision rate is set to 7%. 32 worker threads are launched for trace replay, which is enough to saturate device bandwidth.

B. Write Amplification

Figure 5 shows the WA of the tested traces under different data separation schemes. WA is calculated as $(F-U)/U$, where F and U are flash write size and user write size, respectively. On average, PHFTL reduces the overall WA by 65.1% compared with Base and 22.8%-54.6% compared with rule-based schemes. In Base, pages with different lifetimes are mixed in the same block. During GC, long-living pages remain valid and must be copied elsewhere, leading to high WA. 2R only separates GC writes from user writes based on the heuristic that valid pages during GC are long-living. SepBIT classifies user-written pages according to their inferred lifetime, but simply assumes that the lifetime of a newly written page is equal to its previous lifetime. Such simple heuristics fail to deliver high separation accuracy and only have limited effectiveness in reducing WA. By contrast, PHFTL improves accuracy and adaptiveness using ML, allowing it to achieve the lowest overall WA. When running the traces, we also notice that the small metadata cache in RAM can serve 98.2%-99.9% ML metadata retrievals. This is because ML metadata are fetched from flash in batches with intrinsic temporal and spatial locality, allowing one meta page read to serve many subsequent metadata retrievals.

We therefore have the answer to **RQ1**: *On real-world traces, PHFTL can achieve significantly lower WA compared with conventional FTLs and state-of-the-art data separation schemes.*

C. Page Classifier Performance

Table I presents the performance metrics of Page Classifier on the tested traces. As shown in Table I, the model can deliver 81.4%-

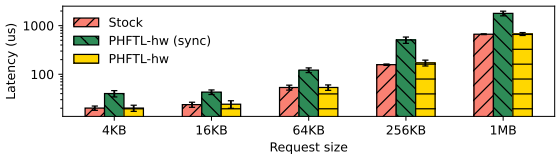

Fig. 6. Effect of off-critical path prediction

98.7% accuracy (90.9% on average). F1 score (the harmonic average of precision and recall) ranged from 21.3% to 98.3% (86.7% on average), with trace #38 being the only one below 70%, which does not exhibit high WA in the first place. We find that the model's ability to learn page lifetime from prolonged historical patterns is of great help: When we truncate the length of the feature sequence to 1, prediction accuracy drops by up to 9.2% (4.0% on average). The ability of the Page Classifier to accurately predict page lifetime is a key factor in the effectiveness of data separation in PHFTL.

As such, we are able to answer **RQ2**: *The Page Classifier can accurately identify short-living and long-living pages in real-world workloads.*

D. Feasibility and Performance on Cosmos+ OpenSSD

Overhead analysis. Training the model for one epoch takes less than one second, which is negligible compared to the duration of a window in real-world workloads (minutes to hours), and CPU usage is at most 15% on our testbed. To analyze prediction overhead, we use fio to issue write requests of different sizes. Figure 6 presents the average latency (error bars are standard deviation). Only write is tested because reads do not trigger prediction. Write offset is capped to 16MB, the size of the RAM data buffer on the OpenSSD, so that no data are written to flash. This setting allows us to stress the FTL to the extreme for otherwise flash latency will be the bottleneck.

In PHFTL-hw (sync), prediction is placed on the critical path by using only one core to perform prediction and other internal tasks. In this case, latencies increase dramatically, by 139.7% on average. When we take prediction off the critical path (PHFTL-hw), average latency returns to normal and is almost the same as the stock FTL. This confirms the effectiveness of off-critical path prediction techniques. Latency standard deviation is higher in PHFTL-hw than in stock because of occasional synchronization between the two cores and more cache line misses due to sharing.

At this point, we can answer **RQ3**: *Almost no visible overhead is incurred by ML at runtime.*

Real-world traces. We further evaluate the performance of PHFTL-hw on the Alibaba Cloud traces (Figure 7). The OpenSSD has ∼500GB flash. We therefore test the most representative traces backed by 500GB drives. Trace #144 with the highest WA and trace #52 with the lowest WA are picked since 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 do not test all the baselines and traces due to device endurance concerns. We break the trace replay into 2 phases. In phase 1, we stress load the trace data (except the last hour) and report bandwidth. In phase 2, we follow the timestamps in the trace and replay the last-hour trace data to measure latency.

As shown in Figure 7, PHFTL-hw has slightly lower bandwidth during the first 3 drive writes in phase 1. After the 4th drive write, WA reduction starts to take effect (even for #52 with low WA), and the bandwidth of PHFTL-hw surpasses that of the stock FTL. As more data are written and the drive enters steady state, the gap widens. During the last drive write, PHFTL-hw delivers 12.1% and 61.6% higher bandwidth on #52 and #144, respectively. In phase

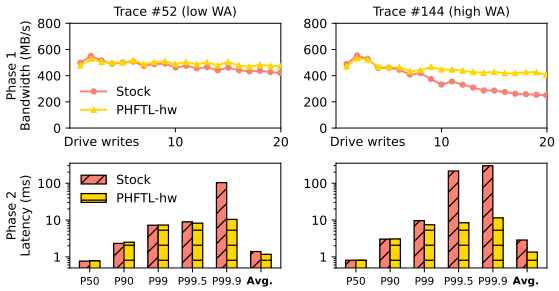

Fig. 7. Impact of WA reduction on bandwidth and latency

2, PHFTL-hw and the stock FTL have almost the same latency distribution (within 5% discrepancy) at low percentiles on both #52 and #144. However, thanks to lower GC overhead, tail latencies are significantly lower in PHFTL-hw, which contributes to a 16.2% and 53.0% reduction in average latency on #52 and #144, respectively.

Hence, the answer to **RQ4** is clear: *The reduction in GC write amplification contributes to substantial improvement in steady-state bandwidth and latency, despite extra machine learning overhead.*

VI. Conclusion

In this paper, we present PHFTL. PHFTL successfully incorporates ML for real-time and fine-grained data separation inside the SSD. The ML model in PHFTL can accurately and adaptively identify short-living and long-living pages. A set of enabling techniques are proposed to integrate the model into the SSD while keeping computation and storage overhead low. Evaluation against state-of-the-art data separation schemes shows that PHFTL can achieve significantly lower WA. Further evaluation on real hardware demonstrates that PHFTL is a practical design and can improve steady-state I/O performance.

References

[1] A. Tai *et al.*, "Who's afraid of uncorrectable bit errors? online recovery of flash errors with distributed redundancy," in *ATC*, 2019.
[2] S. Maneas *et al.*, "Operational characteristics of ssds in enterprise storage systems: A large-scale field study," in *FAST*, 2022.
[3] J. He *et al.*, "The unwritten contract of solid state drives," in *EuroSys*, 2017.
[4] Q. Wang *et al.*, "Separating data via block invalidation time inference for write amplification reduction in log-structured storage," in *FAST*, 2022.
[5] M. Kang *et al.*, "2r: Efficiently isolating cold pages in flash storages," in *VLDB*, 2020.
[6] E. Rho *et al.*, "Fstream: Managing flash streams in the file system," in *FAST*, 2018.
[7] T. Kim *et al.*, "Fully automatic stream management for multi-streamed ssds using program contexts," in *FAST*, 2019.
[8] M. Maas *et al.*, "Learning-based memory allocation for c++ server workloads," in *ASPLOS*, 2020.
[9] C. Li *et al.*, "Ss-lru: a smart segmented lru caching," in *DAC*, 2022.
[10] J. Zhang *et al.*, "L-qoco: learning to optimize cache capacity overloading in storage systems," in *DAC*, 2022.
[11] C. Chakraborttii *et al.*, "Reducing write amplification in flash by death-time prediction of logical block addresses," in *SYSTOR*, 2021.
[12] P. Yang *et al.*, "Reducing garbage collection overhead in ssd based on workload prediction," in *HotStorage*, 2019.
[13] J. Hwang *et al.*, "Rearchitecting linux storage stack for μs latency and high throughput," in *OSDI*, 2021.
[14] H. Li *et al.*, "Leapio: Efficient and portable virtual nvme storage on arm socs," in *ASPLOS*, 2020.
[15] J. Zhang *et al.*, "Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds," in *OSDI*, 2018.
[16] "Marvell ssd controllers," https://www.marvell.com/products/ssd-controllers.html.
[17] J. Kwak *et al.*, "Cosmos+ openssd: Rapid prototype for flash storage systems," in *TOS*, 2020.
[18] S. Wang *et al.*, "Was: Wear aware superblock management for prolonging ssd lifetime," in *DAC*, 2019.
[19] J. Li *et al.*, "An in-depth analysis of cloud block storage workloads in large-scale production," in *IISWC*, 2020.
[20] G. Yadgar *et al.*, "Ssd-based workload characteristics and their performance implications," in *TOS*, 2021.
[21] M. Rosenblum *et al.*, "The design and implementation of a log-structured file system," in *TOCS*, 1992.
[22] H. Li *et al.*, "The case of femu: Cheap, accurate, scalable and extensible flash emulator," in *FAST*, 2018.
[23] "Neon," https://developer.arm.com/Architectures/Neon.
[24] "Cortex-r82," https://developer.arm.com/Processors/Cortex-R82.