Reducing Write Amplification in Flash by Death-time Prediction of Logical Block Addresses

Chandranil Chakraborttii
University of California, Santa Cruz
Santa Cruz, CA, USA
cchakrab@ucsc.edu

Heiner Litz
University of California, Santa Cruz
Santa Cruz, CA, USA
hlitz@ucsc.edu

ABSTRACT

Flash-based solid state drives lack support for in-place updates, and hence deploy a flash translation layer to absorb the writes. For this purpose, SSDs implement a log-structured storage system introducing garbage collection and write-amplification overheads. In this paper, we present a machine learning based approach for reducing write amplification in log structured file systems via death-time prediction of logical block addresses. We define *death-time* of a data element as the number of I/O writes before which the data element is overwritten. We leverage the sequential nature of I/O accesses to train lightweight, yet powerful, temporal convolutional network (TCN) based models to predict death-times of logical blocks in SSDs. We leverage the predicted death-times in designing *ML-DT*, a near-optimal data placement technique that minimizes write amplification (WA) in log structured storage systems. We compare our approach with three state-of-the-art data placement schemes and show that *ML-DT* achieves the lowest WA by utilizing the learnt I/O death-time patterns from real-world storage workloads. Our proposed approach results in up to 14% reduction in write amplification compared to the best baseline technique. Additionally, we present a mapping learning technique to test the applicability of our approach to new or unseen workloads and present a hyper-parameter sensitive study.

ACM Reference Format:
Chandranil Chakraborttii and Heiner Litz. 2021. Reducing Write Amplification in Flash by Death-time Prediction of Logical Block Addresses. In *The 14th ACM International Systems and Storage Conference (SYSTOR '21), June 14–16, 2021, Haifa, Israel.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3456727.3463784

SYSTOR '21, June 14–16, 2021, Haifa, Israel

ACM ISBN 978-1-4503-8398-1/21/06.
https://doi.org/10.1145/3456727.3463784

1 INTRODUCTION

Modern flash-based solid-state drives (SSDs) present as a high-performance and cost-effective storage solution, providing terabytes of capacity, over a million I/O operations per second (IOPS), and sub 100μs read latency [24, 31]. However, SSDs suffer from limited endurance due to wear out. In particular, the existing 3D NAND and quad-level-cell-based SSDs support between 5K-50K write/erase cycles [43], which if exceeded, may result in data loss. Thus, it is imperative to minimize the number of writes applied to a flash cell.

Unfortunately, SSDs suffer from the problem of write-amplification due to lack of support for in-place updates. Instead of overwriting the data directly in-place, SSDs need to first perform an erase operation, before another program operation (erase-then-write) can occur. Furthermore, erase operations are performed at the granularity of blocks, whereas a block can hold multiple 4K pages (the unit of writes). As a result, SSDs support updates by implementing a log-structured storage mechanism [35, 39], where overwritten pages are appended to an open block. A logical-to-physical (L2P) translation table maps logical block addresses (LBA) to physical locations in the flash chips. When an LBA is overwritten, the L2P is updated so that the LBA points to the new physical location of the page, invalidating the old physical location of the LBA. When an SSD exhausts its blocks, garbage collection (GC) cleans up the blocks by moving valid pages to other free blocks, inducing write amplification in the process. Write amplification is problematic for two reasons. First, by introducing additional writes, the lifetime of the SSDs is reduced. Second, the extra GC writes introduce performance interference by delaying the regular user reads.

The magnitude of write amplification (WA) in SSDs depends on two factors, particularly, the mapping mechanism deployed by the flash translation layer (FTL) and the write patterns. For applications that exhibit sequential write patterns with uniform write frequencies across LBAs, WA tends to be low, as there is a high probability that LBAs within the same block are overwritten with temporal proximity. However, for most applications, the write frequency of LBAs follow a highly skewed distribution, as shown in Figure 1 for a virtual desktop trace (VDI) application. As a result, prior

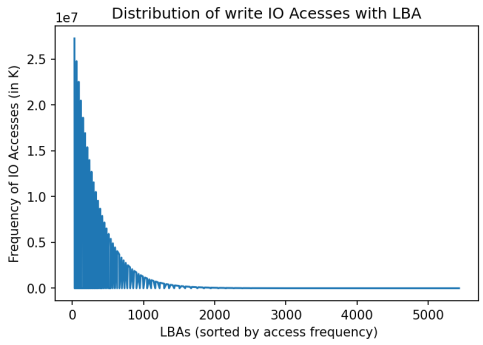

Figure 1: LBA write frequency distribution for VDI

work [12, 23, 26, 30, 34, 38, 50, 51] focused on reducing WA by optimizing the data placement policy within FTL. For instance, by separating the frequently written LBAs (hot) from rarely written LBAs (cold) and placing them into different blocks, write amplification can be reduced. The key idea behind these techniques is that placing the LBAs with similar write frequencies on the same block increases the likelihood that all LBAs within that block will be overwritten by the user-writes with temporal proximity.

We observe that while temperature-based techniques can reduce write amplification, they cannot eliminate it. We propose *Oracle-DT*, a novel mechanism that utilizes the *death-times* of LBAs to eliminate write amplification in log-structured storage, such as in SSDs.Hereby, the death-time of an LBA is defined as the number of I/O writes before said LBA is overwritten. By allocating the LBAs with consecutive death-times into the same block, write amplification can be eliminated. Proposed *Oracle-DT* requires perfect future knowledge about death-times and a potentially large number of concurrently opened blocks. As these requirements are impractical, we also propose *ML-DT*, a mechanism leveraging machine learning to predict the future death-times of LBAs. We evaluate *ML-DT* using VDI, TPC-H, and RocksDB application traces and show that it can reduce write amplification by up to 14% over prior work.

To sum up, this paper makes the following contributions:

- *Oracle-DT*, a data placement strategy that eliminates write amplification
- *ML-DT*, a practical approach leveraging death-time information to minimize write amplification
- Evaluation of machine learning techniques to predict LBA death-times
- Experimental evaluations showing up to 14% improvement over prior work
- Exploration of mapping learning techniques to generalize machine learning models across applications

2 BACKGROUND

In this section, we first introduce the write amplification problem in log structured storage systems and then discuss prior techniques for reducing write amplification.

2.1 Write Amplification Problem

In log-structured storage systems such as an SSD, data is not updated in-place, but instead, appended to a log. The log maintains multiple versions of the same data. To bound the storage capacity of log-structured storage systems, GC needs to be performed to remove the overwritten data elements from the log. Furthermore, a level of indirection (mapping table) is required to map logical data elements to their most recent physical location in the log. The most recent version of a logical data element is considered as *valid*, whereas all other versions are considered as *invalid*. In SSDs, the log is constructed of blocks which hold multiple pages referring to the unit size of a write. As a result, garbage collecting a block requires all valid pages to be moved to a new block, and only then the cleaned block can be erased. Moving pages during this process induces write amplification. There exist two common techniques to choose a victim block for GC. Greedy mechanisms [11, 14] choose the block with the lowest number of valid blocks. Cost-benefit mechanisms [14, 15] consider the future writes that may invalidate additional pages before choosing a victim block. In a fresh SSD, all blocks start out as *free-blocks*. Written pages are appended to an *open-block* and when the block is fully written, it is regarded a *closed-block*. As pages are overwritten, closed-blocks contain an increasing number of invalid pages. When the GC mechanism has cleaned a block, it is added back to the list of free-blocks.

2.2 Hot-Cold Separation

Temperature-based techniques such as hot-cold separation, have been proposed to alleviate the write amplification problem. These techniques maintain multiple logs (open blocks in an SSD) and map LBAs to blocks based on their update frequency. These approaches distinguish between the *user-writes* issued by the application and the *GC-writes* issued internally by the GC mechanism for cleaning blocks. These mechanisms group frequently written pages into the same block to reduce the average number of valid blocks within a hot block, thus reducing GC induced WA. This technique can be extended by maintaining more than two open blocks representing the temperature of its contained pages. For instance, when a hot page is garbage collected, it is first demoted to a warm block and if a page is garbage collected from a warm block, it is further demoted to a cold block. Temperature-based methods do not incur any metadata storage overheads besides tagging each block based on its temperature.

2.3 Frequency-Based Approaches

Frequency-based approaches [26, 34, 43, 50] differ from hot-cold separation, as they measure the update frequency of LBAs directly, instead of inferring the temperature from the block in which the LBA is currently residing in. These approaches map each LBA to a specific stream and then assign an open block to each stream. Frequency-based approaches can react faster to workload changes compared to temperature-based techniques, however, they induce extra overheads for learning the update frequency of a given LBA. Multi-stream SSDs [20] have been deployed to leverage this technique.

3 DEATH-TIME TECHNIQUE

We propose a novel placement mechanism for log-structured stores that minimizes write amplification. While in this section, we focus on SSDs, our technique can be applied to other log-structured stores as well. Our approach leverages death-time of an LBA, defined as the number of I/O writes before said LBA is overwritten. By grouping LBAs with similar death-times into the same block and assuming there are sufficient number of concurrently opened blocks, a write-amplification of 1 can be achieved, which represents an ideal data placement strategy. The idea of grouping blocks using death-times has been proposed before for the application layer [17]. In contrast to this work, we leverage death-time within the FTL, transparently to the user, for minimizing write-amplification. We introduce the basic operating principle of our death-time aware placement technique with an example shown in Table 1. In this example, we assume an SSD where each block can contain only two LBAs and we observe writes to three different LBAs, `A`, `B` and `C`. In Table 1, the first row shows the elapsed time, the second row shows the write sequence of LBAs, and the third row shows the absolute death-time for each LBA write. Rows 4 through 6 show three different allocation policies and how they place LBAs into blocks. Every unique block being used is represented by a color. Furthermore, the rows show the number of blocks in use at every time step. The policies strive to utilize as few blocks as possible to minimize overprovisioning, respectively write-amplification.

The fourth row shows how a conventional baseline FTL that applies writes sequentially to a single open block absorbs the write sequence. The first two blocks are written into the orange block. At time 3 the orange block is closed and the blue block is opened. At time 5, the orange and blue closed blocks still contain valid LBAs (based on death-time information on 2nd row). As a result, the green block is opened absorbing the next writes. At time 7, the blue block can be reused as all LBAs within it have been overwritten. This policy requires *three* blocks to absorb the write sequence.

Time	1	2	3	4	5	6	7	8
LBA	C	A	B	B	C	B	A	A
Death-Time	5	7	4	6	97	98	8	99
Baseline	1	1	2	2	3	3	3	3
Frequency	1	2	2	3	3	3	3	3
Oracle-DT	1	2	2	2	2	2	2	2

Table 1: Baseline vs. Frequency vs. Oracle-DT Policy

The fifth row in Table 1 shows the operation of a hot-cold frequency-based allocation policy. LBA `C` is considered a cold LBA, whereas A and B are considered hot LBAs. As a result, at time 1, `C` is placed into the orange block, whereas A and B are placed into the blue block at times 2 and 3 respectively. At time 4, the blue block is closed and a new hot block (green) is opened. At time 7, both `A` and `B` within the blue block have been overwritten, enabling to reuse the blue block at time 7. This policy requires *three* blocks to absorb the write sequence.

The sixth row of Table 1 shows how the death-time allocation policy places LBAs into blocks. LBA `C`, written at time 1, and LBA `B`, written at time 3, have have similar death-times of 5 and 4 respectively and are hence placed into the orange block. The LBAs written at time 2 and time 4 are placed into the blue block. At time 5, the death-times of all LBAs in the orange block have elapsed and hence it can be reused. *Oracle-DT* minimizes the number of blocks currently in use requiring *two* blocks to absorb the write sequence.

Frequency-based policies suffer from the fact that they classify hot and cold blocks in advance. *Oracle-DT* on the other hand, ignores the write frequency of individual LBAs, potentially placing hot and cold LBAs into the same block as long as they share a similar death-time. In Section 5, we show that a death-time policy with perfect future knowledge (*Oracle-DT*) and sufficient concurrently opened blocks can provide an ideal write amplification of 1.

3.1 Death-Time Analysis

While *Oracle-DT* eliminates write amplification, it is impractical, as it requires perfect knowledge of future writes. It also requires a large number of open blocks at the same time to capture the variability of death-times. This is illustrated in Figure 2 showing the LBA death-times for a write access sequence for the VDI application. Death-times vary significantly and hence a large number of concurrently open blocks is required to capture all active death-times. To address these limitations, we devise a practical solution, by predicting the death-times of LBAs with a machine learning technique and then mapping the death-time ranges to a fixed number of open blocks. Our proposal is based on the analysis of over 700 million written LBAs from 10 real-world traces

from the SNIA repository (VDI [49], RocksDB [49], and tpc-h benchmark (MonetDB)) [29, 44], from which we make the following observations.

(1) Real-world workloads exhibit skewed write patterns (see Figure 1) where a significant number of I/O write accesses are covered by only a few LBAs. As a result, the distribution of LBAs has a long tail.
(2) The death-times of LBAs vary widely, from a few I/Os to thousands of I/Os (see Figure 2). Separating I/O streams based on death-times enables the invalidation of pages within a block solely via user-writes.
(3) User-writes originating from similar locations and time tend to have similar death-times in line with prior work [46], [50]. Hence, the LBAs of I/O writes can be used to separate streams of data with similar death-times together within the same open block.
(4) Real-world workloads often contain multiple jobs running in parallel, causing interleaved write patterns [2, 3, 8, 16] and death-times.
(5) For a given LBA, death-times change over time. Therefore, using prior history of assigning LBAs to blocks can be inefficient, resulting in higher WA.
(6) The requested I/O sizes for the analyzed real-world applications range from 4KB to several MBs, with up to 10,000 different I/O sizes for an individual application, motivating a technique that considers I/O size for computing death-times [8].

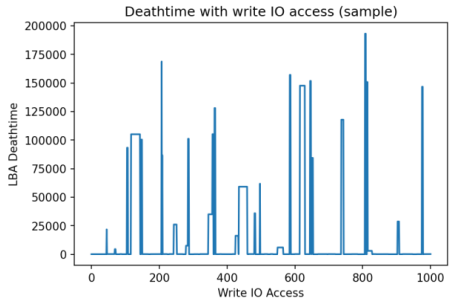

Figure 2: Death-time variation over time

3.2 Learning Death-Time Patterns

Based on the observations above, we developed a machine learning based technique to predict future LBA death-times. As shown earlier, the death-time patterns follow a skewed distribution which depends on both the spatial and temporal properties of I/O write accesses. Furthermore, I/O accesses are sequentially dependent on each other and follow a sparse distribution. Sequence models for time-series data have been shown to be effective in leveraging the spatial and temporal patterns. Some sequence models, such as LSTMs [18] and GRUs [1], also have an attached memory which allows the models to look at recent previous writes to generate effective predictions. We train the machine learning models on real-world trace data. The traces were pre-processed by first removing all the read operations and then determining the death-time for each write operation. Without loss of generality, we express death-time as a monotonically increasing counter that is incremented at every write. We express the problem of predicting the next death-time of an LBA (Next-DT) as a sequence learning problem utilizing three main features: (1) the logical block address, (2) the I/O size, and (3) the previous death-time (Prev-DT) of an LBA.

Next-DT. The goal of *ML-DT* is to accurately predict the death-time of a written LBA. As an SSD block contains multiple (e.g., 64) LBAs, each block covers a death-time range. Furthermore, as the number of open blocks in a practical SSD is limited, we cannot assign an open block to each existing death-time range. As a result, we partition the death-times into N ranges where N reflects the number of open blocks. Instead of predicting the exact death-time for each LBA write, we only predict the death-time range, corresponding to the block that the LBA should be written to. The number of output labels (next-DT) is hence equal to the number of open blocks offered by an SSD.

Logical Block Address. The LBA range supported by modern terabyte sized SSDs is large (exceeding 30 bits) and sparse (applications cover only a subset of LBAs), and hence is difficult to learn for a machine learning model. To address this challenge, we first partition the LBA into a high and a low part. This not only reduces the number of bits of the feature vector but also exploits the fact that different streams within an application can be often identified via the high order bits of an LBA. We experimented with more than two LBA range partitions, however, we did not see additional benefits. The raw LBA values were normalized between 0-1, which allowed the model to learn the locality and sparsity of the I/O write accesses. To address sparsity, we leverage an embedding layer [33] of 500 neurons to map the sparse LBA inputs to a dense internal feature vector. As the distribution of IO writes follows a sparse pattern (most I/O accesses target a few LBAs), we represented it by a sparse vector (embedding layer) as they can represent the input more effectively using less data. The key to this approach is the concept of using a dense distributed representation for each input value. Embedding captures semantic similarities between data points places them close to each other in the embedding space.

I/O Size. Applications can update the same LBA using different I/O sizes, writing multiple LBAs in the process with a single I/O operation. We explored two techniques to handle this. In the first approach, we split every multi-LBA write into multiple single-write LBAs and then predict the

death-time for each single LBA write individually. In the second approach, we leverage I/O size as another feature to enable the ML model to capture the I/O size internally. Both techniques provided equal performance, and hence we opted for the second simpler technique.

Prev-DT. Our model leverages the previous death-time of an LBA as feature. Similarly, next-DT and prev-DT also reflect death-time ranges, instead of precise death-times. Hence, the model prediction is based on the open block to which the LBA was written. The storage overhead for maintaining prev-DT for each LBA is hereby bounded by the number of open slots. Prev-DT of an LBA can be derived from the block ID that the LBA is currently located in (before the overwrite) and hence does not introduce significant meta-data storage overheads.

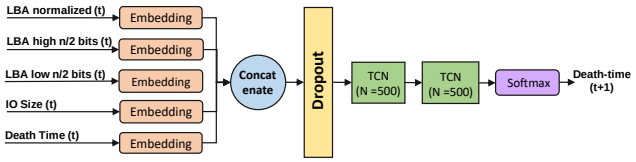

Figure 3: Model Architecture

Model Architecture. The proposed model architecture is shown in Figure 3. The inputs to the model are categorical, one-hot representation of the input features, each being fed to a separate embedding layer for dimensionality reduction. The sparse embedding vectors are concatenated and fed through a Dropout [42] of 0.2 to prevent overfitting of the model. Death-time prediction is performed by a temporal convolutional network (TCN) [28], a variant of convolutional neural network (CNN) [36], employing causal convolutions [4] and dilations [52] to learn from sequential data with temporality. As TCNs implement memory (causal dilated convolutions), it considers recent data to differentiate between interleaved I/O accesses, enabling effective death-time predictions. TCNs also track the behavior of I/O accesses and how they evolve over time to enable accurate predictions based on the current state of the system.

We empirically observed that two hidden TCN layers are sufficient, where each layer includes 500 neurons. The final output layer is a dense layer consisting of Softmax [13] nodes. The number of neurons in the output layer is set to the number of open blocks available in the SSD. Our proposed machine learning model is application-specific and needs to be trained on relevant traces. To improve the generality of our technique and to alleviate the deployment in a real system, Section 4.4 introduces a mapping learning technique that enables reusing of the same model across different traces.

We also explored different machine learning models such as LSTM [18], SVM [19], and Random Forest [48] to perform death-time prediction, however, TCNs turned out to be superior. In particular, TCN allows parallelism of the computed convolutions since the same filter is used in each layer. The convolutions in the architecture are causal, which means that there is no information "leakage" [10] from future to past. TCNs also consume less resources for training and can take in inputs of arbitrary lengths by varying the 1D convolutional kernels [47]. TCNs are capable of effectively capturing very long history sizes (i.e., the ability for the networks to look very far into the past to make a prediction) by using a combination of deep networks (augmented with residual layers) and dilated convolutions [52]. LSTM is chosen as a baseline as it is a popular DNN based technique used in time series forecasting in multiple applications [19], and it also has a memory to look at recent data for handling sequential time series data. Random Forest [48] and SVMs [19] are two popular ML classification algorithms which do not take into account the time series nature of the data. We also use a random classifier which randomly picks a block, as a baseline. We replace the LSTMs with TCN in our model architecture and RF and SVM models are fed 2-dimensional data as input.

3.3 *ML-DT* Flash Translation Layer

To leverage *ML-DT*, our model needs to be integrated into an FTL. We assume a flash based system supporting $N+1$ append points or open blocks, where $N > 1$ and N open blocks are assigned for servicing user-writes and one open block is assigned for servicing GC-writes. Although *Oracle-DT* does not need a GC block by eliminating write amplification, *ML-DT* cannot provide such a strict guarantee, and hence there needs to exist a block to absorb rare GC-writes. Each one of the open user-write blocks is assigned a death-time range and for each user-write, the block with the closest death-time range is chosen for placing the write. For instance, LBAs within the range $0-100$ are directed to the first open block, LBAs within the range $101-300$ are directed to the second open block, et cetera. For a system with n open blocks, the ranges are set according to the n^{th}-percentile of death-times. Each open block keeps track of the start LBA of the block, valid pages bitmap, write pointer, death-time original, death-time counter, and a status flag. When the write pointer reaches the maximum pages per block, the block is closed and a new block is requested, initialized with the death-time of the block that was just closed. The death-time-original is set to the upper limit for the death-time range for the particular block. The death-time counter is initialized as death-time-original and is decremented after every I/O. Each block maintains a death-time counter which is initialized as maximum value for the range, and is decremented for each I/O. If the death-time range of a block is chosen optimally, the death-time counter reaches 0 only when the block is full.

However, as the ML model is imperfect and the number open blocks is limited, we need to handle the case of non-optimal death-time assignments. In particular, in the case where the death-time counter of a block reaches 0 and the block is not full, it will be assigned incoming pages from the adjacent (nearest) two death-time ranges to close the block as quickly as possible. For instance, assuming the SSD has 10 open blocks for user writes, if block 2 (range $11 - 20^{th}$ percentile) is not full while the death-time counter reaches zero, pages usually assigned to to the 1^{st} and 3^{rd} percentile will be temporally assigned to block 2. For edge cases, block 1 ($0 - 10^{th}$ percentile) and block 10 ($91 - 100^{th}$ percentile), we use the nearest two open blocks. Hence, by increasing the death-time range of the block, it will absorb more writes, getting closed faster. By absorbing other block's writes, additional non-full blocks are generated with a death-time of zero. To address this challenge, whenever a non-full block reaches a death-time of zero, the death-times are re-computed for all the three blocks using the following formula: $death\_time\_new = (page\_per\_block - writepointer)/100 * death\_time\_old$. If after a programmable amount of time, the block still does not get closed, all future incoming I/Os, referred to as *priority writes* are redirected to said block. We keep track of the number of user-writes (UW), GC-writes (GW), and priority-writes (PW) required to store our data and compute WA as $WA = UW/(UW + GW + PW)$.

4 IMPLEMENTATION

4.1 Datasets and Data Preparation

We leverage traces from ten real-world workloads from three different sources for our experiments. We parse the traces to transform LBA and I/O size into numeric features. We separate each LBA into two parts: the upper and lower significant half of the bits are separated and hashed into values in the range $0 - 100$. This allows the model to distinguish between the interleaved I/O write accesses using the higher bits while predicting in a stream using the lower bits. In addition, the full LBA is normalized to a value between $0 - l$, where l is the LBA size. The requested I/O sizes for the examined real-world applications ranges from 4KB to several MB with up to 10,000 different I/O sizes for individual applications. In order to reduce the number of possible I/O size values, we round-off each observed I/O size, m, to the nearest number, $s = 2^m$, and use s as an I/O size class. This reduces the number of possible I/O sizes to 16 while still supporting requests of sizes of up to 64MB. We remove the I/O accesses that do not have a death time towards the end of the trace.

4.2 Machine Learning Models

Training of machine learning models was performed on a NVIDIA Titan-X GPU. For each incoming I/O, we examine the input features (LBA, LBA-high, LBA-low, and I/O_size) to predict the LBAs death-times. The training time for one epoch for TCN based models ranged between 84 and 192 seconds, depending on the trace. Here, we observed that the deployed TCN [28] models were significantly faster to train than comparable LSTMs [5], while providing higher performance. The inference was performed on one Intel Xeon CPU core running at 1.7 GHz, and the inference times ranged between 102 and 315 μs. Data to the model was fed in batches of 64. We used *tanh* activation function [22] and Adam optimizer [53]. The models were trained for 10 to 32 epochs until convergence [37]. Each class predicted by the model represents a range of death-times based on the percentile value of death-times observed in 10% of the randomly selected subset of the trace file. The number of classes (open blocks), representing the death-time ranges, is adjustable during training time. In Section 5, we show that 20 open blocks are sufficient to minimize WA for *ML-DT*.

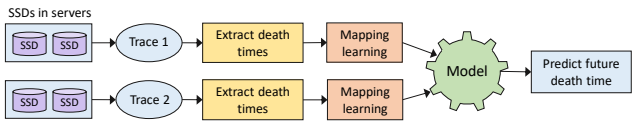

Figure 4: Mapping Learning Architecture

4.3 FTL Simulator

In order to compute WA for each workload using different data placement schemes, we developed a FTL simulator that uses virtual SSDs, where the SSD size can be adjusted for each trace. The SSD size can be computed based on the number of unique LBAs in the trace, and the number of open blocks available can be varied. Whenever the number of free blocks is lower than a threshold value (0.1% of total free blocks, a.k.a. GC Threshold), the simulator picks the block with least number of valid pages for recycling. We use a page size of 4K, where each block contained 64 pages and has a size of 256K. The trace simulation was performed on a single Intel Xeon CPU core running at 1.7 GHz with 16 GB RAM.

4.4 Mapping Learning

To support dynamically changing workloads and to support previously unseen workloads, we propose a mapping learning technique to determine whether models can learn generalized death-time patterns from complex I/O write patterns. Different workloads show similar I/O write patterns due to shared design patterns and commonly used data structures [8]. For example, array-based data structures used by applications generally entail sequential I/O access patterns. Furthermore, as applications generally leverage the same underlying file system, it is likely that I/O writes show common

patterns. An ideal model would need to be be trained once on a varied set of applications, and yet accurately predict death-times for unseen applications. Such a model architecture is also likely to be more robust with respect to dynamically changing data inputs or code changes to the original application. To test the idea that applications share common write patterns that can be learned, we trained the model on traces from one dataset (source) and evaluated the model's performance on another dataset (recipient). The number of prediction classes are kept same in the two workloads and a 1-1 label mapping is done in sequential order. For instance, Class 1 from the first workload corresponds to Class 1 of second workload. We call the process of training the model on source dataset, and using it to predict death-times for the recipient dataset as *Mapping Learning*, and present a block diagram of this process in Figure 4.

5 RESULTS

In this section, we first analyze the performance of ML models for predicting death-time ranges. Then, we evaluate *ML-DT* via trace-driven simulation using real-world cloud storage traces collected at the block level. Then, we compare our approach with three existing data placement schemes and two baselines (*DT-FTL-Greedy* and *Oracle-DT*). We also perform sensitivity studies determining the impact of the number of available open blocks on WA, and finally present the results of our proposed mapping learning technique.

5.1 Evaluation of ML Models

We utilize four baselines to compare our ML approach: Random forest (RF) [48], support vector machines (SVM) [19], DNN based LSTMs [18], and a Random predictor (RP). To evaluate the performance of LSTMs, we replaced the TCN layers in our model with LSTMs layers. RF and SVM models were fed 2-D data as input, as they cannot take in 3D embedding layers as input. Table 2 shows the comparative performance of our ML based approach against the four chosen baselines when using 19 death-time classes, as one class is reserved for GC-writes. Note that each class represents a death-time range that is computed by equally dividing the death-times into 19 groups, each representing the (100)/19*pth percentile of the data in each class where p=[1,2,3..19]. The table provides detailed characteristics of the traces [27, 29, 41, 49], including trace name, number of I/Os in the dataset, and the accuracy of the four chosen baselines for predicting death-times. The LBAs without death-time information are excluded from the training data and we train our models using the first 50% of data in the trace and evaluate it on the second 50% of the trace. The size of the input traces ranged between 0.47 GB and 43 GB, depending on the source. All the workloads used in this study were write-heavy, with read-write ratios less than 0.5, as can been seen in Table 2. For this experiment, we used block size of N, assuming that the system has $N+1$ blocks, as one open block is reserved for GC-writes. We show the impact of using different number of open blocks later in Section 5.3. The results in Table 2 show that TCN-based approach consistently outperforms other baseline techniques. The random predictor performs the worst, achieving lowest accuracy (3.6 - 5.6%). Results from the LSTM-based approach are comparable with our approach (within 5% of accuracy). However, TCN-based models train faster and can support higher dimensional data without prohibitive compute resources. TCN based models train 10× faster and reduce inference latency 2× compared to LSTM based models. RF and SVM based approaches perform significantly worse, achieving highest accuracy of only 61% and 56%, respectively, on real-world traces. We also used three synthetic traces as baselines to test our ML approach for predicting death-time ranges. The first two traces contained only sequential I/O writes and random I/O writes, respectively, and the third trace contained a mix of the 50% synthetic and 50% random workloads.

5.2 Comparison with Baselines

The predictions from ML models are leveraged by the FTL for assigning pages to open blocks. Here, we compare our approach with three existing data placement policies proposed in prior work that are based on update frequency, hot/cold separation, and block update interval. We also used two baselines, *DT-FTL-Greedy* and *Oracle-DT*, the later of which have perfect knowledge of future death-times. We provide a description of these five baselines below.

(1) *DT-FTL-Greedy*: Utilizes a single open block for servicing both user and GC writes. When the FTL runs out of free blocks, GC greedily picks the block with the fewest number of valid pages for cleaning.
(2) *Oracle-DT*: Has perfect knowledge of future death-times and uses the same placement policy as *ML-DT*.
(3) Dynamic Data Clustering (DAC) [12]: Maintains multiple open blocks, and each one assigned a temperature. Whenever an LBA is overwritten by a user-write, it is promoted to a hotter block, and whenever an LBA is moved by GC it is demoted to a colder block.
(4) WARCIP [51]: Writes blocks into segments based on the block update interval, i.e. the elapsed time since the last write to the same LBA, in order to reduce the variance of update intervals of pages within a block.
(5) AutoStream (AS) [50]: Leverages both write frequency and recency to determine the block temperature, for writing pages into the blocks of different temperature levels, and demotes aged LBAs into cold blocks.

Trace Source	Trace Size	File Name	No. of IO (millions)	R-W ratio	No_unique _LBA	Coverage _Top 100	Coverage _Top 1K	Coverage _Top 10K	Accuracy TCN (%)	Accuracy LSTM (%)	Accuracy RF (%)	Accuracy SVM (%)
synthetic_sequential	4.7 GB	synthetic_A	1.3	0	1310721	0.07	0.7	7.6	99	99	91	83
synthetic_random	5.0 GB	synthetic_B	1.04	0	262145	0.27	3.6	4.5	21	17	6	2
synthetic_mixed	5.0 GB	synthetic_C	1.14	0	12145	0.21	2.1	6.6	47	41	39	19
VDI	0.7 GB	2016030917.csv	2.47	0.45	832659	22.1	57.7	63.6	64	57	47	33
	1.64 GB	2016031115.csv	2.4	0.15	430365	21.04	72.6	79.7	65	56	41	37
	1.16 GB	2016030918.csv	4.5	0.44	1370898	24.9	62.2	65.9	66	58	38	29
	1.84 GB	2016030819.csv	2.9	0.27	677630	17.5	69.8	74.4	64	59	33	47
	0.47 GB	2016030916.csv	3.75	0.35	1049707	22.4	61.4	66.5	66	60	49	48
RocksDB	1 GB	ssd-trace00	1.95	0.22	223168	7.5	30.1	36.3	70	59	43	51
	22 GB	ssd-trace0-15	112.3	0.19	1562176	4.8	9.6	11.1	64	57	49	35
	21 GB	ssd-trace16-37	116.4	0.22	1332712	9.2	14.6	16.1	64	59	41	38
	17 GB	ssd-trace-add.	97.2	0.33	1123568	8	16.5	17.9	69	59	43	33
TPC-H (MonetDB)	12 GB	tpc-h-monet	66	0.5	1332176	5.1	25.7	33.3	83	73	61	66

Table 2: Comparison of machine learning approaches for death-time range prediction.

We evaluate the performance of our proposed *ML-DT* approach against the five baselines in Figure 5. For each FTL, we count the number of user-writes and GC-writes for computing the WA. The SSD size used for performing the experiments is based on the number of unique LBAs in the trace such that the user capacity of the SSD equals the number of unique LBAs. The over-provisioning ratio was assigned to 20% and hence, the number of available blocks equals 1.2× the number of user blocks. The GC threshold, defined as the minimum number of available free blocks before GC is enabled, was chosen as 0.1% of total number of blocks available initially. For each trace, we note the range of unique LBAs (A-B) and create Z number of blocks determined by the range computed as : (B-A)*page_size where page_size = 4K and each block contained 64 pages. The coverage of top 100, 1K, and 10K LBAs is also reported showing the skewed nature of write I/O accesses in the traces.

For this experiment, we used 20 open blocks for all baselines as well as *ML-DT*, except for *DT-FTL-Greedy* which only uses one. Figure 5 shows that *ML-DT* consistently outperforms the baselines for every trace both in terms of GC overhead and WA. Our approach reduces the number of GC writes due to effective placement and the priority write overhead is minimal, which causes less than 1% overhead. On the other hand, DAC performs the worst while WARCIP and AS perform the best among the baselines, achieving comparable performances on TPC-H benchmarks and lower number of open blocks, however they perform much worse on VDI based workloads. *Oracle-DT* achieves near optimal WA for RocksDB and VDI traces and as expected performs best.

Our approach works best for VDI and RocksDB traces, where the I/O write patterns are non-uniform and I/O accesses are concentrated more on a few frequently occurring LBAs. The performance is comparatively worse for traces which have more uniform distribution of LBAs in the trace (e.g., TPC-H). Higher coverage workloads helps in stream prediction due to greater density of input vectors fed in for training, and hence can make more accurate predictions.

5.3 Impact of number of open blocks

In this section, we study the impact of the number of open blocks available on WA. Figure 6 compares the performance of our proposed approach with the baselines by varying the number of open blocks available. As DAC does not differentiate between user-written and GC-written LBAs, all N open blocks are made available for both write types. Since AutoStream and WARCIP focus on separating only the user-written pages, we configure $N-1$ classes for user-written pages and one class for GC-written pages. These results show that our approach is comparable to baselines when using a small number of open blocks (<3-5), however, as the number of blocks are increased to between 10 and 30, our approach outperforms the baselines by up to 19%, demonstrating the generalizability of our approach with varying number of open blocks. Future SSD designs are expected to support an increasing number of open blocks. However, albeit outperforming the baselines, increasing the number of open blocks to beyond 20 increases WA due to decreased predictive performance and priority writes overhead. As the data gets more fragmented, that is, separated into multiple streams, more cases arise where the death-time counters of the blocks reach zero. In such scenarios, as described earlier, our approach merges blocks by assigning pages to the closest block with similar death-times, and closing some of the blocks earlier.

5.4 Sensitivity Study on Open Blocks

In this section, we perform a sensitivity study to evaluate the impact of varying the number of open blocks on WA. For each trace, we vary the number of open blocks between 5 and 30. We see that increasing the number of blocks increases the WA initially due to better data organization within the SSD, however, as we increase the number of open blocks to over 25, we see a decrease in WA due to decreased predictive performance and fragmentation of data. This is due to fragmentation, the DT_counter of blocks reaches 0 without the block being full resulting in the priority write overhead. The trend can be seen in Figure 7.

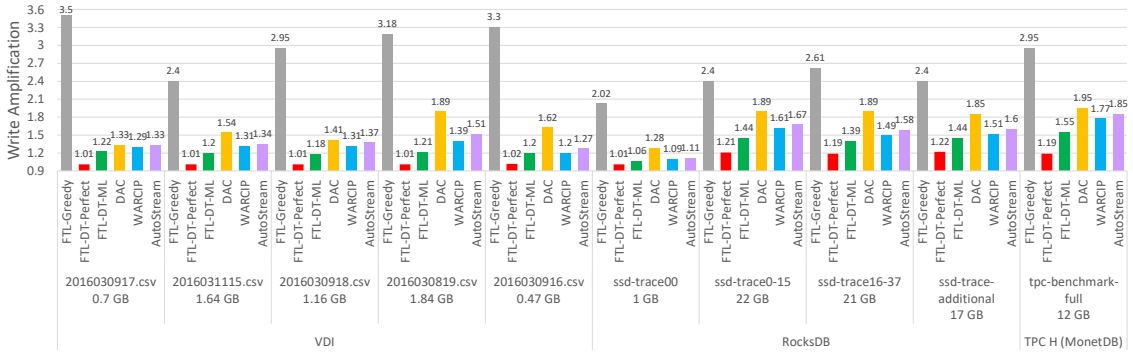

Figure 5: FTL comparison with baselines

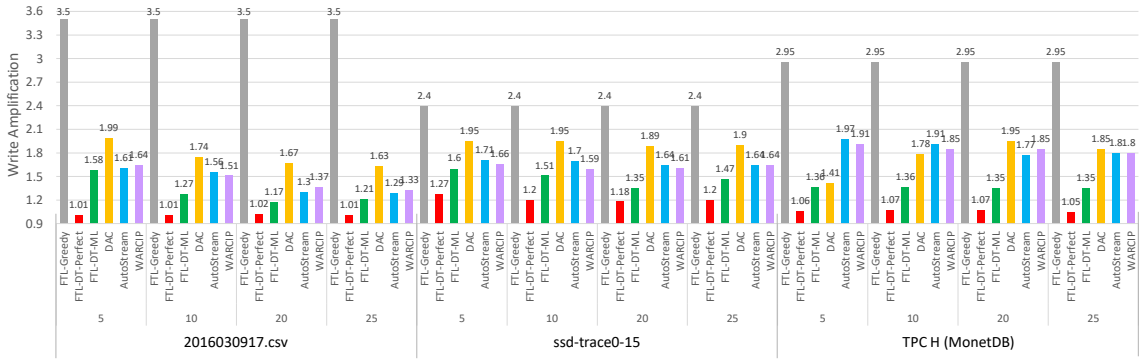

Figure 6: Impact of number of open blocks

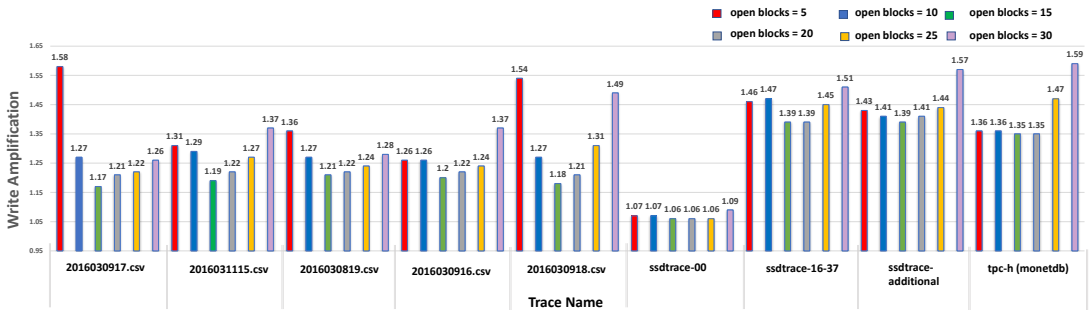

Figure 7: Sensitivity Study (*ML-DT*)

5.5 Evaluation of Mapping Learning

In this section, we evaluate whether our ML models can learn common patterns across workloads to predict death-times of previously unseen applications. In the previous sections, we obtained the training and test datasets from different portions of the same workload and trace file. In this section, we define two types of dataset sources. Similar sources are those where the training and test data are from the same application, however, with different data inputs, different execution times, and only small run-time modifications in applications. Dissimilar sources are those where the training and test data are from completely different applications. Figure 8 shows the performance overhead of the mapping learning technique. We show a comparison of write amplification between the model that is trained and tested on similar source traces (WA_original) and the model that is trained and tested on the dissimilar source traces (WA_mapped). The model is trained on the Source (S) trace and tested on the Recipient (R) trace. For instance, when training the model on ssd-trace-01-15 and evaluated on ssdtrace-16-33 trace files,

the WA of our mapping learning approach is 1.44, which is 4% less compared to training and evaluating the model on the same ssdtrace-16-37 trace file. Note that when the model is trained and tested on dissimilar sources, our approach does not make any assumptions about the sequence of LBAs in the recipient source, and during inference on the recipient trace, the number of classes as well as the death-time ranges based on percentile values were kept same as the source (S). The effectiveness of Mapping Learning depends on the frequency distribution of data within each class of the two datasets. Results in Figure 8 show that our approach is applicable to diverse workloads, as long as they show similar characteristics. This increases the practicality of our approach, as we can train specific models for a variety of workloads, and expect at least a moderate increase in performance.

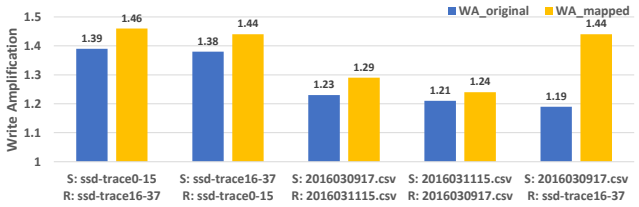

Figure 8: Mapping Learning Technique

6 RELATED WORK

Prior works on reducing WA proposed a greedy reclaiming policy [14, 15] for selecting the block with least number of valid pages for GC, reducing live data migration. Our work follows Greedy to select a block for GC, however, it additionally minimizes the number of valid pages within a block. Multi-Stream SSDs [20] have been introduced to expose multiple open blocks enabling applications to explore different placement policies. Prior work on different data placement policies can be broadly classified into two categories: data separation based on the access frequency of LBAs and those based on clustering. SFS [34] writes blocks into large segments in batches based on the write counts of an LBA divided by the block age. LOCS [45] follows a similar approach of using longer segments optimized for LSM-tree based workloads, however, their performance is limited on real-world applications, where smaller write units dominate I/O accesses. PCStream [23] automatically selects open blocks based on program counters in the Linux kernel. Extent-based identification (ETI) [40] tracks the number of writes to each LBA and separates hot blocks as those whose write counts exceed a pre-defined threshold. Fading Average Data Classifier (FADaC) [26] uses write frequency and recency to allocate incoming writes to open blocks by maintaining a fading average write frequency for each block. Our placement strategy, on the other hand, is based on the knowledge of death-times of the incoming writes, thereby, reducing WA over prior techniques. Dynamic data Clustering (DAC) [12] allocates LBAs to blocks based on their temperature. User-writes promote an LBA to a hotter segment while GC-writes demote an LBA to a colder segment. Multi-Log [43] follows the same approach leveraging the update frequency of each LBA to assign it to an open block. AutoStream [50] uses write frequency and recency to compute the temperature of incoming writes and assigns them to open blocks based on temperature levels. Each block is assigned a different temperature level and old blocks are demoted to cold segments. Grouping LBAs by their death-time was first proposed by He [17]. WARCIP [51] uses the rewrite interval (time elapsed since last write to the LBA) to allocate incoming writes to open blocks in order to reduce the variance of update intervals of LBAs in a block. InferBIT [46] uses inferred the block invalidation time to minimize WA in log-structured storage by placing blocks with similar estimated BITs into the same group. As shown in Section 3, *Oracle-DT* reduces WA over prior works by leveraging death-time instead of write frequency. Several other works have applied ML to optimize storage systems [6, 7, 9, 21, 25, 32].

7 CONCLUSION

In this paper we introduced *ML-DT*, an ML-based approach to reduce write amplification (WA) in log-structured file systems by predicting the death-time of logical block addresses. We leverage time series data in of I/O accesses to train lightweight, TCN-based models to predict death-time ranges of LBAs. Additionally, we propose a near-optimal data placement technique based on death-time which results in minimal write amplification in log-structured file systems. Using the proposed data placement scheme, we present the design of *ML-DT* and compare our approach with three state-of-the-art data placement schemes. We show that *ML-DT* achieves lowest WA by leveraging the learned I/O write patterns from real-world storage workloads. Our approach results in up to 14% WA reduction compared to the best baseline technique while providing better scalability for large numbers of open blocks. We provide insights on the type of workloads benefiting the most using our approach. Finally, we present mapping learning to test applicability of our approach on new unseen workloads and present a feasibility study to demonstrate the applicability of our work to unseen workload traces.

8 ACKNOWLEDGEMENTS

This work has been generously supported by Samsung Semiconductor Inc. and NSF grants CCF-1942754 and CNS-1841545.

REFERENCES

[1] Ben Athiwaratkun and Jack W Stokes. 2017. Malware classification with LSTM and GRU language models and a character-level CNN. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, IEEE, USA, 2482–2486.

[2] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, USA, 513–526.

[3] Peter Braun and Heiner Litz. 2019. Understanding memory access patterns for prefetching. In *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*. NSF, USA, 10187649.

[4] Thomas J Brazil. 1995. Causal-convolution-a new method for the transient analysis of linear systems at microwave frequencies. *IEEE transactions on microwave theory and techniques* 43, 2 (1995), 315–323.

[5] Jason Brownlee. 2018. *Deep learning for time series forecasting: predict the future with MLPs, CNNs and LSTMs in Python*. Machine Learning Mastery, USA.

[6] Chandranil Chakraborttii and Heiner Litz. 2020. Explaining SSD Failures using Anomaly Detection. *Non-Volatile Memory Workshop* 1, 1 (2020), 1.

[7] Chandranil Chakraborttii and Heiner Litz. 2020. Improving the accuracy, adaptability, and interpretability of SSD failure prediction models. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. ACM, USA, 120–133.

[8] Chandranil Chakraborttii and Heiner Litz. 2020. Learning I/O Access Patterns to Improve Prefetching in SSDs. *ECML-PKDD* 1, 1 (2020), 14.

[9] Chandranil Chakraborttii, Vikas Sinha, and Heiner Litz. 2018. Ssd qos improvements through machine learning. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, USA, 511–511.

[10] Cen Chen, Bingzhe Wu, Minghui Qiu, Li Wang, and Jun Zhou. 2020. A Comprehensive Analysis of Information Leakage in Deep Transfer Learning. *arXiv preprint arXiv:2009.01989* 1, 0 (2020), 0000–0000.

[11] M-L Chiang and R-C Chang. 1999. Cleaning policies in mobile computers using flash memory. *Journal of Systems and Software* 48, 3 (1999), 213–231.

[12] Mei-Ling Chiang, Paul CH Lee, and Ruei-Chuan Chang. 1999. Using data clustering to improve cleaning performance for flash memory. *Software: Practice and Experience* 29, 3 (1999), 267–290.

[13] Steven Gold, Anand Rangarajan, et al. 1996. Softmax to softassign: Neural network algorithms for combinatorial optimization. *Journal of Artificial Neural Networks* 2, 4 (1996), 381–399.

[14] Longzhe Han, Yeonseung Ryu, and Keunsoo Yim. 2006. CATA: a garbage collection scheme for flash memory file systems. In *International Conference on Ubiquitous Intelligence and Computing*. Springer, USA, 103–112.

[15] Long-zhe Han, Yeonseung Ryu, Tae-sun Chung, Myungho Lee, and Sukwon Hong. 2006. An intelligent garbage collection algorithm for flash memory storages. In *International Conference on Computational Science and Its Applications*. Springer, USA, 1019–1027.

[16] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. In *International Conference on Machine Learning (ICML)*. PMLR, USA, 1919–1928.

[17] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. The unwritten contract of solid state drives. In *Proceedings of the Twelfth European Conference on Computer Systems*. European Conference on Computer Systems, USA, 127–144.

[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[19] Shujun Huang, Nianguang Cai, Pedro Penzuti Pacheco, Shavira Narrandes, Yang Wang, and Wayne Xu. 2018. Applications of support vector machine (SVM) learning in cancer genomics. *Cancer Genomics-Proteomics* 15, 1 (2018), 41–51.

[20] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The multi-streamed solid-state drive. In *Hot Topics in Storage and File Systems (HotStorage 14)*. 6th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 14), USA, 1–26.

[21] Saeed Kargar, Heiner Litz, and Faisal Nawab. 2020. Predict and Write: Using K-Means Clustering to Extend the Lifetime of NVM Storage. *arXiv preprint arXiv:2011.02556* 1, 1 (2020), 1–1.

[22] Bekir Karlik and A Vehbi Olgac. 2011. Performance analysis of various activation functions in generalized MLP architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems* 1, 4 (2011), 111–122.

[23] Taejin Kim, Sangwook Shane Hahn, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. 2018. PCStream: automatic stream allocation using program contexts. In *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. Hot Topics in Storage and File Systems, USA, 1–12.

[24] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. Reflex: Remote flash==local flash. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 345–359.

[25] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2018. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 {USENIX} Annual Technical Conference ATC 18)*. ACM, USA, 759–773.

[26] Kevin Kremer and André Brinkmann. 2019. FADaC: A self-adapting data classifier for flash memory. In *Proceedings of the 12th ACM International Conference on Systems and Storage*. ACM International Conference on Systems and Storage, USA, 167–178.

[27] Arezki Laga, Jalil Boukhobza, Michel Koskas, and Frank Singhoff. 2016. Lynx: A learning linux prefetching mechanism for ssd performance model. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, IEEE, USA, 1–6.

[28] Colin Lea, Rene Vidal, Austin Reiter, and Gregory D Hager. 2016. Temporal convolutional networks: A unified approach to action segmentation. In *European Conference on Computer Vision*. Springer, Springer, USA, 47–54.

[29] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. 2017. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference*. ACM International Systems and Storage Conference, USA, 1–11.

[30] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*. {USENIX} Conference on File and Storage Technologies, USA, 273–286.

[31] Heiner Litz, Javier Gonzalez, Ana Klimovic, and Christos Kozyrakis. 2021. RAIL: Predictable, Low Tail Latency for NVMe Flash. *ACM Transactions on Storage (TOS)* 1, 1 (2021), 1–1.

[32] Heiner Litz and Milad Hashemi. 2020. Machine Learning for Systems. *IEEE Micro* 40, 5 (2020), 6–7.

[33] Yusuke Manabe and Basabi Chakraborty. 2007. A novel approach for estimation of optimal embedding parameters of nonlinear time series by structural learning of neural network. *Neurocomputing* 70, 7-9 (2007), 1360–1371.

[34] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: random write considered harmful in solid

state drives.. In *FAST*, Vol. 12. {USENIX} Conference on File and Storage Technologies, USA, 1–16.

[35] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. 2014. Improving performance and lifetime of the SSD RAID-based host cache through a log-structured approach. *ACM SIGOPS Operating Systems Review* 48, 1 (2014), 90–97.

[36] Keiron O'Shea and Ryan Nash. 2015. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458* 1, 1 (2015), 14.

[37] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. 2019. On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237* 1, 1 (2019), 14.

[38] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Jooyoung Hwang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. 2018. FStream: Managing flash streams in the file system. In *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*. {USENIX} Conference on File and Storage Technologies, USA, 257–264.

[39] Mendel Rosenblum and John K Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 26–52.

[40] Mansour Shafaei, Peter Desnoyers, and Jim Fitzpatrick. 2016. Write amplification reduction in flash-based SSDs through extent-based temperature identification. In *8th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. {USENIX} Workshop on Hot Topics in Storage and File Systems, USA, 1–8.

[41] SIA. 2001. Block IO Traces. http://iotta.snia.org/tracetypes/3. Accessed on 2021-01-11.

[42] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[43] Radu Stoica and Anastasia Ailamaki. 2013. Improving flash write performance by using update frequency. *Proceedings of the VLDB Endowment* 6, 9 (2013), 733–744.

[44] Gongjin Sun and Sang-Woo Jun. 2020. ColumnBurst: a near-storage accelerator for memory-efficient database join queries. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM, USA, 9–16.

[45] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, USA, 1–14.

[46] Qiuping Wang, Jinhong Li, Patrick PC Lee, Guangliang Zhao, Chao Shi, and Lilong Huang. 2021. In Search of Optimal Data Placement for Eliminating Write Amplification in Log-Structured Storage. *arXiv e-prints* 1, 1 (2021), arXiv–2104.

[47] Shihao Wang, Dajiang Zhou, Xushen Han, and Takeshi Yoshimura. 2017. Chain-NN: An energy-efficient 1D chain architecture for accelerating deep convolutional neural networks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, IEEE, USA, 1032–1037.

[48] Baoxun Xu, Xiufeng Guo, Yunming Ye, and Jiefeng Cheng. 2012. An Improved Random Forest Classifier for Text Categorization. *JCP* 7, 12 (2012), 2913–2920.

[49] Gala Yadgar, MOSHE Gabel, Shehbaz Jaffer, and Bianca Schroeder. 2021. SSD-based Workload Characteristics and Their Performance Implications. *ACM Transactions on Storage (TOS)* 17, 1 (2021), 1–26.

[50] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. 2017. AutoStream: automatic stream management for multi-streamed SSDs. In *Proceedings of the 10th ACM International Systems and Storage Conference*. ACM, USA, 1–11.

[51] Jing Yang, Shuyi Pei, and Qing Yang. 2019. WARCIP: Write amplification reduction by clustering I/O pages. In *Proceedings of the 12th ACM International Conference on Systems and Storage*. ACM, USA, 155–166.

[52] Kielan Yarrow. 2010. Temporal dilation: the chronostasis illusion and spatial attention. *Attention and time* 1, 1 (2010), 163–176.

[53] Zijun Zhang. 2018. Improved adam optimizer for deep neural networks. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*. IEEE, IEEE, USA, 1–2.