UNIVERSITY OF CALIFORNIA
SANTA CRUZ

# IMPROVING PERFORMANCE OF SOLID-STATE DRIVES USING MACHINE LEARNING

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

**Chandranil Chakraborttii**

June 2021

The Dissertation of Chandranil Chakraborttii
is approved:

Professor Heiner Litz, Chair

Professor Ethan Miller

Professor Yang Liu

Quentin Williams
Vice Provost and Dean of Graduate Studies

# Table of Contents

**Abstract** vii

**List of Figures** ix

**List of Tables** xii

**1 Introduction** 1

**2 Background** 9
- 2.1 Flash Memory . . . 9
  - 2.1.1 Garbage Collection and Write Amplification in Flash . . . 13
- 2.2 Prefetching . . . 16
- 2.3 Machine Learning Techniques . . . 18
  - 2.3.1 Oversampling and Boosting . . . 19
  - 2.3.2 Isolation Forests . . . 21
  - 2.3.3 Long Short Term Memory Networks . . . 22
  - 2.3.4 Auto Encoders . . . 26
  - 2.3.5 Embeddings . . . 27
  - 2.3.6 Temporal Convolutional Networks . . . 28

**3 SSD Failure Prediction** 31
- 3.1 Introduction . . . 31
- 3.2 Research Questions (RQ) . . . 35
  - 3.2.1 RQ A: Which SSD telemetry features contribute to SSD failures? . . . 35
  - 3.2.2 RQ B: Can we predict SSD failures by training only on healthy drives (one class training)? . . . 35
  - 3.2.3 RQ C: How does the performance of one-class model training compare with state-of-the-art techniques? . . . 36
  - 3.2.4 RQ D: Can we interpret SSD failures? . . . 36
  - 3.2.5 RQ E: Can one class models be used to predict unseen SSD failures? . . . 36
- 3.3 Research Contributions . . . 37
  - 3.3.1 Accurate Prediction of SSD Failures . . . 37

3.3.2 Predicting unseen failures . . . 42
3.3.3 Interpreting SSD Failures . . . 44
3.4 Methodology . . . 45
3.4.1 Data Preprocessing . . . 46
3.4.2 Feature Selection . . . 47
3.4.3 Oversampling and Boosting . . . 49
3.4.4 1-Class ML Models . . . 50
3.4.5 Deployed System . . . 51
3.5 Results . . . 52
3.5.1 Oversampling and Boosting . . . 52
3.5.2 Accurate Prediction of SSD Failures . . . 56
3.5.3 Adaptivity to Unseen Failures . . . 59
3.5.4 Interpreting SSD Failures . . . 60
3.5.5 Sensitivity Studies . . . 62
3.6 Discussion . . . 64
3.6.1 1-Class Isolation Forest . . . 64
3.6.2 1-Class Autoencoder . . . 65
3.7 Conclusion . . . 66
3.8 Publications . . . 67

**4 Neural Network based Prefetching** **68**
4.1 Introduction . . . 68
4.2 Research Questions . . . 72
4.2.1 RQ A: Can sequence-to-sequence deep learning models learn the IO access patterns in real-world applications? . . . 72
4.2.2 RQ B: Can the neural network address timeliness by predicting multiple accesses ahead of time? . . . 72
4.2.3 RQ C: How does the performance of the neural network-based prefetcher compare with state of the art? . . . 73
4.2.4 RQ D: Can we use the learned IO access patterns to predict IO accesses in new, unseen workloads? . . . 73
4.2.5 Neural Network based Prefetching . . . 74
4.3 Problem Statement . . . 75
4.4 Proposed Prefetching Technique . . . 77
4.4.1 Data Preparation for Reducing the Output Label Space . . . 77
4.4.2 Model Architecture . . . 78
4.4.3 Timeliness . . . 79
4.4.4 Address Mapping Learning . . . 80
4.5 Methodology and Experimental Setup . . . 81
4.5.1 Model Training . . . 81
4.5.2 Prefetcher Simulation Environment . . . 83
4.5.3 Baselines . . . 84
4.6 Results . . . 84

4.6.1 Prefetcher Accuracy, Precision and Recall . . . 84
4.6.2 Impact of Cache Size, Look-Back, and Predict-Ahead . . . 85
4.6.3 Evaluation of Address Mapping Learning . . . 88
4.7 Conclusion . . . 89
4.8 Publications . . . 90

**5 Reducing Write Amplification in SSDs using Machine Learning 91**
5.1 Introduction . . . 91
5.2 Research Questions . . . 94
5.2.1 RQ A: Can sequence-to-sequence deep learning models learn the death-time patterns of logical block addresses in real-world applications? . . . 94
5.2.2 RQ B: Can we design a data placement policy for optimizing GC overhead, having perfect knowledge of future death-times? . . . 95
5.2.3 RQ C: How does the performance of our machine learning-based data placement policy (ML-DT) compare with state-of-the-art techniques? . . . 95
5.2.4 RQ D: Can we use the learned death-time patterns to predict IO death time patterns in new, unseen workloads? . . . 95
5.3 Prior Work on reducing Write Amplification (WA) . . . 96
5.3.1 Write Amplification Problem . . . 96
5.3.2 Hot-Cold Separation . . . 97
5.3.3 Frequency-based approaches . . . 97
5.4 Death-time Technique . . . 98
5.4.1 Death-Time Analysis . . . 100
5.4.2 Learning Death-Time Patterns . . . 102
5.4.3 *ML-DT* Flash Translation Layer . . . 106
5.5 Implementation . . . 108
5.5.1 Datasets and Data Preparation . . . 108
5.5.2 Machine Learning Models . . . 109
5.5.3 FTL Simulator . . . 110
5.5.4 Mapping Learning . . . 110
5.6 Results . . . 111
5.6.1 Evaluation of ML models . . . 112
5.6.2 Guaranteeing no GC overhead with *Oracle-DT* . . . 114
5.6.3 Comparison with baselines . . . 117
5.6.4 Impact of number of open blocks . . . 121
5.6.5 Sensitivity Study on Open Blocks . . . 122
5.6.6 Evaluation of Mapping Learning . . . 124
5.7 Conclusion . . . 126
5.8 Publications . . . 127

**6 Related Work 128**
6.1 SSD Failure Prediction . . . 128
6.2 Neural Network based Prefetching . . . 132

6.3 Garbage Collection Optimization using Machine Learning . . . . . 137

**7 Conclusion** **142**

**8 Acknowledgements** **144**

**BIBLIOGRAPHY** **146**

## Abstract

Improving Performance of Solid-State Drives using Machine Learning

by

Chandranil Chakraborttii

Flash-based storage drives such as solid-state disks are replacing traditional spinning disk drives for an increasing number of applications. User interfacing cloud-based applications benefit from the low, sub-millisecond access latency of solid-state drives (SSDs). Virtually all smartphones are using flash memory as their storage media due to features such as low power consumption, larger storage density, small footprint and shock resistance. SSDs provide faster boot times, higher read and write bandwidth as well as improved durability. Nevertheless, flash-based storage devices show several disadvantages. Technology scaling, 3D integration as well as multi-level bit cells have continuously increased storage density and capacity, however, this has also reduced the reliability of flash. Flash memory also suffer from overheads such as garbage collection, which can reduce write bandwidth and introduce high tail latency. Furthermore, while NAND flash devices provide significantly lower latency than spinning disks, flash has still orders of magnitude higher latency than DRAM.

This work leverages machine learning techniques to improve the performance of flash based storage systems. This improvement reflects in three major directions - improving response time, reliability and lifetime of flash based storage devices. For improving response time, we leverage sequence-to-sequence machine learning techniques to learn the spatial IO

access patterns thereby improving prefetching performance. To achieve high performance, we address the challenges of prefetching in very large sparse address spaces, as well as prefetching in a timely manner by predicting ahead of time. To improve reliability, we propose an approach of automatically predicting and interpreting future drive failures. Finally, we present a machine learning based approach for reducing the number of rewrites required to store data in log structured file systems via death-time prediction of logical block addresses. We leverage the predicted death-times in designing *ML-DT* , a near-optimal data placement technique that minimizes the number of extra writes required to store data in log structured storage systems thereby improving device lifetime.

# List of Figures


1.1 Annual Size of Global Datasphere . . . . . 2

1.2 Investment in Global Data Center Market . . . . . 4

2.1 Representation of flash memory cell . . . . . 10

2.2 General architecture of a Flash SSD . . . . . 11

2.3 Overprovisioning in SSDs . . . . . 14

2.4 Lifecycle of an SSD block . . . . . 15

2.5 Demonstration of Prefetching in SSDs . . . . . 18

2.6 Demonstration of Oversampling process . . . . . 19

2.7 Demonstration of Boosting . . . . . 21

2.8 General architecture of vanilla ANN . . . . . 22

2.9 General architecture of vanilla RNN . . . . . 23

2.10 General architecture of an LSTM cell . . . . . 24

2.11 Internal design of Auto Encoders . . . . . 26

2.12 Demonstration of Embeddings [1] . . . . . 27

2.13 General architecture of temporal convolutional networks . . . . . 29

3.1 PCA with all 21 Features . . . . . 39
3.2 PCA with Top 9 Selected Features . . . . . 40
3.3 Illustration of data collection process . . . . . 41
3.4 Autoencoder Design . . . . . 43
3.5 Block diagram of the Deployed System . . . . . 52
3.6 Predicting 1 week ahead (N=1) . . . . . 54
3.7 Predicting 2 weeks ahead (N=2) . . . . . 54
3.8 Predicting 3 weeks ahead (N=3) . . . . . 55
3.9 Predicting 4 weeks ahead (N=4) . . . . . 55
3.10 Comparison of Different Machine Learning Approaches for Prediction of Healthy (H) and Failed (F) drives . . . . . 55
3.11 ROC AUC Score comparison of the five evaluated Machine Learning Techniques 57
3.12 Accuracy, Precision, Recall and Fscore for the five evaluated Machine Learning Techniques . . . . . 58
3.13 Predicting unseen failures . . . . . 60
3.14 Interpreting SSD Failure Reasons . . . . . 60
3.15 ROC AUC score when predicting up to 4 days ahead . . . . . 62
3.16 Impact of Feature Selection Techniques . . . . . 65

4.1 Prefetching motivation . . . . . 69
4.2 Model architecture . . . . . 79
4.3 Block diagram of the Address Mapping Learning process . . . . . 80

4.4 Block diagram of the evaluation process using our simulator . . . . . 83

5.1 LBA write frequency distribution for VDI . . . . . 92

5.2 Baseline vs. Frequency vs. Oracle-DT Policy . . . . . 100

5.3 Death-times varying widely (sample) . . . . . 101

5.4 Model Architecture . . . . . 104

5.5 Mapping Learning Architecture . . . . . 110

5.6 Comparison of ML Techniques for VDI traces (N Class) . . . . . 114

5.7 Comparison of ML Techniques for RocksDB and TPC-H traces (N Class) . . . 114

5.8 Block usage per trace with *Oracle-DT* . . . . . 115

5.9 Block usage over time with *Oracle-DT* . . . . . 116

5.10 FTL comparison with baselines (VDI traces) . . . . . 117

5.11 FTL comparison with baselines (RocksDB and TPC-H traces) . . . . . 118

5.12 Distribution of writes comparison with baselines (VDI traces) . . . . . 119

5.13 Distribution of writes comparison with baselines (RocksDB and TPC-H traces) 119

5.14 Impact of number of open blocks (VDI trace) . . . . . 122

5.15 Impact of number of open blocks(RocksDB trace) . . . . . 123

5.16 Impact of number of open blocks(TPC-H trace) . . . . . 123

5.17 Sensitivity Study (VDI traces) . . . . . 124

5.18 Sensitivity Study (RocksDB and TPC-H) . . . . . 124

5.19 Mapping learning results using different Source (S) and Recipient (R) workloads 126

# List of Tables


3.1 All 21 features collected . . . . . 46

3.2 Top features selected . . . . . 49

3.3 Results from SMOTEBoost and RUSBoost . . . . . 54

3.4 Model training time (N = 1) . . . . . 66

4.1 Dataset Description . . . . . 82

4.2 Performance comparison of Our proposed prefetcher against baselines . . . . . 86

4.3 Impact of different predict values on our prefetcher performance . . . . . 87

4.4 Impact of cache size on the accuracy of our and two baseline prefetchers . . . . 87

4.5 Performance of Address Mapping Learning (AML) . . . . . 89

5.1 Comparison of machine learning approaches for death-time range prediction. . 112

# Chapter 1

# Introduction

Flash memory is a persistent storage technology. Flash memory is used as a means of persistent storage in mobile devices and as mass storage for cloud or general computing systems. 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 100 microsecond read latency. The two most common types of flash storage include NAND flash and NOR flash. NAND flash provides higher write and erase speeds than NOR flash; hence, it is used more commonly than NOR flash.

Prior to the advent of SSDs, hard disk drives (HDDs) were the most popular secondary storage choice. A conventional HDD is a storage device with spinning disks and a head to read the data, making it susceptible to mechanical failures. On the other hand, SSDs can store data within a chip without requiring any moving parts and hence is less prone to mechanical failures. Flash-based solid-state drives (SSDs) offer superior read/write performance and have higher ability to withstand harsh conditions such as shocks, vibrations, and temperature fluc-

tuations, compared to the traditional hard-disk drives (HDDs). For example, the slowest SSD can outperform the fastest HDD in terms of the number of read/write operations. Flash storage devices can perform read operations as fast as dynamic random-access memory (DRAM) and can perform write operations hundreds of times faster than HDDs. For example, a computing system running Windows 10 operating system (OS) in SSD can start a usable desktop in less than 10 seconds [2]. SSDs are more economical in terms of power usage as they are tuned to go to sleep and wake up quickly and reliably. They can also withstand vibrations, making them durable and ideal for use within portable devices [3]. Although the storage cost per bit is higher for SSDs, they are faster, cooler, and more silent than traditional HDDs. Additionally, SSDs have a lower Annual Replacement Rate (ARR) than their hard-disk drive counterparts [4].

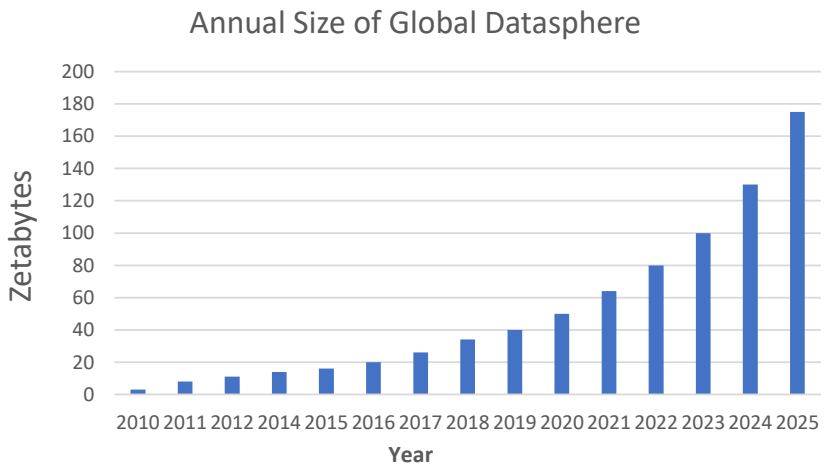


Figure 1.1: Annual Size of Global Datasphere

The recent decade has seen a major shift towards NVME flash-based storage from traditional HDDs, especially for data centers supporting cloud services. Studies [2] have shown

an 5-fold increase in the use of SSDs for cloud systems in the last decade alone. The lower access latency combined with lower power consumption and higher bandwidth render make flash-based solid-state drives (SSDs) a higher-performance alternative to hard disk drives in the cloud and mobile environments.

Recent advances in flash technologies such as transistor scaling and multiple layered cells (MLC) have increased the storage density and capacity, and modern SSDs can offer a storage capacity of tens of terabytes (TB). In addition to the huge volume, SSD's high throughput meets the requirements for enterprise applications by virtue of its massively parallel architecture at the back-end, including multiple channels, multiple dies per channel, multiple logic units (LUNs) per die, and multiple planes per LUN. Parallel I/O operations can be performed among independent planes giving rise to a great potential for very high I/O performance.

Along with the rise of the popularity of SSDs, the recent decade has also seen an explosion in storage requirements for data. From Figure 1.1, we can see a nearly exponential increase in storage requirements since 2010. Future storage requirements are expected to increase further to 175 zeta bytes (ZB) in the year 2025 [5]. Data centers are high-performance computers that store and process data. Every organization, which requires handling of user data uses data centers. Data centers usually consist of racks and cabinets containing computer hardware and batteries with backup generators to guard against power breakdown. They also maintain high-grade cooling systems to keep the computers from overheating. Data in the cloud are stored in data centers located in remote locations, and the users access them via the internet. Data centers store massive amounts of data worldwide and are projected to store most user data

in the future. They allow fast, reliable, and secure access to data. Investments in data centers have seen a continuous year upon year increase, as can be seen from Figure 1.2.

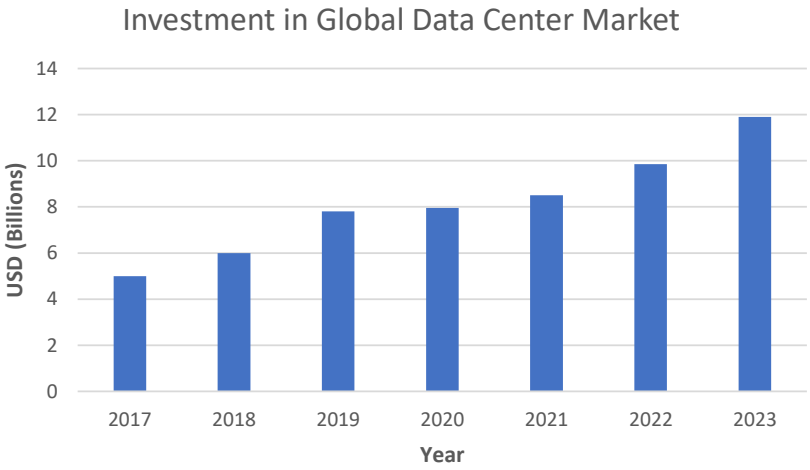


Figure 1.2: Investment in Global Data Center Market

Modern data centers offer a variety of services at different pricing levels with performance depending on cost. For example, training AlexNet [6] on a virtual machine (VM) with no GPU support will take significantly longer time for training than a system with GPU support. In modern data centers, just optimizing for performance is not enough as we need to take into account the cost of the hardware and operational costs. Hence the most important metric for optimization is total cost of ownership (TCO)/Performance. The goal is to maximize the return on investment and performance given the cost or hardware constraints. Requirements of such a system include high reliability, efficient resource utilization, and high endurance. High reliability requires the system components to be robust to unforeseen circumstances (such as power failures or voltage fluctuations) and continue delivering high performance. In contrast,

high resource utilization ensures that we get the maximum performance out of the system. High endurance deems the system can run without replacement or repair for a significant period to recuperate the investment. (5000-50000 write/erase cycles for flash drives).

Flash drives are very attractive for use within mass storage systems. However, they have some limitations. The flash memory chips have a limited lifetime as the number of write operations per memory cell is limited, and the performance of flash chips degrades over time due to wear out. To improve the lifetime of flash, the system firmware needs to distribute the writes as evenly as possible, referred to as wear leveling. Thus, it is challenging to guarantee a 'stable' flash memory-based system that is as highly reliable as a disk-based system. Although SSDs have lower Annual Failure Rates (AFR) than disk-based alternatives [4], they are more susceptible to bad sectors and block errors. With the recent advances in flash technology, the flash chip capacity has increased manifolds over the past decade, but it also resulted in a decrease in the reliability of flash memory [7]. Moreover, SSD-related failures are considered more critical than HDD-related failures. According to a study, SSD-related failures resulted in the replacement of 79% of the drives in the cloud, compared to 11% for HDD-related failure [8]. Hence, SSDs remain vulnerable to sudden device failures resulting in data loss or application crashes, which is undesirable.

Another limitation of Solid-state drives (SSDs) is the access latency or the bandwidth bottleneck. Although SSDs deliver significantly higher speeds than HDDs, SSDs still remain a performance bottleneck of computing systems [9]. Processors and DRAM technologies support three orders of magnitude lower access latency. As a result, the system's performance is often

bottlenecked by the speed of SSDs, resulting in poor utilization of resources as higher levels of memory have to wait for the slower SSDs to respond to the request.

SSDs also suffer from the problem of write-amplification due to a lack of support for in-place data 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 [10], 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 and endurance of the SSDs are reduced. Second, the extra GC writes introduce performance interference by delaying the regular user reads. Modern SSDs and operating systems offer a wide range of telemetry data for analysis. Utilizing I/O access tracing in hardware and software enables the collection of large, clean, and automatically labeled datasets that can fuel powerful machine learning models. This work attempts to alleviate the problems mentioned above using machine learning techniques.

To improve the reliability of SSDs, we introduced a novel machine learning (ML)

based technique to predict which drives are likely to fail within a specific time range in the future. In order to train the machine learning model to predict failures, we first collect the telemetry information from more than 30,000 SSDs running live applications in Google datacenters over a period of 6 years. We show that our proposed approaches can effectively predict failures in drives with high accuracy and recall, thereby capturing all predicted failures. Furthermore, we leverage autoencoders to interpret the reasons for why the machine learning model flags a drive to fail. We discuss the project in more detail in Chapter 3.

For improving resource utilization, we propose a deep neural network (DNN) based prefetching to reduce response time. Prefetching predicts future block accesses and preloads them into the main memory ahead of time. In Chapter 4, we discuss the challenges of prefetching in SSDs, explain why prior approaches fail to achieve high accuracy, and present a neural network-based prefetching approach that significantly outperforms the state-of-the-art. To achieve high performance, we address the challenges of prefetching in very large sparse address spaces, as well as prefetching in a timely manner by predicting ahead of time. We collect I/O trace files from several real-world applications running on cloud servers and show that our proposed approach consistently outperforms the existing stride prefetchers by up to 800$\times$ and prior prefetching approaches based on Markov chains by up to 8$\times$. Furthermore, we propose an address mapping learning technique to demonstrate the applicability of our approach to previously unseen SSD workloads.

In order to improve the endurance of SSDs, we present a machine learning-based approach for reducing write amplification in log-structured file systems via death-time pre-

diction of logical block addresses. We define the death-time of a data element (LBA) as the number of write I/O accesses before which a given 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. 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.

# Chapter 2

# Background

In this chapter, we first provide background on flash memory and provide details on the garbage collection process and write amplification within SSDs. We then describe prefetching in systems and finally discuss the various machine learning models used in this thesis.

## 2.1 Flash Memory

Flash memory was invented by Fujio Masuoka in 1980 and later commercialized by Toshiba in 1987. SanDisk Corporation (formerly SunDisk) followed Toshiba and entered the commercial market in 1989. The first commercial flash drive shipped by SanDisk was a 20MB solid-state drive (SSD) plugged in using a PC card and retailed for over $1000 and was used by IBM in their Thinkpad laptop series [11]. In 1995, some of the mission-critical applications in military and aerospace industries started using flash-based SSDs, as they offered superior read/write performance and the ability to withstand harsh conditions such as shocks, vibrations, and temperature fluctuations. The latest generation of SSDs introduced by GigaByte in 2019

can perform sequential reads at the rate of 15000 megabytes per second (MBps) and sequential writes at the rate of 15200 MBps [11].

In flash memory, the data is written serially in an entire chip or large block of contiguous data bytes [12]. Flash memory is popularly used as a means of persistent storage in mobile devices and as mass storage for cloud or general computing systems. The two most common types of flash storage include NAND flash and NOR flash. NOR flash was the first one to be developed, although NAND flash is used more commonly as NAND flash provides higher write and erase speeds compared to NOR flash. Typically in an enterprise environment, an array of NAND flash memory is used as secondary storage of user data. In contrast to traditional hard disk drives, which use rotating disks, modern SSDs use flash chips to store data.

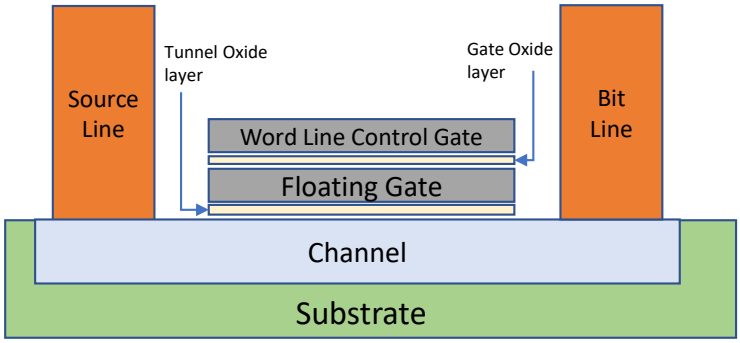


Figure 2.1: Representation of flash memory cell

The Flash memory cell (FMC) is the fundamental building block of flash memory. It follows a similar design to standard MOSFET (metal-oxide-semiconductor-field-effect transistor), but unlike a MOSFET, the FMC has two gates. The cells represent an electric switch where

the two gates, a floating gate (FG) and a control gate (CG), control the amount of current flowing between the two terminals (source and drain). FG and CG are used to control the voltage in the cell resembling the logical states 0 and 1. The internal architecture of a flash memory cell is shown in Figure 2.1. Hence, NAND flash drives or SSDs are non-volatile memory devices that store individual bits on floating gate transistors. Floating gate transistors are arranged in large bit cell arrays, increasing not only the storage capacity but also the access latency. Flash cells suffer from limited endurance and frequent bit errors, which are exacerbated by transistor scaling and the introduction of techniques such as multi-level cells [13]. To ensure data integrity, multiple reads using different reference voltages need to be performed, and the controller needs to perform error detection and correction as part of each read, increasing the read latency. As a result, the I/O access latency of SSDs (100us) is three orders of magnitude higher than the latency of reading DRAM (100ns).

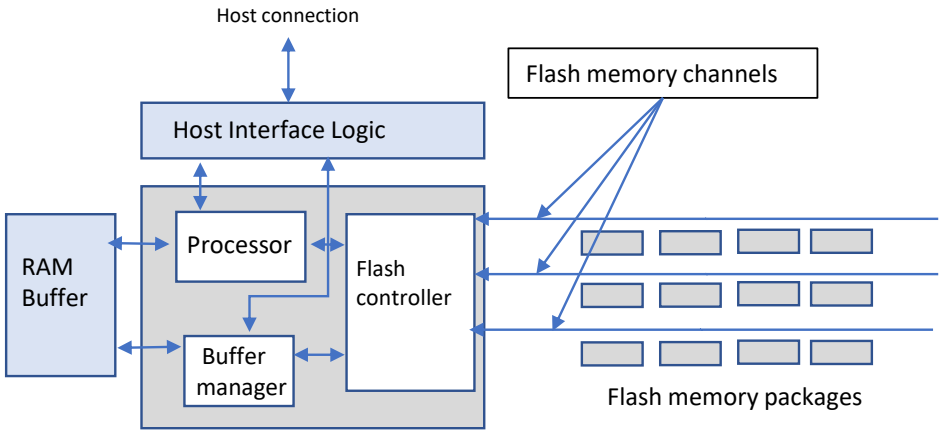


Figure 2.2: General architecture of a Flash SSD

The general architecture of an SSD is presented in Figure 2.2. The SSD contains a NAND flash memory array, an embedded CPU (optional), Flash Memory Controllers (FMCs), and Dynamic Random Access Memory (DRAM). The host interface controller supports a specific bus interface protocol, such as Serial Advanced Technology Attachment(SATA), Statistical Analysis System (SAS), or Peripheral Component Interconnect (PCI-e) [14]. Lately, NVMe (Non-Volatile Memory Express) has come forth as a new interface providing additional features such as lower latency and scalable bandwidth compared to other storage interfaces. The NVMe interface allows the host resources to be utilized for memory and computations for flash management, such as address mapping and wear leveling.

Flash memory is composed of multiple blocks, which are further composed of multiple pages. A page is a unit of a read operation, while a block is a unit of erase operation. In contrast to HDDs, SSD's performance scales with the number of NAND flash chips on a device [15]. The flash memory controllers transfer data between DRAM and NAND flash chips using Direct Memory Access (DMA). They are also responsible for maintaining data integrity using Error Correction Codes (ECC) such as BCH (Bose Chaudhuri Hocquenghem code) [16], or Reed Solomon [17]. An FMC is also responsible for multi-way interleaving over multiple NAND flash chips on a shared I/O bus using multiple chip-enabled signals [15]. Multi-channel interleaving can also be used with independent channels and FMCs. This combination of multi-channel interleaving over multiple NAND flash chips deployment improves the performance of both sequential and random accesses. Dedicated hardware for FMC is also used for further improving the performance and power efficiency. It can also be used for implementing an

application-specific instruction set processor which can support a wide variety of NAND flash memory commands. The Embedded CPUs, in combination with SRAM, are responsible for maintaining the execution environment for supporting the Flash Translation Layer (FTL) [18], the flash management firmware. The FTL interprets the host commands and maintains a mapping table [15] as it translates a logical block address (LBA) to a physical address on the NAND flash memory chips. Usually, a 32 bit RISC (Reduced instruction set computer) processor is used, but based on the requirements, multiple embedded CPUs can be integrated to support multiple host requests simultaneously. The DRAM is responsible for storing FTL metadata and user data temporarily. The DRAM is typically operated at a high clock frequency as it is the target of data transfers from both the FMCs and the host interface [15].

### 2.1.1 Garbage Collection and Write Amplification in Flash

Garbage collection is method of automatic memory management in log structured storage systems [19]. An essential property of flash is that in order to update the data already written, it needs to erase the old data first and then rewrite the whole data again as erase operations happen at a block-level. Flash memory is typically divided into blocks, which are further divided into pages. Data can be written at the page level. However, erases or updates can only be performed at the granularity of blocks. Hence, to free-up, the space taken up by stale data, all the valid pages within the block must be copied to empty pages of another free block first. Thereafter, we can perform the erasure at block level to prepare it for new valid incoming data [20].

If the data within the pages of a block are invalidated (also called stale pages), only the

pages with valid data in that block are read and rewritten into another previously erased or empty block. This frees up the pages by erasing all stale data to make space for new incoming data. This process of copying live data to new blocks and reclaiming old blocks is called Garbage Collection (GC). GC is a cardinal process with all SSDs, but it can be designed in various ways, which can impact the overall SSD performance and lifetime.

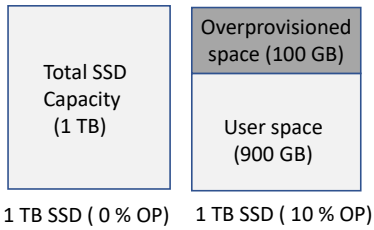


Figure 2.3: Overprovisioning in SSDs

To facilitate copying of valid data to new free blocks, the SSD usually reserves a portion of the memory as spare, also known as the over-provisioned capacity (OP). It is defined as the ratio between the reserved capacity (physical capacity - user capacity) vs. the physical capacity of the device (Figure 2.3). Increasing the OP ratio increases the speed of the device by reducing the number of writes needed in GC. It also increases device lifetime by spreading wear-leveling over more physical blocks. However, it comes at the cost of reduced storage capacity of the device.

Over Provisioning (OP) = (Total SSD capacity - User capacity ) / Total SSD capacity

The garbage collection algorithms are responsible for copying all valid data into a free

space and erasing the original invalid data. The lifecycle of a block within an SSD is shown in Figure 2.4. 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 as a closed-block. As pages are overwritten, closed-blocks contain an increasing number of invalid pages. Finally, the GC mechanism cleans a block by moving all the valid pages to another block before erasing it, so that it can be added back to the list of free-blocks. Garbage collection is performed when there are no free blocks available or when the number of free blocks in the device is lower than a predefined threshold value. The steps followed during a typical garbage collection process is listed below are:

- The virtual blocks meeting the conditions are shortlisted for deletion.
- The valid physical pages are copied into a block with free pages available.
- The entire physical block is freed to make space for new data

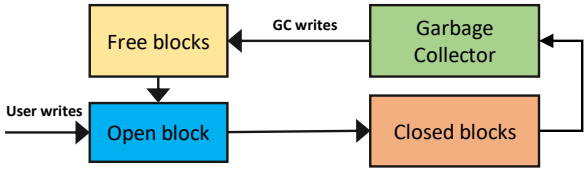


Figure 2.4: Lifecycle of an SSD block

Garbage Collection is performed at the Flash Translation Layer [18] for NAND Flash memory devices. The Flash translation layer (FTL) is a software/hardware mechanism inside NAND flash memory that maps logical blocks in an SSD to physical blocks. Since the data within the logical blocks may span multiple physical blocks, the garbage collection may erase more than one physical block. Garbage collection induces a performance overhead on flash

devices as more number of writes are used to store the data within the flash memory. This overhead is measured using a metric referred to as called Write Amplification (WA), defined as the ratio of the physical pages required to be programmed inside the device for storing logical block data from the host.

WA = Num pages programmed inside SSD / Num page updates issued by host

Write amplification due to internal copying directly reduces application throughput and the lifetime of the device due to increased writes and wear leveling. The impact of garbage collection on write amplification is influenced by the level of over-provisioning and the choice of reclaiming policy. There exist two common techniques for reclaiming victim blocks for GC. Greedy mechanisms [21] choose the block with the lowest number of valid blocks. Cost-benefit mechanisms [22] 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 as a closed block. As pages are overwritten, closed blocks contain an increasing number of invalid pages. Finally, the GC mechanism cleans a block by moving all the valid pages to another block before erasing it so that it can be added back to the list of free blocks.

## 2.2 Prefetching

A computer system typically consists of several levels of memory running at different performance (registers, cache, DRAM, SSD, SATA, etc.) with the goal of reducing the overall

access time. A memory hierarchy [23] has been developed which determines the memory organization of the system. Prefetching in systems is the process of preloading data from a slow storage device into faster memory, generally DRAM, to decrease the overall read latency. Accurate and timely prefetching can effectively reduce the performance gap between different levels of memory.

There are three important metrics used to compare prefetchers, including coverage, accuracy, and timeliness of prefetchers [9]. Coverage is the ratio of the number of SSD reads that can be prefetched to the total number of SSD reads. Accuracy is the ratio of the number of data blocks being prefetched to the number of prefetched data blocks that were actually requested by the application. Timeliness requires data blocks to be prefetched sufficiently ahead of time so that the data is present in DRAM whenever the application performs the read request. If the prefetched data blocks are not available when they are needed, the application is required to stall, rendering prefetching ineffective. Furthermore, if the data is prefetched too early, it may not be available anymore when it is actually needed due to the eviction from the capacity-limited cache. Inaccurate prefetches that read in unneeded data are harmful as they waste I/O bandwidth and DRAM capacity. If prefetching is performed too conservatively, coverage is low, and the overall performance gains are limited. Hence, the ideal prefetcher has high coverage, high accuracy and executes timely prefetches so that the data is fetched exactly when needed. A basic prefetching mechanism is shown in Figure 2.5.

The SSD prefetcher (P) is responsible for predicting candidate data blocks (C) to prefetch from the SSD (S) into a fast cache (DRAM) buffer (B) of size. The cache eviction

policy (E) is responsible for evicting the data blocks from B in order to make space for new incoming data. Candidates x2 and x3, however, were present in the cache when requested, and hence, resulted in a cache hit. In this example, at times, P determines candidates x1, x2, x3, and x4 for prefetching, but the actual data requested at time t is x1, x2, and x3. Here, x1 was prefetched too early while x4 was inaccurately prefetched, resulting in cache miss in both cases.

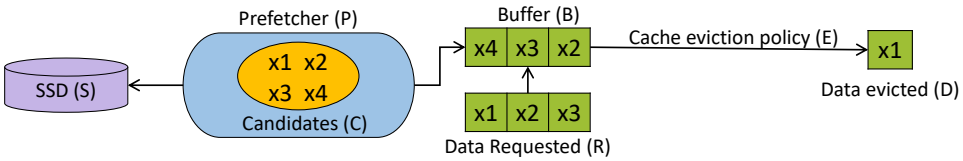


Figure 2.5: Demonstration of Prefetching in SSDs

## 2.3 Machine Learning Techniques

Machine learning is the science of developing systems that can learn automatically from data to make decisions or inferences without being explicitly programming the system to perform the task [24]. There are several broad areas of machine learning including supervised learning [25], unsupervised learning [26], semi-supervised learning [27], and reinforcement learning [28]. One of the goals of supervised learning is to build predictive models that can learn from labeled examples, where the labels represent the category to which a particular example belongs, and upon training the model, it can be used to make predictions on unseen examples of data that the model has not seen before. The goal of unsupervised learning is to draw inference from the data in the absence of labels, for example, is to find patterns or groupings within the unlabeled data set [29]. Semi-supervised learning [30] approaches utilize a large

number of unlabeled examples together with a relatively small number of labeled examples of data to train predictive models. In this thesis, we use supervised, semi-supervised, and unsupervised learning approaches to improve the performance of storage systems. Next, we describe the specific machine learning techniques that we used in this thesis.

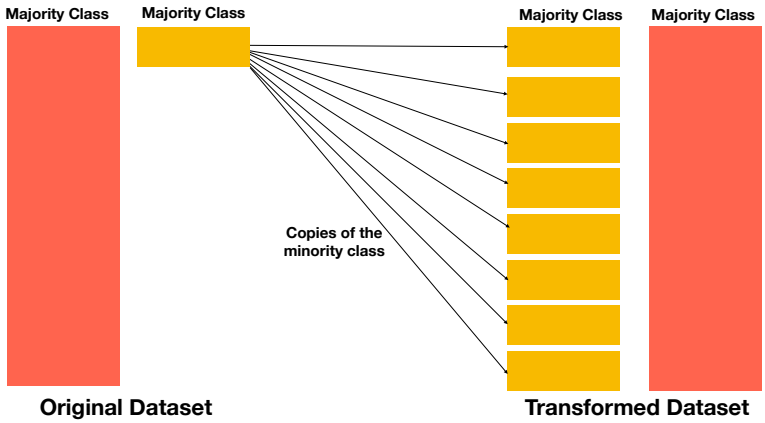


Figure 2.6: Demonstration of Oversampling process

### 2.3.1 Oversampling and Boosting

One of the problems that is sometimes encountered while building predictive models is that there are a very few examples of one category (also known as the minority class) and relatively large number of examples for the other category (also known as the majority class) present in the labeled or training dataset. This often leads to a model that overfits to the examples of majority class. One of the ways to deal with this class imbalance problem is to use oversampling techniques that artificially add samples to the minority classes using various methods (such as SMOTE [31]) to make the dataset more balanced and improve learning (Figure 2.6). The performance of Machine Learning models can also be improved by combining

several learners, a process called Boosting [32]. The basic principle behind the working of the boosting algorithm [33] is to generate multiple weak learners and combine their predictions to form one strong rule. These weak rules are generated by applying base ML algorithms on different distributions of the data set. These algorithms generate weak rules for each iteration. After multiple iterations, the weak learners are combined to form a strong learner that will predict a more accurate outcome.

We used SMOTEBoost [34] and RUSBoost [35] algorithms for classifying the observations. It employs a mix of SMOTE [31] and the standard boosting procedure AdaBoost [36] to model the minority class. It works by giving the model not just with minority class data points that were misclassified in the previous boosting iteration, but also with broader representation of those instances (achieved by SMOTE). SMOTE also increases the diversity in the minority class samples in each iteration by creating synthetic samples. The SMOTE (Synthetic Minority Over-Sampling Technique) function takes feature vectors with dimension(v, n) and the target class with dimension(v,1) as the input. It returns final features vectors with dimension (v', n) and the target class with dimension (v',1) as output SMOTE [31]. Figure 2.7 demonstrates boosting process used in machine learning. Boosting uses the performance of trees from previous iteration to assign weights to the next tree to be built. More weight is assigned to data that are difficult to separate. Models are produced sequentially one by one to update each of the weights. After all trees are generated, new information is predicted based on the weighted performance of the trees on input data.

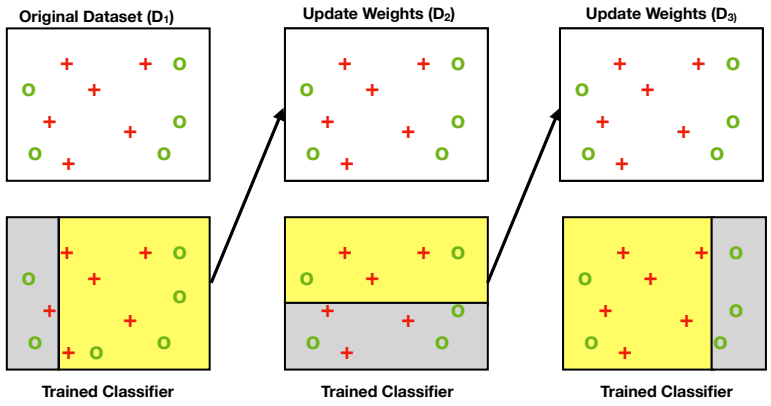


Figure 2.7: Demonstration of Boosting

### 2.3.2 Isolation Forests

Anomaly detection [25] methods are commonly used to find rare events or observations in the data which differ significantly from the majority of the data points. Anomaly detection can be performed in a supervised, semi-supervised, and unsupervised manner. It works well with imbalanced data since it makes a model using the majority class. We used unsupervised and semi-supervised approaches for detecting anomalous data points using Isolation Forests [37]. Semi-supervised approaches use only good, non-anomalous data points for training. It assumes that we only know which data points are non-anomalous, and we do not have any information on the anomalous data points. While making predictions, the model evaluates similarity between the new observations with respect to the training data and checks whether it fits the model. A supervised approach, on the other hand trains on both genuine and anomalous data points. It doesn't require labelling which can be hard and time consuming. We use the isolation forest algorithm to separate out the anomalous data points. The Isolation Forest algorithm separates data points by randomly selecting a feature, followed by randomly selecting a split

value between the maximum and minimum values of the selected feature. We build multiple decision trees so that the trees isolate the observations in their leaves. Ideally, each leaf of the tree isolates exactly one observation from the data set. Isolation Forest can scale up to tackle huge data sizes with high-dimensionality.

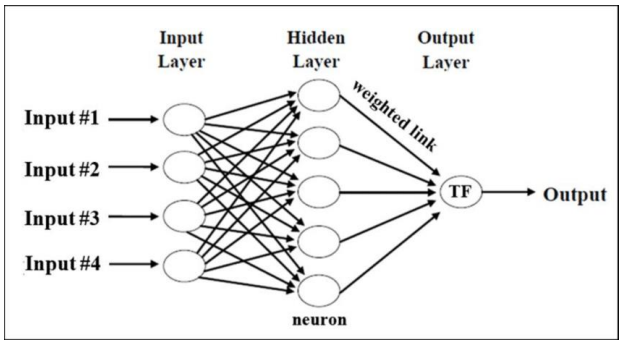


Figure 2.8: General architecture of vanilla ANN

### 2.3.3 Long Short Term Memory Networks

Artificial neural networks (ANN or NN) [38] combine machine learning algorithms for solving specific tasks. The architecture of neural networks is analogous to a synapse within our nervous system where a signal can be transmitted from one neuron to the other. The layers are connected further by links defined between neurons from one layer to the next. Neural Networks similarly consist of layers, where each layer contains many artificial neurons. A NN typically comprises multiple layers, each performing a computational task that contributes towards a solution to the task at hand. The first layer is called the input layer, which feeds the input to the NN in a numerical format. The input layer is followed by one or more layers which eventually lead to the final layer called the output layer. The output layer performs similarly

to a decision-maker, and the layers between the input and the output layer are called “hidden” layers. The hidden layers are responsible for the ‘actual learning’ of the data. The general structure of a vanilla ANN is shown in Figure 2.8. Deep learning, also called deep structured learning or hierarchical learning, is part of a broader family of ML techniques based on ANN [39].

The data in an ANN ”propagates” by way of transformation sequentially starting from the input layer, through the hidden layers, eventually ending at the output layer. Such a network is called a ’feedforward neural network’. Recurrent neural networks (RNN) [40], work similar to vanilla NN, with the difference that RNNs utilizes a concept of memory using a different kind of link, which enables feedback. Contrary to feedforward NN, the outputs of some layers are fed back to the inputs of previous layers. This enables them to analyze sequential data while taking into account the input sequence of the data. The feedback links work in the reverse direction and help to learn abstractions based on context. The general architecture of an RNN is shown in Figure 2.9.

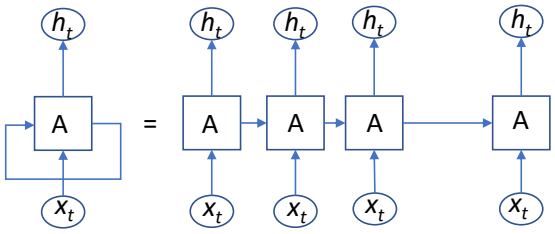


Figure 2.9: General architecture of vanilla RNN

Long short term memory (LSTM) [41] is a variant of recurrent neural network ar-

chitecture used in the field of deep learning. LSTMs are a subset of artificial neural networks, which can take time and sequence into consideration. (i.e., they have a temporal dimension). LSTMs were designed by Sepp Hochreiter and Juergen Schmidhuber to learn and recognize patterns in sequential data (for e.g., weather data [42], stock markets [43], IO accesses [9], etc.) and are capable of learning long-term dependencies in data due to the feedback connections. They can process the entire data sequence, not only just single data points as in pictures. LSTMs are popularly used in a wide variety of problems such as speech recognization, market prediction, grammar learning and protein homology detection.

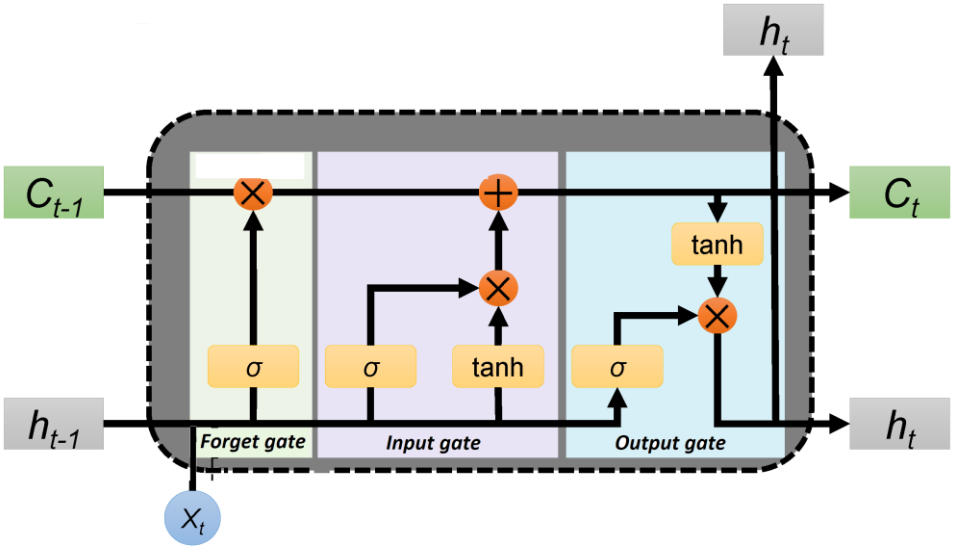


Figure 2.10: General architecture of an LSTM cell

The architecture of an LSTM cell is shown in Figure 2.10. It is composed of four layers with three logistic gates and a 'tanh' [44] layer in contrast to RNNs, which have a single neural net layer of 'tanh'. Gates are used to control the information passed through the cell. The

output is typically a value in the range between 0-1, which determines the amount of information retained from the previous layer. ( output '0' means 'reject all' and output of '1' means 'include all'). At a given time, C(t) and h(t) represents the cell state and hidden state, respectively, and the current data input is represented by x(t). The first layer in an LSTM cell is the sigmoid layer, and it takes into two inputs - h(t-1) and x(t) and generates two outputs - h(t) and C(t). It is called the forget gate as its output (between 0-1) determines the amount of information to be discarded. The output is then point-wise multiplied with the previous cell state C(t-1). The second sigmoid layer takes in two inputs - h(t-1) and x(t) is called the additional gate as it determines what new information is added to the current cell state. The layer computes a vector of the new candidate values. The point-wise multiplication decides the amount of information to be added in that particular cell state. The result is added to the output of the forget gate multiplied with the previous cell state C(t). Finally, the output is computed using a combination of sigmoid and 'tanh' layer. The sigmoid layer determines the part of the cell state be kept in the output while the 'tanh' layer transforms the output in the range (-1,1). The output is generated by point-wise multiplication to produce the output of the cell h(t).

We propose to utilize Long Short-Term Memory (LSTM) based sequence-to-sequence neural networks to learn spatial I/O access patterns of application block-level I/O traces for prefeteching. LSTMs can leverage the sequential nature of IO accesses and have a "memory" that allows the model to look at recent accesses while making a decision. The architecture is capable of capturing long-term dependencies in data and can IO sequences of different lengths. LSTMs integrate model training and representation learning together without requiring addi-

tional domain knowledge, enabling the discovery of unseen patterns in the data to improve the generalization capability of a model. We leveraged LSTMs to separate the complex and interleaved I/O streams in data and addressed the challenge of timeliness of predictions by predicting multiple I/O accesses ahead of time. This enabled us to build a neural network-based prefetcher which we discuss in Chapter 4.

### 2.3.4 Auto Encoders

Auto encoders [45] are artificial neural networks which can learn efficient representations of input data, known as codings. These codings typically represent input data in lower dimensions, making it a popular use case for dimensionality reduction [46], [47]. Auto encoders work by learning to copy their inputs to their outputs. They work by compressing the input into a latent-space representation, and then reconstructing the output from this representation. This type of network is composed of two parts:

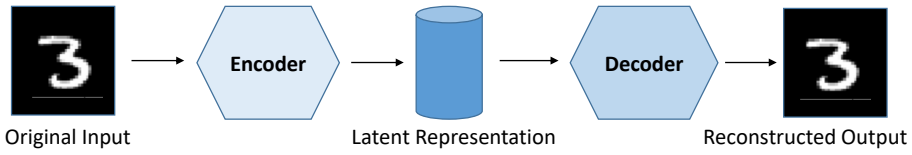


Figure 2.11: Internal design of Auto Encoders

- Encoder: This is the part of the network that compresses the input into a latent-space representation. It can be represented by an encoding function h = f(x).
- Decoder: This part aims to reconstruct the input from the latent space representation. It can be represented by a decoding function r = g(h).

For example, it is possible to limit the size of the internal representation, or you can add noise to the inputs and train the network to recover the original inputs. These constraints prevent the auto encoder from trivially copying the inputs directly to the outputs, which forces it to learn efficient ways of representing the data. In short, the encodings are byproducts of the auto encoder's attempt to learn the identity function under some constraints.

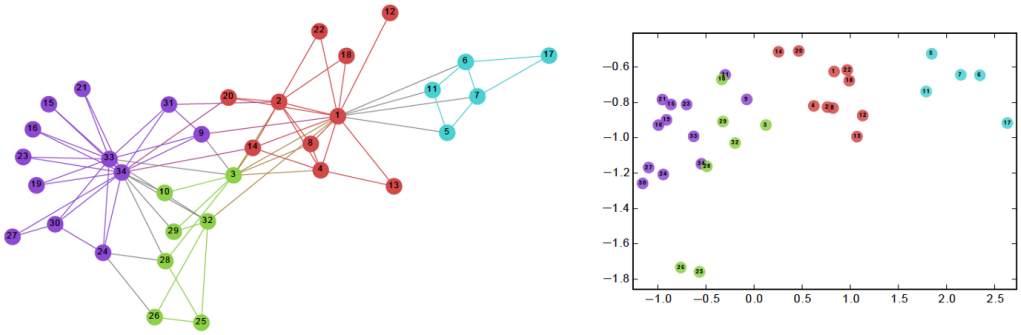


Figure 2.12: Demonstration of Embeddings [1]

### 2.3.5 Embeddings

An Embedding is a lower-dimensional space representation that can translate high dimension vectors. Embeddings are particularly useful for sparse vectors as they can represent the sparse input more effectively using less data such as a representation of words. The distribution of block address IO access follows a similar sparse pattern and can be represented by a sparse vector. 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. Embeddings can be learned from multiple models and can be reused. Popular Embeddings such as AlexNet [6], Word2Vec [48] and ImageNet

[49] are reused for various natural language processing (NLP) tasks. Embeddings are learned jointly with a neural network model with a certain task in mind (typically related to natural language processing tasks such as designing a chatbox, or document classification). The size of the vector space is specified as part of the model, such as 50, 100, or 300 dimensions. The vectors are typically initialized with small random numbers. The embedding layer is used before the first layer of a neural network and is fit in a supervised way using the backpropagation algorithm [50]. When the input to a neural network contains symbolic categorical features (e.g., features that take one of k distinct symbols, such as words from a closed vocabulary), it is common to associate each possible feature value (i.e., each word in the vocabulary) with a d-dimensional vector for some d. These vectors are then considered parameters of the model and are trained jointly with the other parameters. The one-hot encoded-words are mapped to the word vectors. If a multilayer perceptron model is used, then the word vectors are concatenated before being fed as input to the model. If a recurrent neural network is used, then each word may be taken as one input in a sequence. This approach of learning an embedding layer requires a significant amount of training data [51]. In Figure 2.12, we see an example of embeddings transforming a sequence of numbers to a higher dimensional representation.

### 2.3.6 Temporal Convolutional Networks

Temporal Convolutional Networks (TCNs), proposed by Lea et al. (2016) [52], is a variant of convolutional neural network (CNN) [53] which employs causal convolutions [54] and dilations [55] to learn from sequential data with temporality. TCNs typically follow an encoder-decoder architecture and can process sequences of any length, mapping it to an output

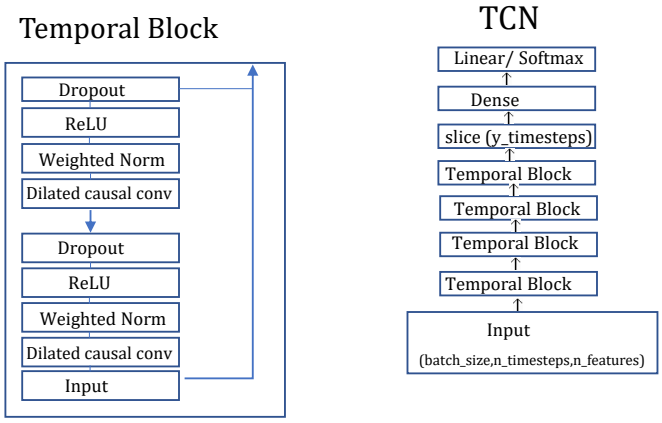


Figure 2.13: General architecture of temporal convolutional networks

sequence of identical length. To accomplish this, the TCN uses a 1D-fully convoluted network architecture (FCN), where the length of every hidden layer is identical to the input layer. Padding of zero length (kernel size - 1) is added to keep succeeding layers the same length. i.e., convolutions where output at time t is convolved only with data elements from time t and earlier in the preceding layer. Using dilated convolutions, the TCNs can look back at history with size linear in the depth of the network and the filter size. An important feature is that the output (at time t) is only convolved with the data elements that appeared before t. Furthermore, as the convolutions in TCN are causal, is no information "leakage" from future to past [56] . TCNs can be represented as 1D FCN + causal convolutions (Figure 2.13)

TCNs provide several advantages over RNN based architectures by offering parallelism of computation, flexible receptive field size, and requiring lower memory for training. We used TCNs for learning representations and identifying patterns in IO accesses. As TCNs

implement memory (causal dilated convolutions), it considers recent data to differentiate between interleaved I/O accesses, enabling effective 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.

# Chapter 3

# SSD Failure Prediction

## 3.1 Introduction

NAND flash based solid state drives (SSDs) represent an important storage tier in data centers holding most of today's warm and hot data. However, as SSDs are built from semiconductors lacking mechanical components such as spinning disks, they are also more reliable and less prone to failures compared to HDDs. The number of SSDs shipped each year has increased steadily by 42.5% over the last decade, now exceeding exabytes of storage capacity every year [57]. SSD manufacturers have employed three main techniques to increase the storage density over the past years including planar scaling, 3D integration, and multi-level cells. While beneficial for the storage density, these mechanisms have reduced the endurance, retention, and reliability of SSDs [58], [13], [59], requiring increasingly sophisticated encoding and fault tolerance mechanisms. Nevertheless, even with advanced fault tolerance techniques and low failure rates, large Hyperscale data centers utilizing 100,000's of SSDs suffer from

multiple device failures daily. Data center operators are interested in predicting SSD device failures for two main reasons. First, even with RAID [60] and replication [61] techniques in place, device failures induce transient recovery and repair overheads affecting the cost and tail latency of storage systems. Second, predicting near-term failure trends helps to inform the device acquisition process enabling to save costs and avoid capacity bottlenecks. As a result, it is important to predict both the short-term individual device failures as well as near-term failure trends.

Prior studies on predicting storage device failures [62], [63], [64], [65] focused primarily on traditional hard disks, however, due to the fundamentally different architecture of SSDs, prior techniques and findings are not readily applicable to SSDs. Research that has particularly focused on SSDs [66], [67], [18], [68] generally concentrated on understanding specific errors and issues within SSDs, limited to a controlled laboratory environment. Most studies that analyzed SSDs in the field focused on understanding correlations among specific workloads, their induced number of writes and bit errors, as well as their effect on the reliability of SSDs [69] [70], [71]. Alter [72] and Schroeder [7] analyzed authentic SSD logs collected in the Google cloud to leverage machine learning (ML) techniques for predicting the likelihood of SSD failures. While most related to our work, their proposed models either fall short on determining failed drives, or produce a large number of false positives, thereby lowering the performance of the prediction models. In particular, these two prior works suffer from the following main challenges. First, as they utilize black-box ML techniques, they are unaware of the underlying failure reasons rendering it difficult to determine the failure types that these

models can predict. Second, the models in prior work struggle with dynamic environments that suffer from previously unseen failures that have not been included in the training set. These two challenges are especially relevant for the SSD failure detection problem which suffers from a high class imbalance. In particular, the number of healthy drive observations is generally orders of magnitude larger than the number of failed drive observations, thus posing a problem for many ML models.

We determine the best performing ML approaches for predicting SSD failures and then explore optimization techniques, including feature selection and data normalization, to address the challenges of large feature spaces and highly imbalanced datasets. Consistent with the prior work [72], we use the receiver operating characteristic area under the curve (ROC AUC) score [73] to evaluate the performance. With these optimizations in place, our best approach outperforms all prior approaches by at least 9.5% ROC AUC score. To address the challenges mentioned earlier, we propose utilizing *1-class ML models* that are trained only on the majority class. By ignoring the minority class for training, our 1-class models avoid overfitting to an incomplete set of failure types, thereby improving the overall prediction performance by up to 9.5% in terms of ROC AUC score. The benefit of our proposed technique becomes even more evident when we reduce the types of failures included in the training set of the baselines approaches, showing 13% to 33% improvements using our proposed 1-class approaches over prior work. Furthermore, we introduce a new learning technique for SSD failure detection, *1-class autoencoder*, which enables interpretability of the trained models while providing high prediction accuracy.

In particular, 1-class autoencoders provide insights into what features and their combinations are most relevant to flagging a particular type of device failure. This enables categorization of failed drives based on their failure type, thus informing about specific procedures (e.g., repair, swap, etc.) that need be applied to resolve the failure. Given the low overall failure rates of SSDs, and the importance of predicting all failures, the goal is to predict all SSD failures with fewest number of false positives. An incorrect prediction of a drive going to fail as healthy is much more costly (due to the possibility of data loss, application crashes, etc.) than wrongly flagging a healthy drive as a potential failure. In particular, for highly imbalanced datasets where typically only one out of 10,000 SSDs fail, only a recall of greater than 99% represents a useful result.

To summarize, in this work, we address the challenge of accurately predicting the failure of individual SSDs with a comprehensive analysis [74], [37], [45], [75] of various machine learning (ML) models. For analysis and evaluation of our proposed techniques, we leverage a cloud-scale dataset from Google that has already been used in prior work [72], [76]. This dataset contains 40 million observations from over 30,000 drives over a period of six years. For each observation, the dataset contains 21 different SSD telemetry parameters including SMART (Self-Monitoring, Analysis and Reporting Technology) parameters, the amount of read and written data, error codes, as well as the information about blocks that became non-operational over time.

## 3.2 Research Questions (RQ)

The following lists the research questions that we are looking to answer in this project.

### 3.2.1 RQ A: Which SSD telemetry features contribute to SSD failures?

The current knowledge about SSD failure characteristics is mainly supplied by vendors based on accelerated lab testing under controlled conditions using synthetic traces. There are some prior large-scale field studies on SSD failures from Facebook [4], and Google [12], [72], but they are limited in their scope, and little is understood about SSD failures in real-world situations. In this project, we collect a wide variety of SSD telemetry information (24 parameters) and run feature selection algorithms to select the most relevant characteristics that contribute to SSD failures.

### 3.2.2 RQ B: Can we predict SSD failures by training only on healthy drives (one class training)?

One of the major challenges in predicting SSD failures is class imbalance, as healthy drive data are easier to collect compared to failures (which are inherently rare). To address the challenge, we proposed a technique of predicting SSD failures which only uses the majority class for training the models. The approach is easier to scale and is independent of the number of failed drive data samples in the data.

### 3.2.3 RQ C: How does the performance of one-class model training compare with state-of-the-art techniques?

Training on one class offers many advantages, as discussed above. In this project, we compare the performance of our one-class models with state-of-the-art SSD failure prediction techniques and show that our approach improves the model performance by up to $1.3\times$. Our approach also takes lower training and inference time due to the lower number of input features.

### 3.2.4 RQ D: Can we interpret SSD failures?

Recent advances in the field of interpretable machine learning focus on understanding how a machine learning model reaches its decisions. It helps in making the models more transparent and less biased. In this project, we want to query the model to find out why a model flags a particular SSD is destined to fail. This helps vendors understand why a drive failed and also can help in re-servicing the drives to prolong the lifetime. While training on autoencoder based models, we used the reconstruction error to interpret reasons why a particular drive was flagged by the model.

### 3.2.5 RQ E: Can one class models be used to predict unseen SSD failures?

Since one class training does not require failed SSD samples in training, we show that our approach based on training on healthy drives can predict not only unseen SSD failures but also outperform the state-of-the-art approaches for predicting SSD failures.

## 3.3 Research Contributions

Prior work on SSD failure prediction suffers from three shortcomings:

(i) the limited overall accuracy of predicting failures,

(ii) the inability of reliably predicting previously unseen failure types, and

(iii) the lack of interpretability of predictions.

To address these challenges, we provide the following contributions. First, we provide a comprehensive analysis of machine learning techniques to predict SSD failures with the highest recall and accuracy for both the majority and minority classes with fewest number of false positives. We optimize our approaches by addressing the challenges of imbalanced data sets and feature explosion. Second, we show how 1-class predictive models can be used to predict previously unseen failures in a dynamic data center environment. Third, we propose 1-class autoencoder, an approach to interpret the predictions of our model, to enable understanding of the most important reasons for failures.

### 3.3.1 Accurate Prediction of SSD Failures

Our data set contains observations from over 30,000 drives from a major cloud service provider over a time span of 6 years where each SSD observation contains the values of 24 distinct features including SMART features, the amount of read and written data, error codes, board temperature and power characteristics and more. Predicting device failures from this data poses two challenging problems. First, due to the large feature space, machine learning models

suffer from the *curse of dimensionality* as the time requirements of an algorithm grow with the number of features, often exponentially. Secondly, the data from which our models need to infer failures suffer from a significant class imbalance problem, as there exist a significantly greater number of healthy than failed devices in our data set.

#### 3.3.1.1 Feature Selection

To address the curse of dimensionality introduced by large feature sets we developed a feature selection mechanism for improving SSD failure prediction. The goals was to select the most distinguishing features from this data set for training that enable to detect SSD failures. In contrast to prior work on finding anomalous behavior in cloud systems [77], we performed an extensive study of eight different filtering mechanisms to rank the different features in order of their importance. We observed that except for the top 12 candidates, the order computed by the different selection algorithms varied substantially and, in fact, utilizing the mechanisms individually lead to high variation in model performance. To address this challenge, we developed an approach to combine the rankings of different feature selection algorithms subsequently leading to the best model performance both in terms of training runtime and accuracy, as we will be showing in Section 3.5. To further motivate the application of feature selection mechanisms, we can perform a principal component analysis (PCA) of the feature space which transforms high dimensional data into a 3D space. The distance between points in the lower dimensional space hereby reflect the correlation between the selected features. Figure 3.1 shows the PCA for the full feature space.

Each blue dot represents an observation of a healthy drive and each red cross repre-

sents an observation of a failed drive in the low-dimensional feature space. As can be seen, some of the red crosses reside within the blue observation cloud rendering it challenging for the predictive model to separate healthy from failed SSDs. In contrast, Figure 3.2 shows the PCA after feature selection. Failed SSD observations are now clearly separated from the healthy observation enabling the machine learning model to perform accurate predictions. As a result, applying feature selection does not only improve the runtime but also the accuracy of the machine learning models.

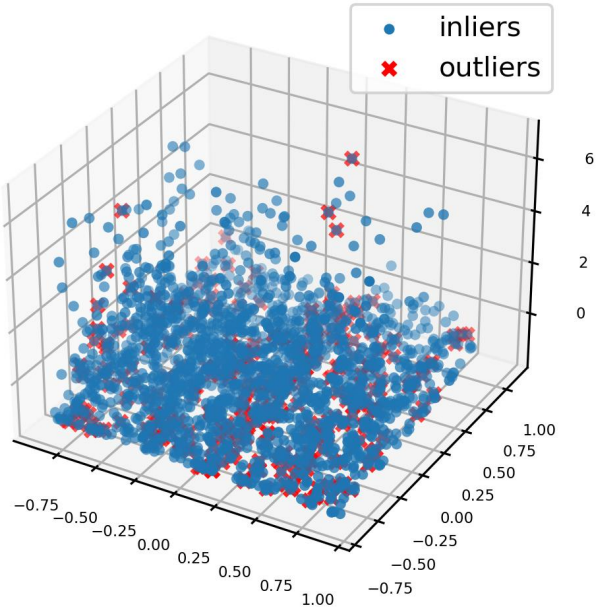


Figure 3.1: PCA with all 21 Features

In contrast, Figure 3.2 shows the PCA after feature selection. Failed SSD observations are now clearly separated from the healthy observations enabling the machine learning model to perform accurate predictions. As a result, applying feature selection does not only improve the runtime but also the accuracy of the machine learning models.

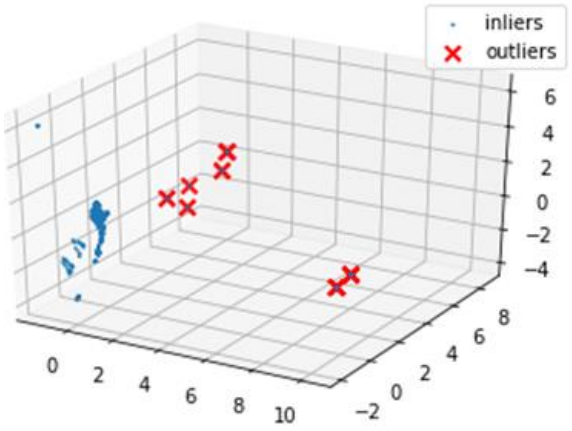


Figure 3.2: PCA with Top 9 Selected Features

#### 3.3.1.2 Class Imbalance

A major challenge in anomaly detection is to deal with the inherent class imbalance problem. Among the over 30,000 drives that we examined, about 4,000 SSDs failed at some point in time, however, for the most of its lifetime, every SSD behaves like a healthy drive. This resulted in a training dataset containing over 40 million data points for healthy drives (majority class) while only 15,000 data points for failed drives (minority class). Distinguishing between healthy and failed drive observations is further aggravated by the fact that some of the drives were put back into service after repair and then failed again, requiring to be treated as separate failure observations. Some drives that failed during the data collection process were removed and hence we no longer received data for them. Hence, while we had thousands of observations from each healthy drive, we only had limited observations from each failed drive as illustrated in Figure 3.3.

Since the size of the majority class is three orders of magnitude larger than the size of

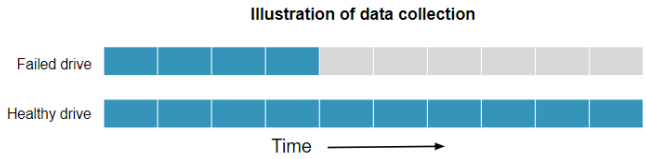


Figure 3.3: Illustration of data collection process

the minority class, recognizing instances of the minority class during classification is challenging, since many of the ML algorithms are designed to be biased toward the majority class. The data points at which the minority instances are positioned among the majority instances in an imbalanced scheme contributes to the increase in misclassification rate, thus commonly referred to as data difficult factors [31]. These factors include, but are not limited to, small disjuncts, class overlap, borderline, noise, outliers, and rare instances [74].

Prior research [72], [77], [78], [79] used techniques including Random Forest [72], Neural Networks [80], k-Nearest Neighbours (k-NN) [81], and 2-Class Support Vector Machines (SVM) [82] for predicting storage device failures. While these techniques work well for many applications, they are not free from limitations. We observe that these approaches cannot cope well with the high class-imbalance and overfit to the failure types contained in the training dataset. In Section 3.5 we show that our proposed techniques based on 1-class predictive models, and oversampling and boosting outperform prior works by up to 9.5% and reduce training times by up to 1.8×. We also evaluate our proposed feature selection techniques and perform a sensitivity study on how far ahead the models can predict the failures.

### 3.3.2 Predicting unseen failures

As outlined in Section 5.3, flash devices suffer from a variety of different failures induced by write amplification, grown bad blocks, controller errors, and backup battery issues. As some of the failures are workload-dependent and SSD technologies change, that is, the move from TLC to QLC cells, it is difficult to collect data about every failure type. Hence, it is unlikely that any training dataset would cover all types of device failures that may occur in the future.

We observed that previous approaches to detect SSD failures generally fail to predict unseen failure types that have not been experienced by the model during training. In this work, we propose to improve the adaptivity of the predictive models by training them only on the majority class instances. By utilizing only healthy drives as training data, the models can learn a strong representation of healthy drives, without overfitting to a limited set of known or previously seen failure types.

We introduce two mechanisms to enable this approach including *1-class isolation forest* and *1-class autoencoders* [47]. The generic isolation forest [37] is a popular algorithm for performing anomaly detection based on Random Forest. The algorithm leverages the fact that anomalous data points generally satisfy fewer conditions than normal data points. Hence, an anomaly score can be computed by counting whether the number of conditions required to separate a given data point is below a certain threshold. We also explored different contamination factors (the fraction of anomalous data points) to inform the model about this additional information. Utilizing these optimizations, we show in Section 3.5 that anomalous drives can

be determined with a high recall of 0.99, even though the model had never seen a failed drive during training. Furthermore, to the best of our knowledge, this is the first work to use 1-class autoencoders for predicting SSD failures. We designed an 1-class autoencoder based model that generates a compressed knowledge representation of the original input of healthy drive observations as well as a trained decoder which, in return, tries to generate healthy drive observations from the compressed representation. We remove all failed drive observations from the dataset for training the autoencoder model, in order to enable the model to learn a compressed representation of what a healthy SSD should look like. Reconstruction error [83] is used to interpret the decisions emitted by this model. Figure 3.4 shows the internal design of autoencoders. We first encode and then decode a particular sample of SSD using the autoencoder model. If the input and output are similar, the input likely corresponds to a healthy drive, whereas, if the input and output suffer from a large reconstruction error, then the sample is flagged as an anomaly (failed drive).

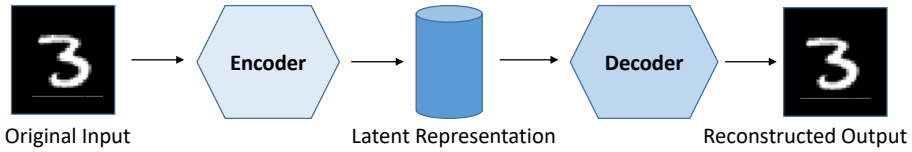


Figure 3.4: Autoencoder Design

As we show in Section 3.5, training on only the healthy drives provides the following benefits. First, the training is not limited by learning from a few samples in the minority classes. Second, the training examples from healthy drives are easier and cheaper to record, which improves the scalability of our approach. Third, ignoring the minority class during training

improves the ability of the model to predict previously unseen failures.

### 3.3.3 Interpreting SSD Failures

Understanding the reasons for an SSD drive failure is of primary concern for manufacturers and data center operators to improve the reliability and to inform about the required maintenance and repair procedures. This enables them to choose appropriate drives for a particular workload, providing the best reliability as well as enabling fast re-servicing of drives. Providing an understanding of SSD failures also helps with increasing the transparency of our predictions and avoids running full diagnostic tests to determine the causes of a failure.

Autoencoders have shown promise in interpreting model predictions generated by DNN based models (Cite). We leverage our neural network based 1-class autoencoder approach to enable this capability by creating a compressed lower-dimensional representation of healthy drive observations as explained in the previous section. We then use this representation to select anomalous observations that do not conform to the representation, thereby generating an output that differs significantly from representation of healthy observations.

The observations that produce a reconstruction error greater than a chosen threshold are flagged as failures. We then categorize these generated outputs by separating them into buckets, each one representing the error while reconstructing the input for each feature. The features that produce a larger than average error for a particular drive are then marked as significant and reported. We show in Section 3.5 how interpreting this data provides insights into why the model predicted a particular device as a failed drive.

## 3.4 Methodology

Our dataset contains SSD Telemetry data from over 30,000 drives over a period of 6 years collected from Google datacenters. In total the dataset contains 40 million observations with 21 different telemetry parameters. Around 4,000 drives failed during this period leading to 15,000 observations classified as failed from the total of 40 million observations. The dataset contained information on four different SSD models (MLC A, B, C and D) and contained no information on specific vendors. Our feature selection process (see Section 3.5.5.2) did not select the model as a significant feature and hence we excluded it during training process. Of the drives that failed, approximately 90% of the drives failed only once while the rest failed up to four times. For our work, we label each failure as a separate case. Drive replacement times, upon failure varied widely ranging from under a week (80% of the cases), to over three months (10% of the cases).

Around 30% of the drives that failed during the data collection process were replaced while the rest were removed, and hence no longer appeared in the dataset. As a result, we obtained approximately 300 observations for each healthy drive and 4 to 140 observations for each failed drive. The entire list of metrics of features present in the dataset is shown in Table 3.1.

Traditionally, the drive replacement policy at cloud service providers uses a rule-based approach [84]. Whenever certain parameters such as UECC error count, reserve block count, etc., reach a certain value, the drive is replaced. However, this approach suffers from two shortcomings. First, these rules do not comprehensively predict all the failures, and hence the drives fail unexpectedly in certain cases, resulting in data loss and application crashes. Second,

| Features | Datatype | Description |
|---|---|---|
| drive id | string | Unique ID assigned to each drive |
| model | string | Drive model type |
| timestamp | int | Time (in us) since the drive was first put in use |
| read count | int | Number of read operations in the drive's lifetime |
| write count | int | Number of write operations in the drive's lifetime |
| erase count | int | Number of erase operations in the drive's lifetime |
| status read only | boolean | Status flag indicating if the drive is operating in read only mode |
| cumulative p/e cycle | int | Number of times a memory cell is erased and reprogrammed |
| factory bad block count | int | Number of non-operational data blocks upon drive purchase |
| cumulative bad block count | int | Number of blocks which became non-operational during the drive's lifetime |
| status dead | boolean | Status flag indicating if the drive is currently failed |
| correctable error count | int | Number of uncorrectable ECC errors during read |
| erase error | int | Number of erase operations that resulted in an error |
| final read error | int | Number of read operations that resulted in an error, even upon retry |
| final write error | int | Number of write operations that resulted in an error, even upon retry |
| meta error | int | Number of errors while accessing the drive's internal metadata |
| read errror | int | Number of read operations that resulted in error, but succeeded upon retry |
| response error | int | Number of bad responses from the drive |
| timeout error | int | Number of operations that timed out without completion |
| uncorrectable error (UECC) | int | Number of uncorrectable ECC errors encountered during read operations |
| write error | int | Number of write operations that resulted in error, but succeeded upon retry |

Table 3.1: All 21 features collected

these rule sets have also been shown to be overly conservative, leading to many cases where drives are replaced even though they were still operating normally. The aggregate number of drive failures per week is also beneficial for cloud providers as they can order replacements in advance. These issues motivated us to develop a more flexible and accurate approach based on machine learning techniques.

### 3.4.1 Data Preprocessing

The data collected contained features in string, date time, and integer format. We ensured that all the data collected was transformed into numeric format so that it can be processed by the machine learning models. String values, such as Drive model name, were converted into categorical features, and date and time were converted into UNIX timestamps. We treated each

data point as an independent observation and normalized all the non-categorical data values to be between 0 and 1. We created separate datasets, identified by the parameter $N$, by selecting daily observations before a predicted failure occurred. For instance, $N = 3$ contains all observations for each drive 3 days before the drive either failed or was still functional. We leverage this data in order to find out how far ahead our proposed models can predict the failures.

### 3.4.2 Feature Selection

One of our primary goals was to select the most distinguishing features that are highly correlated to the failures for training. We used three different feature selection methods, Filter [85], Embedded [86], and Wrapper [87] techniques, and implemented eight different algorithms including Pearson ranking [88], Spearman ranking [89], Chi square test [90], Analysis of Variance (ANOVA) [91], Recursive Feature Elimination [92], Extra Trees [93], Lasso Regularization [94], Elastic Net [95], and Ridge Regression [96], for selecting the most important features contributing to failures for our dataset.

#### 3.4.2.1 Filter Methods

Filter methods for feature selection use statistical measures to provide scores for each feature. The features were then ranked by this score and only the top significantly correlated features were selected. Specifically, we used Pearson correlation [97], Spearman correlation [98], Elastic Net [99], and Kendall Tau [100] ranking algorithms to rank the features.

#### 3.4.2.2 Wrapper Methods

Wrapper methods select different combinations of features and then evaluate them to pick the most relevant features. A prediction model is typically used to evaluate the combinations and assign scores based on model accuracy. We used different search processes including Random Forest [101], Recursive Feature Elimination with Extra Trees classifiers [102] and Logistic Regression [103] to select the top features.

#### 3.4.2.3 Embedded Methods

Embedded methods pick the most relevant features that contribute to the accuracy of the model during the creation and training of the model. LASSO (L1) [104], Elastic Net [99], and Ridge Regression (L2) [105] are the most commonly used regularization methods. These methods optimize the learning procedure by training models with lower complexity, where features with non-zero coefficients are selected for training the model, thus serving as methods for feature selection. The three methods above provide feature rankings which were then merged into a single list, giving equal importance to each method. As we show in Section 3.5, the elaborate feature selection process improves both the training time and the prediction accuracy significantly over the baseline that utilizes all 21 features. The resulting set of top features is shown in Table 3.2. We validated the feature selected with domain experts, who confirmed that there is a strong correlation between the features that were picked by the feature selection algorithms and actual parameters which indicate wear out and failures in SSDs.

| Final Selected Top Features |
|---|
| correctable error count |
| cumulative bad block count |
| cumulative p/e cycle |
| erase count |
| final read error |
| read count |
| factory bad block count |
| write count |
| status read only |

Table 3.2: Top features selected

### 3.4.3 Oversampling and Boosting

Prior research [72], [77], [78], [79] used techniques including Random Forest [72], Neural Networks [80], k-Nearest Neighbours (k-NN) [81], and 2-Class Support Vector Machines (SVM) [82] for predicting storage device failures. While these techniques work well for many applications, they are not free from limitations. We observe that these approaches cannot cope well with the high class-imbalance and overfit to the failure types contained in the training dataset.

We noticed there were two directions to improve the model's ability to separate the two classes (healthy and failed). One way is to enhance the learning of the minority class which has fewer training examples and the other one is to address the class imbalance itself. To improve the learning of the minority class, several learners can be combined which is referred to as Boosting [32]. The basic principle behind the boosting mechanism is to leverage multiple weak learners and combine their predictions to form one strong rule. SMOTEBoost [34], uses oversampling and boosting (using AdaBoost classifier [36] to generate additional synthetic data

samples for the minority class. Using SMOTE and AdaBoost [36], the algorithm models the minority class not only by providing the learner with the minority class (failed) observations which were mispredicted in the previous boosting iteration but also with broader representation of those instances (generated by SMOTE). Traditionally, Boosting algorithms provide equal weights to all mispredicted examples and since the training set consists of observations from the majority class (healthy drives), the training set is still skewed towards the majority class. To address this, SMOTE is introduced during each round of boosting to increase the number of minority class samples for the weak learners and focus on these cases in the distribution at each boosting round. This process also enhances the diversity among the classifiers in the ensemble by producing a different set of synthetic samples at each iteration [106].

Supervised techniques, in particular, have shown promise in isolating anomalies, but due to the inherent high-class imbalance, there exist very few training examples of the minority class. In order to capture all the failed drives, a relatively large class separation threshold has to be chosen producing a large number of false positives. Furthermore, we show that our introduced 1-class approaches can outperform prior work and can predict unseen failure types not seen during training.

### 3.4.4 1-Class ML Models

For training the 1-class models, autoencoder and isolation forest, we used the H2O library [107] and split the dataset into training and test set. The training set contains data from 90% of the healthy drives but does not contain any samples of failed drives. For 1-class isolation forest, we use 250 trees, with a $max\_depth$ of 20 to get a good representation of a healthy drive

from the input data. Increasing the tree size and $max\_depth$ beyond these values decreased precision of the model, indicating overfitting. We also experimentally explored the best value for the *contamination factor* hyperparameter. The initial hyperparameter values were based on domain knowledge and we performed extensive parameter sweeping and tuning (also for the baselines) to come up with the final hyperparameter values and models. While our training set has zero contamination (no failed drives), we need to inform the model about the contamination factor during inference so that the model can adjust the threshold to select between failed and healthy drives. The empirically determined contamination factor depends on the number of days the model needs to predict ahead and ranges between 0.016 and 0.002.

The 1-class autoencoder model utilizes 4 hidden layers comprising of 50, 25, 25 and 50 neurons respectively. The neurons utilize a $tanh$ activation function. We utilize the Adam optimizer [108] and train the model for 100 epochs. We use early stopping with a patience value of 5 ensuring that the training of the model stops when the loss does not decrease after 5 consecutive epochs. Increasing the number of hidden layers beyond 4 increases the training time significantly without providing performance benefits. We use 10-fold cross validation to evaluate all models.

### 3.4.5 Deployed System

The processed dataset containing only the top selected features is subsequently used for training the different ML models. In a datacenter we envision our SSD failure prediction

technique to be implemented as shown in the block diagram in Figure 3.5. The telemetry traces are collected periodically from all SSDs in the datacenter and sent to the preprocessing pipeline transforming all input data into numeric values while filtering out incomplete and noisy values.

Following data preprocessing, feature selection is performed to extract the most important features from the data set. The preprocessed data is then either utilized for training or inference. For inference, device anomalies are reported and classified according to our 1-class autoencoder approach. SSDs can then be manually analyzed by a technician or replaced directly. As an alternative, a scrubber can be leveraged to validate the model predictions by performing a low level analysis of the SSD, finding grown bad sectors and other drive issues.

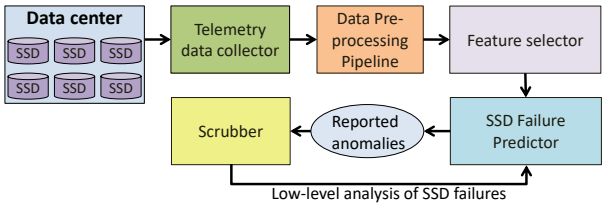


Figure 3.5: Block diagram of the Deployed System

## 3.5 Results

### 3.5.1 Oversampling and Boosting

In this section, we compare the performance of different ML techniques for predicting SSD failures with our oversampling and boosting technique. For SSD failure prediction, the primary goal is to predict all SSD failures since the cost of not catching (mispredicting) a drive

that is going to fail is much higher than classifying a healthy drive as a failure which can be refuted by scrubbing [109]. Nevertheless, as performing scrubbing induces a performance overhead, both achieving high recall for failed devices and high accuracy for healthy devices is important. To satisfy these requirements, for all the experiments we chose a high enough threshold to capture all failures and then try to predict these failures with the fewest number of false positives. We also record the receiver operator characteristic area under curve (ROC AUC) score [110] for each experiment which is a measure of the predictive performance of the binary classifier. This metric is selected as it is insensitive to the class imbalance as in our case and have also been used in prior work [72]. SMOTEBoost and RUSBoost algorithms were chosen as it is specifically designed to learn from imbalanced datasets. They employ oversampling to make the dataset more balanced and use boosting [111] to improve learning. Since these methods are designed to learn from the minority class, we included 90% of healthy and faulty observations in the training set and remaining observations in the test set. We only included the selected top features for training, using k_neighbor = 5, n_estrimators = 50 and learning rate of 1.2.

We can see that SMOTEBoost works best in predicting drive failures achieving near perfect precision, recall and Fscore as well as a ROC AUC score of when predicting one week in advance. This model is up to seven times better in predicting failed drives compared to the next best model, Isolation Forest. Neural network and One-class SVM based models as used in prior works [72, 109, 77] performed poorly delivering low precision and suffering from a high number of false positives in the failed class. One-class SVM and neural network

| Learning Technique | Class | Accuracy (%) | Precision | Recall | Fscore |
|---|---|---|---|---|---|
| SMOTEBoost | N-Fail (0) | 97.5 | 0.98 | 0.97 | 0.97 |
| | Fail (1) | 94.3 | 0.96 | 0.95 | 0.95 |
| RUSBoost | N-Fail (0) | 99.7 | 0.99 | 0.99 | 0.99 |
| | Fail (1) | 57.6 | 0.24 | 0.4 | 0.34 |

Table 3.3: Results from SMOTEBoost and RUSBoost

based methods performed almost as bad as random, with AUC scores close to 0.5. The Random Forest and Isolation Forest based models performed more than twice better than Neural Network and 1-class SVMs for precision for failed drives, but still provide much lower accuracy than SMOTEBoost.

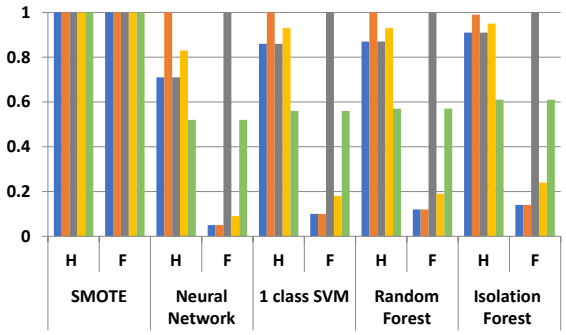


Figure 3.6: Predicting 1 week ahead (N=1)

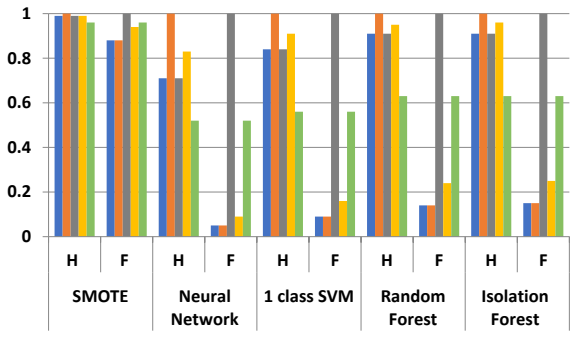


Figure 3.7: Predicting 2 weeks ahead (N=2)

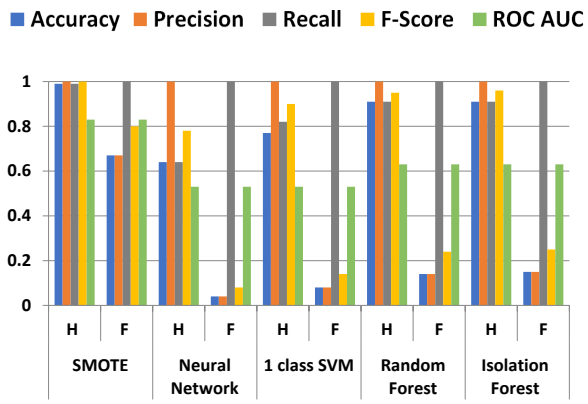


Figure 3.8: Predicting 3 weeks ahead (N=3)

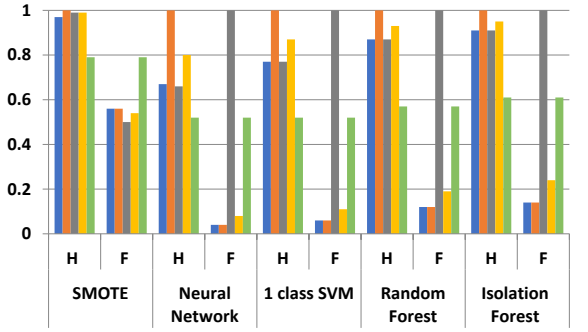


Figure 3.9: Predicting 4 weeks ahead (N=4)


Figure 3.10: Comparison of Different Machine Learning Approaches for Prediction of Healthy (H) and Failed (F) drives

.

The models were trained using the SAMME.R [112] algorithm. Table 3.10 shows the performance of SMOTEBoost and RUSBoost for predicting SSD failures when predicting 1 week ahead. From the figures and table above, we can see we oversampling and boosting is effective is predicting SSD failures with high precision, recall and accuracy. The classifier achieves highest ROC-AUC score compared to the baseline techniques we used. However,

one critical limitation of this approach is the inability to predict failure types not seen during training. Hence, the models tend to overfit to the set of known failure types seen during training and the performance degrades on predicting failures on new unseen data. To overcome this problem, we discuss the performance of 1-class approaches next which requires training only on the majority class.

### 3.5.2 Accurate Prediction of SSD Failures

In this section, we compare the performance of our proposed 1-class isolation forest and 1-class AutoEncoder techniques to three baselines used in prior work. In particular, we compare against, Random Forest, 2-Class SVM, and Neural Networks (NN) as those have been used in prior work on SSD failure detection [72]. For the baselines, whenever available, we use the same model architecture and hyperparameters as proposed in prior work [72]. For the hyperparameters that we could not find in prior work, we performed a design space exploration and report the best numbers that we could find.

For imbalanced datasets, traditional metrics (accuracy, precision, recall and fscore) alone can be deficient in measuring the performance of the classifier. Since the dataset is imbalanced, overfitting to the majority class (predicting all observations as the majority class) can skew performance and still reflect good overall precision, recall and fscore. The receiver operating characteristic curve, or ROC curve [73], is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. The ROC curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold values. The true-positive rate is also known as sensitivity, recall or probability

of detection in machine learning [113]. The false-positive rate is also known as probability of false alarm [114] and can be calculated as (1 - specificity). The area under the curve (ROC AUC) [110] is calculated to give a single score for a classifier model across all threshold values. This is inline with prior work that utilizes the ROC AUC metric for evaluating anomaly detection models [72].

To evaluate the five ML techniques we first label all 40 million observations in the dataset to separate between healthy and failed drive observations. We then perform a 90% - 10% split of the dataset into training set and evaluation set. For training the 1-class models we remove all failed drive observations from the training set, however, the evaluation set is identical between our proposed 1-class techniques and the three baselines. We use 10-fold cross validation for evaluating all approaches.

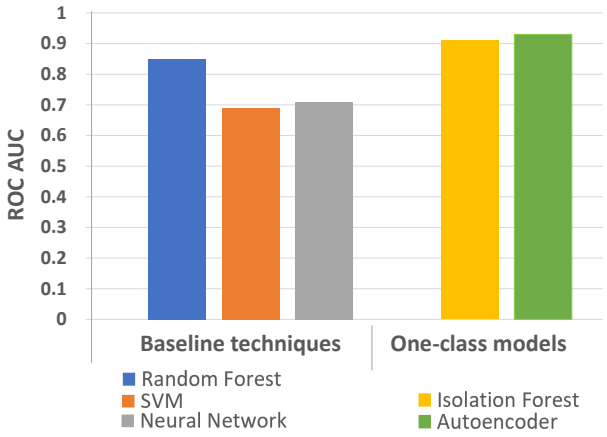


Figure 3.11: ROC AUC Score comparison of the five evaluated Machine Learning Techniques

Figure 3.11 illustrates the comparative performance of different ML techniques for predicting SSD failures one day ahead. Among the baselines, Random Forest performs best, providing a ROC AUC score of 0.85. Both our 1-class models outperform the best baseline.

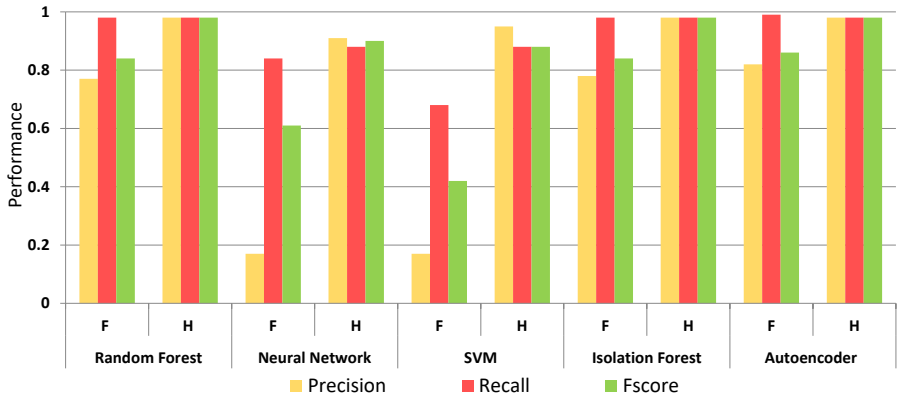


Figure 3.12: Accuracy, Precision, Recall and Fscore for the five evaluated Machine Learning Techniques

In particular, 1-class isolation forest achieves a ROC AUC score of 0.91, representing a 7% improvement over the best baseline while 1-class AutoEncoder, outperforms Random Forest by 9.5%.

ROC AUC determines the ability of a model to distinguish between classes (failed vs. healthy in our application). To achieve good performance, models need to achieve both high recall and precision for both failed and healthy classes. Figure 3.12 explores these metrics for the five approaches in more detail for N=1.

It shows that Random Forest performs equally well than our proposed 1-class Models in terms of Precision and Recall on the majority class of healthy (H) drives, however, performs considerably worse on predicting the minority class of failed (F) drives. For the minority class, 1-class AutoEncoders improve precision by 6% over Random Forest as well as by 68% and 72% over the Neural Network and SVM baselines respectively.

### 3.5.3 Adaptivity to Unseen Failures

In the previous section we showed that our proposed 1-class models are capable of outperforming the 2-class models by up to 72% under the best case scenario for the baselines in terms of precision (31% in terms of ROC AUC score), where 90% of all failures types are contained in the training set. We now evaluate the model's ability in adapting to new datacenter environments, induced, for instance, by new workloads or new hardware. Therefore, in Figure 3.13, we sweep the number of failed drive observations included in the training set from 10% to 100%, simulating dynamic environments where new failure types emerge over time.

Figure 3.13 shows the ROC AUC score for the three baselines and our proposed 1-class techniques with a variable percentage of failed drives included in the training set. Note that our 1-class techniques do not include any failed drives in the training set and hence we plot their performance as a straight line. The baselines' performance, however, depends significantly on the number of minority samples in the training set. For instance, if only 50% of the failed drive observations are included in the training set, our proposed 1-class Autoencoder technique outperforms Random Forest by 13% and NN and SVM by 33%. This shows that particularly in dynamic environments, our 1-class techniques are a better choice than the techniques utilized in prior works.

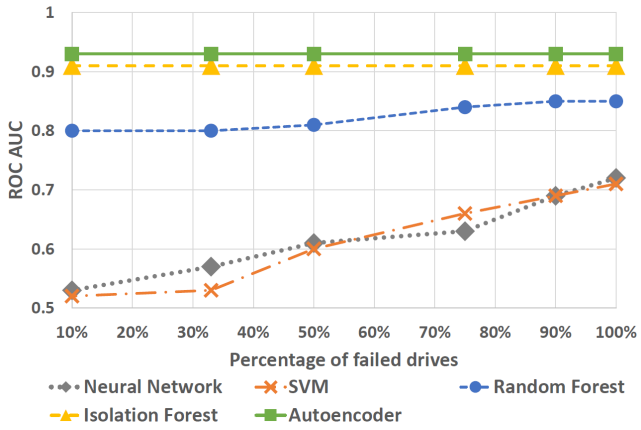


Figure 3.13: Predicting unseen failures

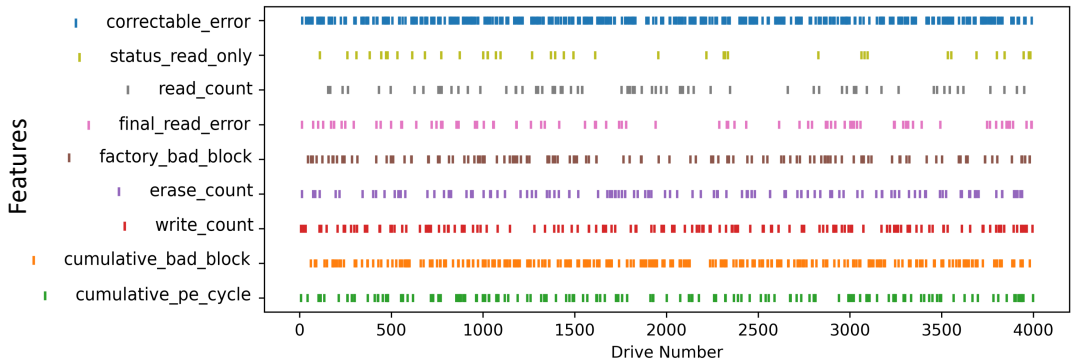


Figure 3.14: Interpreting SSD Failure Reasons

### 3.5.4 Interpreting SSD Failures

Autoencoders have shown promise in interpreting model predictions generated by DNN based models [115, 116, 117, 118]. This work proposes 1-class Autoencoders for interpreting SSD failures. In particular, our technique exposes the reasons determined by our model to flag a particular device failure. This is achieved by utilizing the reconstruction error generated by the model while reproducing the output using the trained representation of a healthy drive. The failed drives do not conform to the representation, thereby, generating an

output which differs significantly from the actual observations producing a large reconstruction error. We study the reconstruction error per feature to generate the failure reasons. The features which contribute more than the average error per feature to the reconstruction error, is defined as a *significant* reason.

Figure 3.14 shows how often a feature was flagged as a significant failure reason by the autoencoder model, aggregated for all observations from failed drives. The y-axis displays all features utilized by the model, representing a potential failure reason while the x-axis shows the failed drive number. For each drive, we report the failure reason by means of a scatterplot. From Figure 3.14, we can see that many failed drives show a higher than normal number of $correctable\_errors$ counting the number of failed reads that could be corrected leveraging error correcting codes (ECC). This indicates that a high number of uncorrectable errors frequently leads to failures, however, it is also only a significant feature in approximately 35% of the drives.

$Cumulative\_bad\_block$ represents another important reason determined by the model indicating SSD failures as it shows frequent anomalies, however, again only in less than 30% of the cases.

In summary, this analysis shows that there exist particularly relevant features that indicate device failures in many cases, however, only the combination of several features enables accurate failure prediction. . This shows that some of the failed drives has different amount of bad blocks as compared to healthy ones. Degradation of this parameter value is usually an indication of increase of number of blocks being unusable. Other features which appeared as a

reason for failure, albeit much sparingly than the two above include write count, erase count and cumulative pe cycle which is a measure of the drive age. This shows that there does not exist a single reason but rather a combination of different parameters that lead to SSD failures. We also note that, cumulative UEC (Uncorrectable error count) which has been researched extensively [119, 120, 121] for SSD failure correlation contributed to less than 1% of the failures according to our Autoencoder based model.

### 3.5.5 Sensitivity Studies

In the following we provide two additional sensitivity studies. In the first, we evaluate the ability of our models to predict failures multiple days in advance. Predicting further ahead is beneficial for logistical reasons and acquisition purposes. In the second study, we evaluate the effect of feature selection on the five approaches.

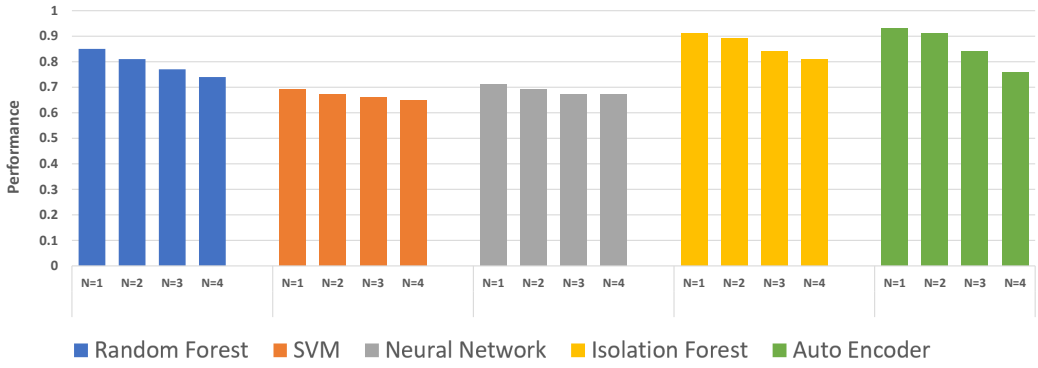


Figure 3.15: ROC AUC score when predicting up to 4 days ahead

.

#### 3.5.5.1 Predicting ahead in Time

To optimize drive maintenance and the acquisition of new spare drives, it is preferable to predict drive failures further ahead in time. While the previous sections have focused on predicting one day ahead, Figure 3.15 evaluated ROC AUC performance on predicting multiple days (N) ahead. As expected, for all five models, prediction performance degrades when predicting further ahead. So while for 1-class Autoencoders performance degrades considerably, 1-class isolation forest can maintain the performance better. In particular, for $N = 4$, the 1-class isolation forest model becomes the best performing technique outperforming the Random Forest baseline by 6% and SVM and NN by 11% and 13% correspondingly in terms of RUC-AUC score.

#### 3.5.5.2 Feature Selection

As mentioned above we used feature selection algorithms for selecting the most important features contributing to failures in our dataset. Tables 1 and 3.2 list the features before and after feature selection. Figure 3.16 demonstrates the potential benefit of using feature selection by comparing the model performance (in terms of ROC AUC score improvement) with the original 21 features against the performance of the model trained with only 9 features. As can be seen, all techniques benefit from feature selection, for instance, Random Forest's absolute ROC AUC score improves by 3.7% when utilizing feature selection, while 1-class Autoencoder's ROC AUC performance increases by 3.3%. Feature selection also reduces the number of features used for training the models resulting in up to 41% (for autoencoder models) reduction in training times as can be seen from Table 3.4.

## 3.6 Discussion

We discuss the results from our proposed Anomaly Detection based 1-class models below. By ignoring the minority class for training, our 1-class models avoid overfitting to an incomplete set of failure types, thereby improving the overall prediction performance. This makes them particularly useful in a real world setting where new types of failures emerge over time.

### 3.6.1 1-Class Isolation Forest

Anomaly detection approaches leveraging isolation forests are generally trained on both the majority and minority class. Perhaps surprisingly, we found that isolation forests trained only on the minority class performed exceptionally well, particularly for detecting unseen failures outperforming the performance of baseline approaches (Random Forest, 2 class SVM and Neural Network based models). 2-class models use data from both classes to learn a representation for each class. Since the number of samples for failed drives in our case is significantly lower than good working SSDs, the model has less samples to learn from and hence is more likely to misclassify previously unseen failed SSDs. 1-class models, in contrast, learn a representation of a good working SSD and are more likely to classify previously unseen anomalies correctly (1-class models do not suffer from overfitting to the limit training set of failed SSDs).

Our approach does not require training on all different failure types to detect failures and hence both generalizes and scales well when provided with new healthy observations. The

approach outperforms autoencoders when predicting more than two days ahead and is faster to train requiring fewer training samples. Nevertheless, it was not able to outperform autoencoders (for N=1 and N=2) due to a higher false positive rate (anomalies as reported by the model which are not actual failures). We plan to use second level supervised binary classification in the future to teach the model about the known failures to eliminate more false positives during evaluation.

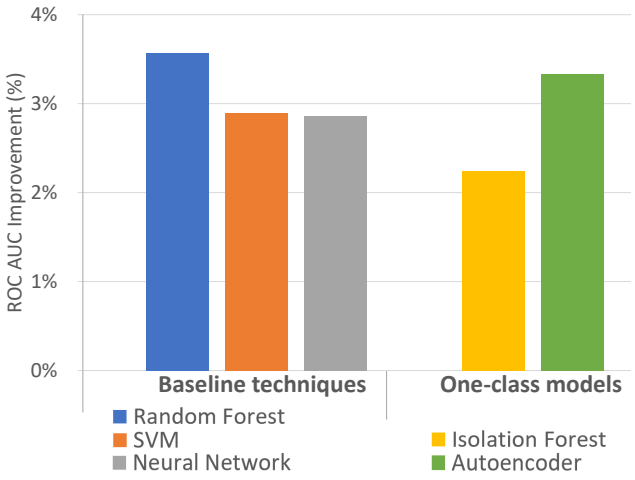


Figure 3.16: Impact of Feature Selection Techniques

### 3.6.2 1-Class Autoencoder

To our knowledge, this is the first application of a deep learning based 1-class autoencoder for predicting SSD failures. We used the data from healthy drives to create an encoded representation of a healthy drive. Upon providing the test data points to the encoded representation, we recorded the difference between the observed and generated output. Since the anomalous data points do not fit the encoding well, they tend to have higher error values. As in 1-class isolation forests, the autoencoder does not need to train on the minority dataset. autoencoders performed best while predicting failures up to 2 days ahead, achieving highest accuracy,

precision and ROC AUC score with a recall of 0.99. It performed worse than 1-class isolation forests when predicting ahead 3 or more days achieving lower precision, however, autoencoders enable interpretation of the model predictions. In particular, we can learn why the model flagged an observation as a failure to inform the repair and maintenance procedure.

| ML Technique | Features | Training Time (sec) |
|---|---|---|
| Random Forest | 9 | 695.4 |
| | 21 | 1095.6 |
| Neural Network | 9 | 1496.87 |
| | 21 | 2550.87 |
| SVM | 9 | 1156.6 |
| | 21 | 1885.6 |
| Isolation forest | 9 | 499,57 |
| | 21 | 686.54 |
| Autoencoder | 9 | 1750.57 |
| | 21 | 2781.89 |

Table 3.4: Model training time (N = 1)

## 3.7 Conclusion

This paper provides a comprehensive analysis of machine learning techniques to predict SSD failures in the cloud. Therefore, we collect SSD telemetry information from over 30,000 drives over a period of six years from Google's datacenters. We observe that prior works on SSD failure prediction suffers from the inability to predict previously unseen failure types motivating us to explore 1-class machine learning models such as 1-class isolation forest and 1-class autoencoder. We show that our approaches outperform prior work by 9.5% ROC-AUC score by significantly improving on the prediction accuracy for failed drives. For dynamic environments, where only a subset of the different drive failure types are part of the training

set, our 1-class techniques improve over the baselines by 13%. Finally, we show that 1-class autoencoders enable interpretability of model predictions by exposing the reasons determined by the model for predicting a failure.

## 3.8 Publications

Research papers summarizing our investigations were accepted at Symposium on Cloud Computing, 2020 (Improving the accuracy, adaptability, and interpretability of SSD failure prediction models), and Non-volatile Memories Workshop, 2020 (Explaining SSD failures using anomaly detection).

# Chapter 4

# Neural Network based Prefetching

## 4.1 Introduction

The recent deceleration of Moore's law [122] calls for new approaches for optimization of resources. SSDs have replaced the spinning disks (HDDs) for many applications in cloud services due to their higher I/O performance [123], lower failure rate [2], and better endurance [124]. Nevertheless, although SSDs deliver significantly higher speeds than HDDs, SSDs still remain a performance bottleneck of computing systems [125], as processors and DRAM technologies support three orders of magnitude lower access latency. Two common approaches to hide the high access latency of storage devices are caching and prefetching. Caching utilizes less dense but faster types of memory to store frequently used data items, filtering out many accesses of the slow SSDs. Examples include Linux's page cache [126] and filesystem caches [127]. Prefetching [128] approaches read data from SSDs in advance, in order to serve the later demand accesses from the cache with low latency. Prefetching can be implemented

either in software, e.g., within operating system [129], [130] or within the SSD itself [131].

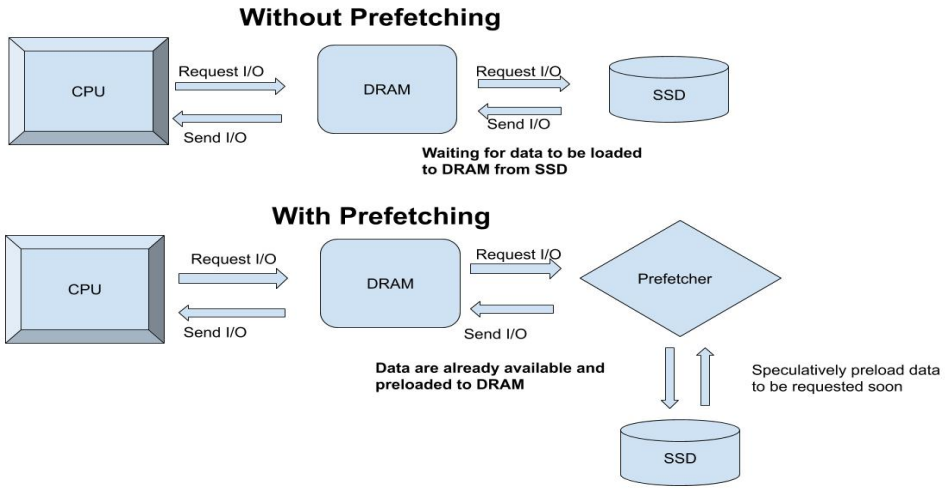


Figure 4.1: Prefetching motivation

In Figure 4.1, we show the benefits of using prefetching in SSDs. Without prefetching, as shown in the figure, every IO request from the DRAM is sent to the SSD, and the SSD responds with the data. Due to the difference in the two devices' performance as described previously, the DRAM needs to wait until the SSD responds with the data. This process is wasteful, and hence prefetching is used to overcome this problem. As shown in the figure below, with prefetching, the SSD speculatively preloads SSD data to DRAM. Therefore, the DRAM does not need to wait for the SSD to respond with the data as it is already available in the DRAM. In case the requested data is not present, the IO request is sent to SSD, which responds with the appropriate data blocks.

Existing prefetching mechanisms [132], [133] are limited by the computational complexity and difficulty of correctly predicting future I/O accesses. For instance, the read-ahead

prefetcher [134] is limited to prefetching the next data item within a file to accelerate sequential accesses. More advanced prefetchers [135], [136] that can learn complex I/O access patterns have been dismissed because of their computational cost. Recently, storage vendors, including Samsung, have proposed SmartSSDs [137], adding computational capabilities to SSDs. These devices offer new opportunities as they enable offloading of prefetching to hardware, removing the burden from the host CPU. While this approach addresses the compute overhead of prefetching, predicting future I/O accesses accurately remains a challenge. Real-world applications not only perform sequential accesses, but also exhibit complex workload patterns [138]. Applications are frequently used by multiple users simultaneously, performing independent tasks, resulting in a mix of sequential and random I/O requests which are difficult to model and challenging to predict. Furthermore, in existing systems, I/O accesses need to traverse a deeply layered software stack, transforming the easy to predict accesses on the application side into seemingly random accesses on the SSD level. Predicting future memory accesses from multiple interleaved I/O access streams on the SSD device layer hence represents a challenging problem.

Modern SSDs and operating systems offer a wide range of telemetry data for analysis. Utilizing I/O access tracing in hardware and software enables the collection of large, clean, and automatically labeled datasets that can fuel powerful machine learning models. In this work, we leverage Long Short-Term Memory (LSTM) [139] based sequence-to-sequence neural networks to learn spatial I/O access patterns of applications from block level I/O traces collected from a diverse set of data center applications. LSTMs are capable of capturing long-term depen-

dencies in data and can address sequences of different lengths. LSTMs integrate model training and representation learning together, without requiring additional domain knowledge, enabling the discovery of unseen patterns in the data to improve generalization capability of a model. In this work, we leverage LSTMs to deliver the following contributions. First, our model provides high accuracy even in the presence of complex interleaved I/O streams. Second, it addresses the challenge of timeliness by predicting multiple I/O accesses ahead of time. Third, to cope with the dynamic behavior of applications and to improve the reusability of our model, we propose an address mapping learning (AML) technique enabling our model to predict different types of workloads.

To demonstrate the practicality of our approach, we build a simulator enabling us to measure timeliness in addition to prediction accuracy. We utilize I/O traces to train the neural network models offline and predict future logical block addresses (LBAs) at runtime using the simulator. To reduce address space, we take the $l_1$ norm between a pair of consecutive memory accesses as input to the model in addition to the requested I/O size. This enables the model to also predict the *size* of the incoming I/O request, representing the amount of data blocks to be prefetched ahead of time. We show that our approach enables predicting LBAs sufficiently far ahead to compensate for the read latency of accessing flash as well as for the inference latency of our model. We present an analysis of the impact of predicting $N$ steps ahead into the future and evaluate the impact of cache size on the performance of our prefetcher. We compare our work with three baselines, a naive approach that only prefetches the most frequently accessed LBAs, a stride prefetcher [140], and the Markov chain based prefetcher [141], showing an

improvement of up to 800× over the stride prefetcher and up to 8× over the Markov chain prefetcher.

## 4.2 Research Questions

The following lists the research questions that we are looking to answer in this work.

### 4.2.1 RQ A: Can sequence-to-sequence deep learning models learn the IO access patterns in real-world applications?

Real-world applications typically exhibit complex IO patterns. Often an application is used by multiple users simultaneously, each running their own jobs. This often results in a mix of sequential and random IO requests, which are is difficult to model and hard to learn by any algorithm as used in traditional prefetchers. Time series neural networks (NN) have been shown to learn complex data patterns keeping temporality in mind [40]. In this project, we show that sequence-to-sequence LSTM based models can be used to separate the interleaved streams to generate accurate predictions about future IO accesses.

### 4.2.2 RQ B: Can the neural network address timeliness by predicting multiple accesses ahead of time?

In order to compensate for model prediction time and the time required to preload the data, the neural network prefetcher needs to predict several accesses ahead of time. Time

series forecasting allows models to predict several accesses ahead but usually at the cost of performance. In this project, we show that the NN prefetcher can maintain performance even while predicting multiple accesses ahead of time to address the timeliness of predictions.

### 4.2.3 RQ C: How does the performance of the neural network-based prefetcher compare with state of the art?

For some simple IO workload patterns, NN-based prefetching might be costly to compute as it needs additional resources. In this thesis, we perform a comparative study between NN-based pre-fetchers and state-of-the-art pre-fetchers to find out which workload patterns are complex enough to demand NN-based prefetching.

### 4.2.4 RQ D: Can we use the learned IO access patterns to predict IO accesses in new, unseen workloads?

Different workloads show similar I/O access patterns due to shared design patterns and commonly used data structures. An ideal prefetcher would be trained once on a varied set of applications, providing high-performance even for previously unseen applications. Such a prefetcher is also likely to be more robust with respect to dynamically changing data inputs or code changes to the original application. We show that learned IO access patterns can be used to predict IO accesses in unseen workloads. We also discuss the conditions where such an approach would be useful.

### 4.2.5 Neural Network based Prefetching

While most work on I/O prefetching has focused on conventional techniques, some prior works have explored using machine learning techniques. Hashemi [142] used neural network based sequence models for prefetching DRAM accesses. Prior work has also utilized semantic locality and context for prefetching in primary memory using Reinforcement Learning [143]. Peled et. al. [144] argue semantic locality can capture the relationship between data elements resulting in a high-level abstraction of data locality based on inherent program semantics instead of memory layout, and that semantic locality transcends spatio-temporal concerns. In addition, the work also used program context, which approximated semantic locality by utlizing machine context (hardware and software) as features for model training. The prefetcher identifies access patterns in DRAM accesses by applying reinforcement learning methods over code and machine attributes, which provide hints on memory access semantics.The models proposed in this work, however, cannot be applied to our problem as prefetching I/O accesses differs significantly from prefetching DRAM accesses. First, I/O accesses do not contain instruction information to enable stream disambiguation, second, I/O accesses do not have a fixed size like DRAM accesses, third, I/O accesses and DRAM accesses interact differently with the OS, and fourth, I/O prefetching models need to account for timeliness. A second line of work utilized Markov chains [145] for prefetching data from SSDs [141, 131]. We compare our approach with these prior works in Section 4.6, confirming prior observations that Markov chain based prefetchers perform poorly on real world applications where the I/O streams are more complex [146].

## 4.3 Problem Statement

We assume a digital system that consists of the following components. A flash based digital storage device (SSD) that provides high capacity but low performance, and a high access latency. A central processing unit (CPU) that can process data at orders of magnitude faster than the SSD. In addition, the system is comprised of a cache (usually DRAM) that is placed in between the CPU and the SSD. The CPU can access data with low latency from the cache, however, the cache capacity is orders of magnitude smaller than the SSD capacity. Reads access a specific logical block address (LBA) and are generally more performance critical than writes, as future operations depend on the data supplied by the reads, which is why this work focuses on reads. The goal we aim to achieve is to accurately predict future LBAs so that they can be prefetched into the cache, enabling low latency accesses by the CPU. In addition to the LBA, we also need to predict the size of the I/O, as prefetching only parts of an I/O access is useless. Thus, an efficient prefetching mechanism requires optimizing three metrics, particularly, the *coverage*, *accuracy*, and *timeliness*.

Coverage or recall refers to the ratio of future memory accesses that are attempted to be prefetched. Prefetching of an LBA is accurate if the same LBA is subsequently accessed by a demand read. Accuracy is hence defined as the ratio of accurate prefetches to executed prefetches. A prefetch is timely if it is executed sufficiently ahead of time of the demand read. In particular, $T_{cand} + T_{read} < PA * T_{arrival}$ must hold, where $T_{cand}$ represents the time to compute a prefetch candidate, $T_{read}$ represents the time to perform a read from the SSD, $T_{arrival}$ represents the inter arrival time between demand reads, and $PA$ represents prefetch-ahead, which is

the number of accesses we need to predict into the future. Executing prefetches too early is generally of a lesser concern as prefetches can be stored for a finite time in the cache. As a result, the time that a prefetch can be executed too early is bounded only by the cache capacity.

Storage accesses to an LBA are generally handled by the operating system. User applications, however, generally communicate with the storage devices by reading and writing files. Consequently, the filesystem layer within the OS needs to map file accesses to LBA accesses before they can be submitted to the storage device. Furthermore, to improve performance, the OS maintains several caching layers in the filesystem and logical block layer, aiming to filter out a significant fraction of all application accesses. The result of this architecture is that even a seemingly easy to predict operation on the application layer, such as reading a file sequentially, may result in a very hard to predict access patterns on the LBA level, as perceived by the SSD. Finally, the storage device is generally accessed by different application threads simultaneously, resulting in multiple interleaved I/O streams that are indistinguishable by the SSD. In summary, the existing storage stack architecture renders predicting future I/O accesses a challenging problem. Predictive models need to be able to separate multiplexed I/O streams and then predict future LBAs from within the hard to predict sequences. In addition, they need to provide information on the number of data blocks to prefetch, starting from the initial predicted LBA.

## 4.4 Proposed Prefetching Technique

Learning SSD storage accesses for prefetching is a challenging task for the following reasons. As SSDs are increasing their storage capacity to 16TB and beyond, drives are now supporting billions of logical block addresses. As prefetching is only successful if every bit of the logical block address is predicted accurately, models are required to predict which LBA to prefetch with perfect accuracy within a very large LBA space. This space is often sparse, as the operating system allocates blocks within the filesystem layer, and hence, even sequential data within files may be mapped to arbitrary LBAs within the SSD. Furthermore, as prefetches need to be timely, predicting only the next LBA and the requested I/O size is not sufficient, and it is required to predict several accesses into the future. Finally, to support dynamically changing workloads, we evaluate our proposed address mapping learning technique to determine whether prefetching models can learn generalized patterns within complex I/O access patterns.

### 4.4.1 Data Preparation for Reducing the Output Label Space

We preprocess the input dataset to address the problem of large logical block address space. The number of unique memory addresses within an SSD is typically of the order of billions, rendering a separate class for each memory address impractical. To reduce the address space, we take the $l_1$ norm of each pair of consecutive LBAs (LBA delta). For example, if consecutive I/O accesses starting from LBA 10000 are requested as 10001, 10003 and 100006, the corresponding LBA deltas were recorded as 1, 2, and 3, respectively. This significantly reduces the number of classes that our model needs to predict. We identify the top 1000 frequently oc-

curring LBA deltas and assign each one of them to a class in decreasing order of frequency. All remaining LBA deltas are assigned to a separate class representing a "no prefetch" operation, thus limiting the number of classes for model to predict to 1001. The reason for choosing LBA deltas over actual addresses is to increase the coverage of LBA deltas in the data. For example, for Microsoft Research Cambridge traces [147] (MSR_1), the top 1000 most frequently occurring LBAs covered only 2.77% of all the LBA accesses, whereas the top 1000 most frequently occurring LBA deltas covered 91.66% of all LBA accesses. The coverage of top 1000 frequent LBA deltas for the datasets used in this study ranged between 54% and 92%, as seen in Table 1. Expanding the number of classes to beyond 1000 is possible with more computational power, however, for our datasets, we chose 1000, as it provides a considerable coverage for LBAs and is a sufficiently large size to prove the practicality of our approach.

The requested I/O sizes for the analyzed real world applications ranged from 4KB to several MBs with up to 10,000 different I/O sizes for an individual application. In order to reduce the number of possible target I/O size values, we round off each observed I/O size to the nearest number that is a power of 2, $2^n$, and use $n$ as an I/O size class. This reduces the number of possible target I/O sizes for most applications to 16 while still supporting requests of size up to 64MB. A limitation of this approach is that, in the worst case, roughly twice as many as required 4KB blocks may be prefetched from the SSD.

### 4.4.2 Model Architecture

We designed our proposed neural network model to predict both the I/O size and LBA deltas at the same time. The model has two separate input layers, one for I/O size and one for

LBA delta, where each input layer is an embedding layer [148] consisting of 500 neurons. The inputs to the model are categorical, one-hot, representation of the two features, LBA deltas and I/O size, each being fed to a separate embedding layer. The model has two hidden LSTM layers, where each LSTM layer has 500 hidden nodes. The outputs of the two embedding layers are first concatenated and then fed to the shared LSTM layers. The final output layer is split into two branches, where each branch is a dense layer consisting of softmax [149] nodes. The number of neurons in the LBA delta output layer is 1001, representing top 1000 LBA deltas and a "no prefetch" LBA delta, and the number of neurons in the I/O size output layer ranged between 12 and 20, depending on the I/O sizes present in each dataset. The model architecture is shown in Figure 4.2. The number of neurons in each of the first three layers of the model was set to 500 to ensure a good representation of input features, and we used a dropout [150] of 0.2 to prevent overfitting of the model. Having an initial embedding layer facilitates better representation of the input features and helps the subsequent LSTM layers to learn effectively from sequential data.

### 4.4.3 Timeliness

As discussed in Section 4.3, a prediction from the prefetcher is timely only if the following equation holds: $T_{cand} + T_{read} < PA * T_{demand}$. We empirically determined $T_{cand}$ to be

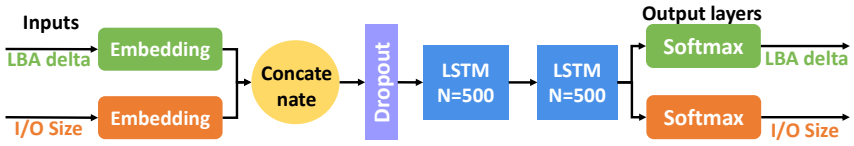


Figure 4.2: Model architecture

734$\mu$s by measuring the inference latency of our model. We measured the latency of accessing an Intel P3600 NVMe based SSD using the flexible I/O tester (FIO) [151] to be 300$\mu s$ on average under 80% workload. For the traces that we examined, the average time between two successive I/O requests ranged between 800$\mu$s and 1200$\mu$s, and the minimum time was 10$\mu$s. As a result, a good *PA* value is in the range of $5 > PA > 100$. We evaluate a range of *PA* values and its impact on prediction accuracy in Section 4.6. Predicting further ahead in the future typically reduces the accuracy due to the increased uncertainty. We find that, in order to increase the accuracy in case of a high *PA* value, training the model with longer history of sequences can improve performance.

### 4.4.4 Address Mapping Learning

Different workloads show similar I/O access patterns due to shared design patterns and commonly used data structures. For instance, array-based data structures used by applications generally entail sequential I/O access patterns. Furthermore, as most applications leverage the same underlying filesystem, it is likely that I/O accesses show common patterns. An ideal prefetcher would be trained once, on a varied set of applications, providing high performance even for previously unseen applications. Such a prefetcher is also likely to be more robust with respect to dynamically changing data inputs or code changes to the original application.

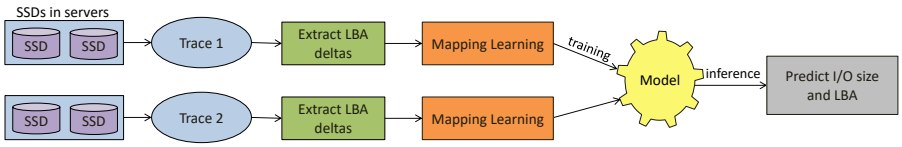


Figure 4.3: Block diagram of the Address Mapping Learning process

To test the idea that applications share common patterns that can be learned, we train the model on traces from one dataset (source) and evaluate the performance of the prefetcher on another dataset (recipient). The mapping of addresses to labels is done by sorting the frequency distribution of LBA deltas from *both* the source and recipient traces and assigning them labels in decreasing order of frequency of occurrences. We call the process of extracting the LBA deltas, training the model on source dataset, and using the model to predict LBA deltas and I/O sizes for the recipient dataset as Address Mapping Learning (AML) and present the block diagram of this process in Figure 4.3.

## 4.5 Methodology and Experimental Setup

### 4.5.1 Model Training

For our experiments, we used a total of 10 block-level I/O traces from three different sources running applications in live production servers. The datasets included traces describing enterprise storage traffic in commercial office virtual desktop infrastructure (VDI) [147], as well as traces from live production servers at Microsoft SNIA [152] and Microsoft Research Cambridge [153]. We did not use any synthetic benchmarks, as used in previous work [141, 131], as those traces do not accurately represent the complexity and interleaved patterns exhibited in real applications. The utilized trace files are open-source and can be obtained online [154], [155], [152].

Table 4.1 provides information about the datasets used in this study. From the table, we see that the coverage of top 1000 LBA deltas is consistently higher than direct memory

Table 4.1: Dataset Description

| Trace Source | Dataset Name | Represented Name | Num obs | Coverage Offset (%) | Coverage LBA Delta (%) |
|---|---|---|---|---|---|
| VDI | 2016022315.csv | VDI_1 | 5226120 | 58.76 | 66.96 |
| VDI | 2016030817.csv | VDI_2 | 4443487 | 63.94 | 70.08 |
| VDI | 2016030819.csv | VDI_3 | 2902328 | 68.94 | 69.8 |
| VDI | 2016031115.csv | VDI_4 | 2408227 | 68.65 | 72.35 |
| MSR | proj_3.csv | MSR_1 | 2244642 | 2.77 | 91.66 |
| MSR | mds_0.csv | MSR_2 | 1211034 | 63.46 | 76.94 |
| MSR | src1_1.csv | MSR_3 | 45746222 | 28.6 | 77.7 |
| MSR | usr_1.csv | MSR_4 | 45283980 | 2.64 | 82.12 |
| Microsoft | buildserver-2.csv | MS_1 | 1600430 | 2.77 | 28.84 |
| Microsoft | buildserver-7.csv | MS_2 | 1714151 | 8.97 | 55.49 |

addresses (offset), and hence it was selected as one of the features for training the model. The datasets also contained other information such as the I/O size, response time, filename, file location, etc. In this work, we only used the timestamp, offset (LBA), and I/O size as features. We trained our model using Google's Tensorflow [156] library on a Intel Xeon server with 8 CPU cores running at 1.7 GHz containing 96 GB of DRAM. The server also had 4 NVIDIA Tesla 2080TI GPUs for training the model. We split the dataset into training and test set, where the training set contained the first 70% of the I/O accesses, and the test set contained the last 30% of the I/O accesses. The sequence of LBA deltas, ordered by timestamps, is fed to the model for training. For all the experiments, we trained our model using Adam optimizer [157] with a cross-entropy loss function, and a learning rate of $10^{-3}$ for up to 1000 epochs, and stopped model training if there was no improvement in validation loss, with validation loss not decreasing by at least $10^{-5}$ for five consecutive epochs.

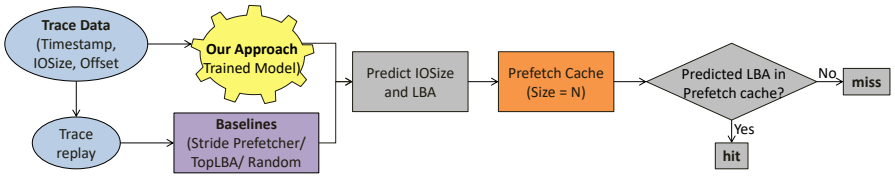


Figure 4.4: Block diagram of the evaluation process using our simulator

### 4.5.2 Prefetcher Simulation Environment

To enable the comparison of our prefetcher against prior baselines, evaluating only recall and precision is not sufficient. As motivated before, analyzing the prefetcher's timeliness is required to evaluate the end-to-end performance gains of prefetching, as even the most accurate prefetcher will not improve the performance if it lacks timeliness. As shown in Section 4.4.3, in order to compensate for the model's prediction latency and the latency to perform a read from the SSD, it is required to generate predictions ahead of time (*PA*). We evaluate the end-to-end performance as follows. As we iterate through the test dataset, the evaluation models continuously generate prefetch candidate predictions that are inserted into the cache.

Every I/O access is checked against the cache to see if the LBA is present, where the access is recorded as a hit, otherwise it is recorded as a miss. We utilize the Least Recently Used (LRU) [158] eviction policy for our experiments. The architecture of the simulator is presented in Figure 4.4. We choose variable cache sizes of LBAs for the stride, Markov-based, and our proposed prefetcher, and run experiments to provide a comparative study in Section 4.6.

### 4.5.3 Baselines

We compare our proposed prefetcher to three baselines. The first, naive prefetch-er, baseline always predicts the most common delta of a trace. The second baseline implements a Markov chain predictor [141], [131]. This method treats each LBA access as a state and predicts the next LBA based on the previous state by computing a probability distribution over the probabilities of transition from one state to another. The third baseline is a stride prefetcher which is commonly used in software and hardware systems. The stride prefetcher concurrently observes 128 I/O access streams. Each access is mapped to a stream based on hashing the most significant bits of the LBA. For each stream, the stride prefetcher tracks the last three I/O accesses. If the difference between the three I/O accesses match, the prefetcher detects a stride and prefetches the next access. Note that the stride prefetcher's results are optimistic, as it only prefetches one access ahead of time and does not compensate for timeliness. In the next section, we evaluate our proposed prefetcher in terms of prediction accuracy, timeliness, and capability to generalize to different workloads.

## 4.6 Results

### 4.6.1 Prefetcher Accuracy, Precision and Recall

Table 4.2 shows the comparative performance of our neural network based pre-fetcher against the three chosen baselines. The table lists the dataset name, number of samples in the dataset, and the accuracy for the three chosen baselines, Naive prefetcher, Stride prefetcher, and Markov chain based prefetcher. For our approach, we provide the accuracy, precision, and recall

results. For each sample, our prefetcher predicts both LBA and I/O size in increments of 4KB blocks, as the minimum block size for a drive operation in SSD is typically of 4KB size [137]. We only count the actual blocks that are correctly prefetched. For each data sample, we prefetch only the top predicted LBA and I/O size using the prediction with the highest confidence. We used a batch size of 64, look back of 64, and predict-ahead of 64 in this experiment. Each prefetcher has a cache size of 1000 for this experiment. In the next section, we present a more detailed analysis of the impact of cache size on the performances of the prefetchers.

As shown in Table 4.4, our proposed prefetcher consistently outperforms all three baselines delivering up to 11× improvement over the Markov chain based prefetcher using Microsoft SNIA traces with the same cache size (Dataset: MS_2). For VDI traces, our proposed prefetcher achieves the highest accuracy, providing 800× improvement over the stride prefetcher (Dataset: VDI_4). Our prefetcher also achieved the highest precision and recall compared to the baselines. The Markov chain based prefetcher performed considerably worse compared to our prefetcher, with the accuracy ranging between 7% and 25%, performing even worse than the Naive prefetcher in several cases (Dataset: MS_1, VDI_1, VDI_2, MSR_4).

### 4.6.2 Impact of Cache Size, Look-Back, and Predict-Ahead

In this section, we present an analysis of the impact of look back, predict-ahead, and cache size on our proposed prefetcher's performance. In order to ensure the availability of data in the cache when the data block is requested, we trained the model to predict $N$ steps ahead for varying values of $N$, and evaluated the performance of the prefetcher. Higher values of $N$ typically resulted in lower accuracy due to the increased uncertainty in predicting further

Table 4.2: Performance comparison of Our proposed prefetcher against baselines

| Dataset Name | No. Samples | Naive Prefetcher | Stride Prefetcher | Markov Prefetcher | Our (Accuracy) | Our (Precision) | Our (Recall) |
|---|---|---|---|---|---|---|---|
| VDI_1 | 5226120 | 0.17 | 0.01 | 0.09 | 0.73 | 0.76 | 0.71 |
| VDI_2 | 4443487 | 0.21 | 0.01 | 0.07 | 0.59 | 0.75 | 0.49 |
| VDI_3 | 2902328 | 0.19 | 0.02 | 0.12 | 0.66 | 0.73 | 0.57 |
| VDI_4 | 2408227 | 0.21 | 0.05 | 0.09 | 0.73 | 0.77 | 0.69 |
| MSR_1 | 2244642 | 0.14 | 0.01 | 0.21 | 0.41 | 0.66 | 0.31 |
| MSR_2 | 1211034 | 0.09 | 0.21 | 0.17 | 0.49 | 0.65 | 0.33 |
| MSR_3 | 45746222 | 0.12 | 0.001 | 0.16 | 0.79 | 0.89 | 0.46 |
| MSR_4 | 45283980 | 0.33 | 0.007 | 0.15 | 0.53 | 0.66 | 0.38 |
| MS_1 | 1600430 | 0.27 | 0.02 | 0.25 | 0.63 | 0.79 | 0.53 |
| MS_2 | 1714151 | 0.41 | 0.003 | 0.07 | 0.77 | 0.83 | 0.61 |

ahead in the future, while improving timeliness. To improve our prefetcher's predict-ahead performance, we found that it is necessary to increase the look back size for increasing values of *PA*, where, as described in Section 4.4.3, good values for *PA* are in the range of $5 < PA < 100$. Low values ($< 5$) of PA result in cache misses as the data cannot be fetched soon enough, whereas higher values of PA ($> 100$) result in untimely predictions as the data gets evicted before requested. Table 4.3 shows the performance of our prefetcher for different values of *PA* showing the accuracy of predicting the LBA and I/O size, as well as the cache hit ratio (Net Hit ratio). We measured accuracy as the actual number of 4KB data blocks that were correctly prefetched for three different values of *PA*, 32, 64, and 128.

In general, the accuracy of predictions decreases as we predict further ahead, producing the worst performance when predicting 128 samples ahead. For MS SNIA traces, the performance was comparable for *PA* equal to 32 and 64, and the accuracy degraded significantly for *PA*=128, whereas for VDI and MSR Cambridge traces, the performance degradation was gradual. These results show that our approach is successful in prefetching SSD accesses, as *PA* equal to 32 or 64 is generally sufficient to ensure timeliness in real-world settings. Nev-

Table 4.3: Impact of different predict values on our prefetcher performance

| Dataset | Predict Ahead = 32 | | | Predict Ahead = 64 | | | Predict Ahead = 128 | | |
|---|---|---|---|---|---|---|---|---|---|
| | Accuracy (LBA) | Accuracy (Size) | Net Hit ratio | Accuracy (LBA) | Accuracy (Size) | Net Hit Ratio | Accuracy (LBA) | Accuracy (Size) | Net Hit Ratio |
| VDI_1 | 0.72 | 0.65 | 0.71 | 0.69 | 0.65 | 0.73 | 0.42 | 0.6 | 0.33 |
| VDI_2 | 0.76 | 0.51 | 0.58 | 0.64 | 0.51 | 0.59 | 0.41 | 0.42 | 0.29 |
| VDI_3 | 0.73 | 0.88 | 0.69 | 0.48 | 0.88 | 0.66 | 0.42 | 0.67 | 0.37 |
| VDI_4 | 0.71 | 0.66 | 0.71 | 0.71 | 0.66 | 0.73 | 0.32 | 0.34 | 0.31 |
| MSR_1 | 0.65 | 0.49 | 0.41 | 0.65 | 0.49 | 0.41 | 0.34 | 0.19 | 0.29 |
| MSR_2 | 0.59 | 0.69 | 0.49 | 0.59 | 0.69 | 0.49 | 0.19 | 0.61 | 0.33 |
| MSR_3 | 0.95 | 0.67 | 0.66 | 0.91 | 0.61 | 0.79 | 0.13 | 0.61 | 0.19 |
| MSR_4 | 0.59 | 0.77 | 0.51 | 0.49 | 0.77 | 0.53 | 0.49 | 0.47 | 0.28 |
| MS_1 | 0.93 | 0.67 | 0.61 | 0.93 | 0.52 | 0.63 | 0.62 | 0.52 | 0.49 |
| MS_1 | 0.89 | 0.71 | 0.73 | 0.88 | 0.69 | 0.77 | 0.57 | 0.69 | 0.47 |

Table 4.4: Impact of cache size on the accuracy of our and two baseline prefetchers

| Dataset Name | Cache Size = 10 | | | Cache Size = 100 | | | Cache Size = 1000 | | |
|---|---|---|---|---|---|---|---|---|---|
| | Markov Prefetcher | Stride Prefetcher | Our Prefetcher | Markov Prefetcher | Stride Prefetcher | Our Prefetcher | Markov Prefetcher | Stride Prefetcher | Our Prefetcher |
| VDI_1 | 0.05 | 0.001 | 0.68 | 0.05 | 0.001 | 0.69 | 0.09 | 0.011 | 0.73 |
| VDI_2 | 0.05 | 0.0001 | 0.55 | 0.05 | 0.0001 | 0.55 | 0.07 | 0.0015 | 0.59 |
| VDI_3 | 0.04 | 0.0001 | 0.64 | 0.04 | 0.0001 | 0.64 | 0.12 | 0.0014 | 0.66 |
| VDI_4 | 0.01 | 0.006 | 0.7 | 0.01 | 0.006 | 0.71 | 0.09 | 0.005 | 0.73 |
| MSR_1 | 0.12 | 0.00005 | 0.39 | 0.12 | 0.00005 | 0.39 | 0.21 | 0.0011 | 0.41 |
| MSR_2 | 0.09 | 0.1 | 0.41 | 0.09 | 0.1 | 0.41 | 0.17 | 0.21 | 0.49 |
| MSR_3 | 0.07 | 0.0002 | 0.75 | 0.07 | 0.0002 | 0.76 | 0.16 | 0.001 | 0.79 |
| MSR_4 | 0.06 | 0.0005 | 0.51 | 0.06 | 0.0005 | 0.51 | 0.15 | 0.007 | 0.53 |
| MS_1 | 0.16 | 0.004 | 0.57 | 0.16 | 0.004 | 0.57 | 0.25 | 0.02 | 0.63 |
| MS_1 | 0.02 | 0.0003 | 0.71 | 0.02 | 0.0003 | 0.71 | 0.07 | 0.003 | 0.77 |

ertheless, to support upcoming storage devices that support even higher request ratios, reducing the inference latency and predicting even further ahead will be required.

Table 4.4 presents the impact of varying cache size on our prefetcher's performance. The table shows the accuracy of our approach compared to the Markov and Stride prefetchers for cache sizes of 10, 100, and 1000 LBAs, respectively. From the table, we can see that our prefetcher consistently outperforms the baselines for each cache size, and the performance improvement using VDI traces is as high as 800× over the Stride prefetcher, and 8× over the Markov prefetcher (Dataset: MS_1) . While the baselines show marginal improvements using larger cache sizes, our prefetcher benefits significantly from a larger cache size. This

suggests that while our prefetcher provides high accuracy and coverage, its timeliness can still be improved. For a large cache, prefetched blocks remain in the cache for a longer time and hence, prefetching exactly at the time when the LBA is requested is less important. Achieving perfect timeliness would require adjusting *PA* dynamically, as the inter-arrival time between requests varies at runtime.

### 4.6.3 Evaluation of Address Mapping Learning

In this section, we evaluate whether our prefetcher can learn common patterns among workloads to predict accesses for previously unseen workloads. In the previous sections, we obtained the training and test datasets from different portions of the same workload and the 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.

Table 4.5 shows the prediction accuracy for different types of sources. We show the accuracy of the model when it is trained and tested on *similar* source traces, and also when it is trained and tested on the *dissimilar* source traces. In Table 4.5, for our proposed AML technique, the model is trained on the source trace and tested on the recipient trace. For instance, when training on MS_1 and evaluating on MS_2 trace files, the accuracy of our address mapping approach is 84% which is only 3% less than training and evaluating both on MS_2 (fourth column). The overall effectiveness of AML depends on the frequency distribution of LBA deltas in the two datasets. The results in Table 4.5 show that our approach can be applied

Table 4.5: Performance of Address Mapping Learning (AML)

| | Similar Source | | | | | Dissimilar Source | | |
|---|---|---|---|---|---|---|---|---|
| Source Trace | MSR_3 | MSR_1 | MS_1 | VDI_1 | VDI_3 | MSR_3 | MS_1 | VDI_4 |
| Recipient Trace | MSR_2 | MSR_4 | MS_2 | VDI_2 | VDI_4 | VDI_3 | VDI_1 | MSR_2 |
| Accuracy on Source Trace | 0.95 | 0.63 | 0.93 | 0.75 | 0.87 | 0.92 | 0.92 | 0.82 |
| Acuracy on Recipient Trace | 0.59 | 0.59 | 0.87 | 0.72 | 0.75 | 0.75 | 0.75 | 0.72 |
| AML Accuracy | 0.37 | 0.39 | 0.84 | 0.52 | 0.47 | 0.31 | 0.22 | 0.35 |

to diverse workloads, as long as they share some similar characteristics. This increases the practicality of our approach, as we can train specific models for various workloads, and expect at least a moderate increase in performance for other workloads.

## 4.7 Conclusion

In this paper, we showed how to leverage neural network models to predict future storage I/O accesses to improve SSD performance via prefetching. We addressed several challenges such as the large and sparse logical block address space, ensuring timeliness of prefetching, predicting both the address and size of I/O accesses, as well as the challenge of training predictive models that can generalize across different workloads. We achieved generalization across workloads by leveraging a large set of real world cloud application traces. We compared the performance of our prefetcher to existing techniques and used an in-house simulator developed to test the accuracy, coverage, and timeliness of our proposed prefetcher. Our proposed model outperforms prior approaches such as the stride prefetcher by up to 800$\times$ and Markov chain based prefetcher by up to 8$\times$.

## 4.8 Publications

Research papers summarizing project findings were accepted at European conference on Machine Learning, 2020 (Learning I/O Access Patterns to Improve Prefetching in SSDs) and Symposium on Cloud Computing, 2018 (SSD qos improvements through machine learning).

# Chapter 5

# Reducing Write Amplification in SSDs using Machine Learning

## 5.1 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 100$\mu$s read latency. However, SSDs suffer from limited endurance due to wear out. In particular, the existing 3D NAND and quad-level cell (QLC) based SSDs support between 5000-50000 write/erase cycles [159], which if exceeded, may result in data loss. Thus, it is imperative to minimize the number of writes applied to a flash storage 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 [160], 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.

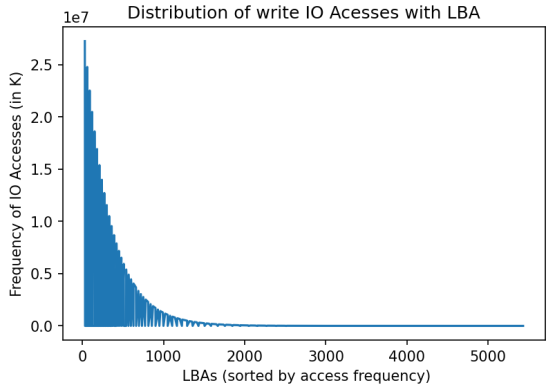


Figure 5.1: LBA write frequency distribution for VDI

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 follows a highly skewed distribution, as shown in Figure 5.1 for a virtual desktop trace (VDI) application. As a result, prior work [161], [162], [163], [164], [10], [165], [166, 167] 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, death-time is defined as the number of I/O writes after which an LBA will be overwritten in the future, and by grouping LBAs with similar death-times within the same block, write amplification can be eliminated. *Oracle-DT* requires perfect future knowledge about death-times and a potentially large number of concurrently opened blocks. As these requirements are impractical, we 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

## 5.2 Research Questions

The following lists the research questions we are looking to answer in this project.

### 5.2.1 RQ A: Can sequence-to-sequence deep learning models learn the death-time patterns of logical block addresses in real-world applications?

As mentioned earlier, real-world applications usually have complex workload patterns due to the mix of sequential and random IO requests from multiple users. Additionally, the data in an SSD is often fragmented due to the inherent inability of flash chips to support in-place writes [168]. Due to this, an SSD needs to perform GC to provide free blocks for additional writes. In this thesis, we are interested in leveraging deep learning techniques to learn the spatio-temporal death-time patterns to predict death-time of an incoming LBA. This allows us to place pages with similar lifetime together and reduce garbage collection overhead.

### 5.2.2 RQ B: Can we design a data placement policy for optimizing GC overhead, having perfect knowledge of future death-times?

We introduce a data placement strategy based on perfect knowledge of death times and compare the performance with existing data placement strategies. We show that our approach results in the near elimination of garbage collection overhead. We used this placement strategy while allocating blocks to incoming LBAs using predicted death times generated by the trained ML model.

### 5.2.3 RQ C: How does the performance of our machine learning-based data placement policy (ML-DT) compare with state-of-the-art techniques?

We compare the performance of SOTA data placement strategies with our approach and show that our approach can reduce GC by up to 14% on ten traces from three real-world applications.

### 5.2.4 RQ D: Can we use the learned death-time patterns to predict IO death time patterns in new, unseen workloads?

As discussed earlier, different workloads show similar I/O write patterns due to shared design patterns and commonly used data structures. We show that trained models on one source can be reused to predict death times on different applications with similar characteristics and discuss the situations when the approach is useful.

## 5.3 Prior Work on reducing Write Amplification (WA)

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

### 5.3.1 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 item. To bound the storage capacity of log-structured storage systems, garbage collection (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. 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 as a *closed-block*. As pages are overwritten, closed-blocks contain an increasing number of invalid pages. Finally, the GC mechanism cleans a block by moving all the valid pages to another block before erasing it, so that it can be added back to the list of free-blocks.

### 5.3.2 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 to clean 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 write amplification. 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 mechanisms do not incur any metadata storage overheads besides tagging each block according to its temperature.

### 5.3.3 Frequency-based approaches

Frequency-based approaches [159], [165], [164], [10] 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 [169] have been deployed to leverage this technique.

## 5.4 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 [170]. 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 5.2. 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 5.2, 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.

The fifth row in Table 5.2 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 5.2 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 5.6, 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.

| 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 |

Figure 5.2: Baseline vs. Frequency vs. Oracle-DT Policy

### 5.4.1 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 5.3 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 [171], RocksDB [171], and tpc-h benchmark (MonetDB)) [147], [172], from which we make the following observations.

1. Real-world workloads exhibit skewed write patterns (see Figure 5.1) where a significant number of I/O accesses are covered by only a few LBAs. As a result, the distribution of LBAs has a long tail and the corresponding death-times follow a similar pattern.

2. The death-times of LBAs vary widely, from a few I/Os to thousands of I/Os (see Figure 5.3). Separating I/O streams based on death-times enables the invalidation of pages

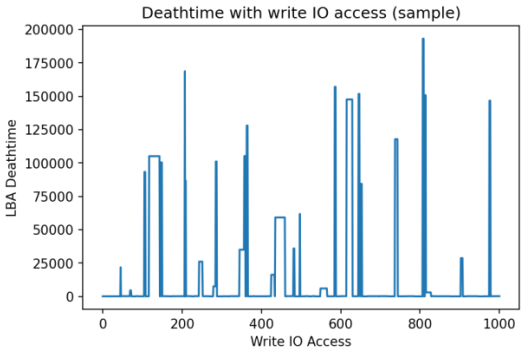


Figure 5.3: Death-times varying widely (sample)

within a block solely via user-writes.

3. User-writes originating from similar locations tend to have similar death-times. Hence, the LBAs of I/O accesses 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, generating interleaved write patterns. We noticed that the death-times are also interleaved.

5. For a given LBA, death-times often change over time. As a result, using prior history of assigning data to blocks can be inefficient, resulting in higher WA.

6. The requested I/O sizes for the analyzed real-world applications ranges 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.

### 5.4.2 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 writes. 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 [139] and GRUs [173], also have an attached memory which allows the models to look at recent previous accesses 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 [174] 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.

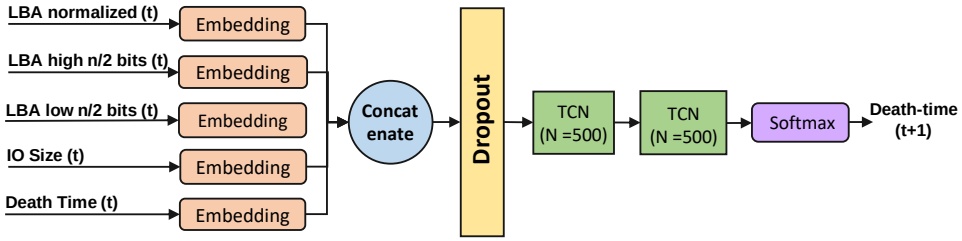


Figure 5.4: Model Architecture

**Model Architecture.** The proposed model architecture is shown in Figure 5.4. 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 [175] of 0.2 to prevent overfitting of the model. Death-time prediction is performed by a temporal convolutional network (TCN) [52], a variant of convolutional neural network (CNN) [176], employing causal convolutions [177] and dilations [178] 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 [179] 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 5.5.4 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 [139], SVM [180], and Random Forest [181] 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" [182] 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 [183]. 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 [178]. LSTM is chosen as a baseline as it is a popular DNN based technique used in time series forecasting in multiple

applications [180], and it also has a memory to look at recent data for handling sequential time series data. Random Forest [181] and SVMs [180] 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.

### 5.4.3 *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: $death\_time\_new = (page\_per\_block - write pointer)/100 * death\_time\_old$. 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

$$WA = UW/(UW + GW + PW).$$

## 5.5 Implementation

In this section, we discuss the implementation details of *ML-DT* and its integration into an SSD simulator. We also describe our dataset preparation technique. Then, we describe the experimental setup used for training the models and replaying the trace on a virtual SSD. The size of virtual SSD is adjusted based on each trace. More specifically, we adjusted the size of the SSD so that the normal capacity, excluding the over provisioning, can tightly fit the number of unique LBAs present in the trace. This ensures that GC needs to be performed multiple times during the experiments, causing write amplification.

### 5.5.1 Datasets and Data Preparation

We leverage traces from ten real-world workloads from three different sources for our experiments. We consider two features for input to ML models - LBA and I/O size. 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 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.

### 5.5.2 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 [52] models were significantly faster to train than comparable LSTMs [184], 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 $\mu$s. Data to the model was fed in batches of 64. We used *tanh* activation function [185] and Adam optimizer [108]. The models were trained for 10 to 32 epochs until convergence [186]. 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 5.6, we show that 20 open blocks are sufficient to minimize

WA for *ML-DT* .

### 5.5.3 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.

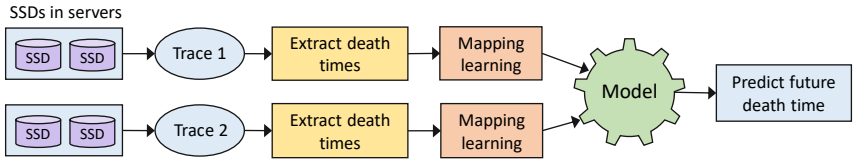


Figure 5.5: Mapping Learning Architecture

### 5.5.4 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 access patterns. Different workloads show similar I/O access patterns due to shared design patterns and commonly used data structures [9]. 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 accesses 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 access 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 5.5.

## 5.6 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.

| 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_sequential _random_50_50** | 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-additional | 97.2 | 0.33 | 1123568 | 8 | 16.5 | 17.9 | 69 | 59 | 43 | 33 |
| **TPC H benchmark (MonetDB)** | 12 GB | tpc-h-monet | 66 | 0.5 | 1332176 | 5.1 | 25.7 | 33.3 | 83 | 73 | 61 | 66 |

Table 5.1: Comparison of machine learning approaches for death-time range prediction.

### 5.6.1 Evaluation of ML models

As mentioned earlier, we used four baselines to compare our ML approach: Random forest (RF) [181], support vector machines (SVM) [180], DNN based LSTMs [139], 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 5.1 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*p$^{th}$ percentile of the data in each class where p=[1,2,3..19]. The table provides detailed characteristics of the traces [187, 147, 171, 141], 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 5.1.

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 5.6.4. The results in Table 5.1 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 accesses and random I/O accesses, respectively, and the third trace contained a mix of the 50% synthetic and 50% random workloads.

Figures 5.6 and 5.7 shows the accuracy comparison for the five ML techniques used in this work for varying output classes. For every experiment, as mentioned earlier, we reserved one open block for handling GC writes, hence N-1 classes were available for placing user writes.

From the figures, we can see that TCN-based approach consistently outperforms other baseline techniques for each case (N = 10, 15, 20, 25). 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). But, TCN-based models train faster

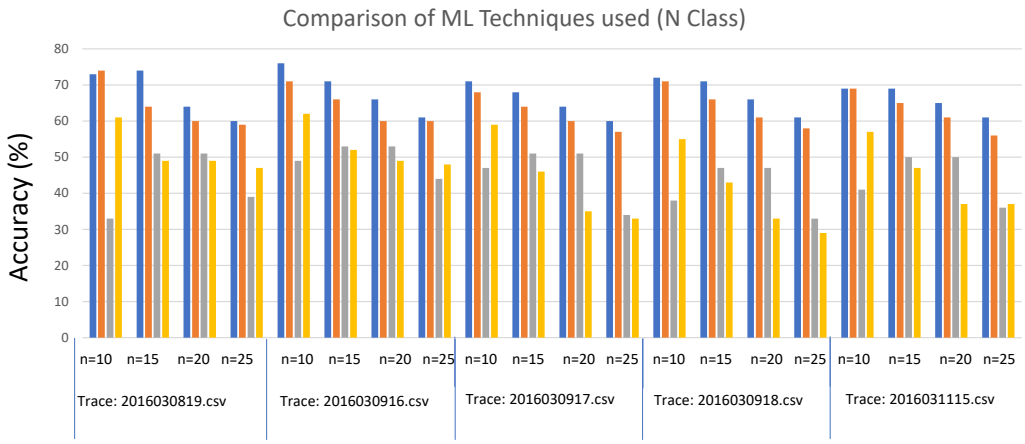


Figure 5.6: Comparison of ML Techniques for VDI traces (N Class)

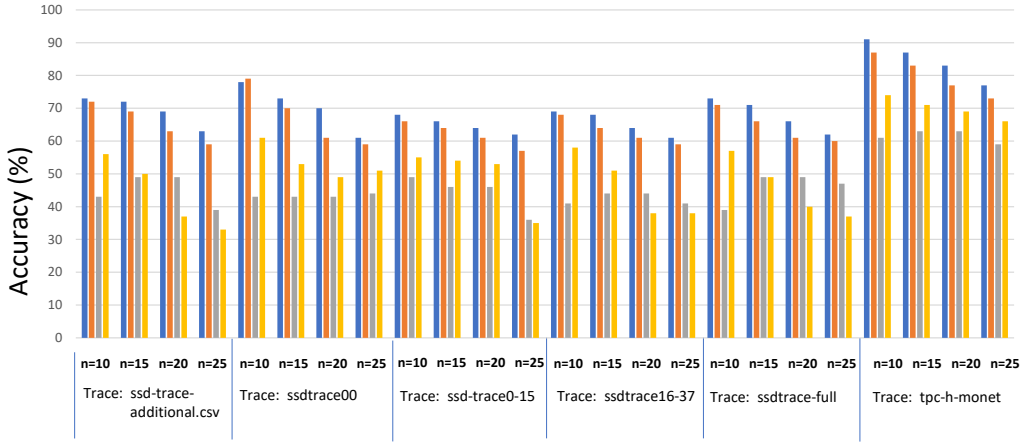


Figure 5.7: Comparison of ML Techniques for RocksDB and TPC-H traces (N Class)

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.

### 5.6.2 Guaranteeing no GC overhead with *Oracle-DT*

As mentioned earlier, given a large number of open blocks, *Oracle-DT* can guarantee no extra copying of live data thereby achieving a write amplification of 1. The assumptions are -

each LBA will have finite death-time and will invalidate at most one LBA for every IO. For each incoming write IO, we choose a block for data placement based on the death-time of the logical block address. The block is selected by performing a right shift operation with log of number of pages in a block (deathtime >>log(pages-per-block)). If the block number does not appear in the list of open blocks, we open a new block and place the data there. Right shift operation ensures the LBAs with similar death time ranges are placed within a block. For example, for an SSD with 64 pages per block, LBAs with deathtime 1-64 will be placed in block 1, 65-127 in block 2 and so on). Figure 5.8 describes the results of our experiments. For each trace, we report the source, the filename, the read-write ratio, the number of unique LBAs and the number of blocks needed to store the data directly (unique LBAs/page size). We experimentally find out the maximum number of open blocks used per trace using *Oracle-DT* and report the findings in maximum blocks column.

| Trace Source | File Name | R-W ratio | Num unique LBAs | User writes (millions) | GC writes | Max_Blocks | Num unique LBAs/page_size |
|---|---|---|---|---|---|---|---|
| Generated | synthetic_B | 0 | 262145 | 1.04 | 0 | 124970 | 8192 |
| | 2016030917.csv | 0.45 | 832659 | 2.47 | 0 | 66198 | 26020 |
| | 2016031115.csv | 0.15 | 430365 | 2.4 | 0 | 89411 | 13449 |
| | 2016030819.csv | 0.27 | 677630 | 2.9 | 0 | 80604 | 21175 |
| VDI | 2016030916.csv | 0.35 | 1049707 | 3.75 | 0 | 57633 | 32803 |
| RocksDB | ssd-trace00 | 0.22 | 223168 | 1.95 | 0 | 3541 | 6974 |

Figure 5.8: Block usage per trace with *Oracle-DT*

From Figure 5.8, we can see the maximum number of open blocks used depends on the IO write patterns of the input trace. Traces with higher write ratio and randomness of IO patterns (such as synthetic B or 2016031115) used higher number of blocks per write LBA compared to RocksDB traces (ssdtrace-00) which has higher coverage and more sequential IO write accesses. Specifically, for synthetic trace B, which has 100% random accesses and no read

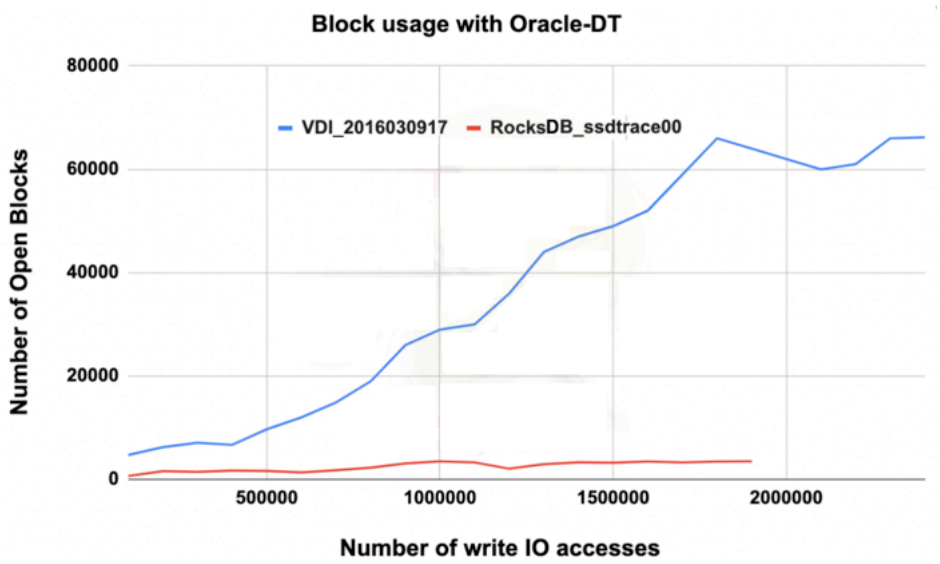


Figure 5.9: Block usage over time with *Oracle-DT*

IOs, the maximum number of blocks used was more than the number of blocks needed to store the data directly (num unique LBA/page size) indicating inefficiency of using this approach. On the other hand, for the traces sourced from VDI (2016030916) and RocksDB traces (ssdtrace-00), *Oracle-DT* used less blocks than needed directly to store the data. Nevertheless, in all cases, *Oracle-DT* was able to guarantee no garbage collection overhead. However, the usage of the number of blocks depends on the time as can be seen from Figure 5.9.

In Figure 5.9, we show the distribution of number of blocks used over time for *Oracle-DT* with two selected traces (RocksDB and VDI trace 2016030917.csv). In general, RocksDB trace used less number of blocks due to the increased ratio of sequential IO write accesses and lower read-write ratio. For VDI trace 2016030917, we see that although the traces used more blocks in peak condition, generally less number of blocks are used to store the data. This shows us that given a high number of open blocks and accurate death time information,

garbage collection can be eliminated. However, such a system might be impractical in real world conditions due to increased latency overhead of maintaining additional open blocks.

### 5.6.3 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.

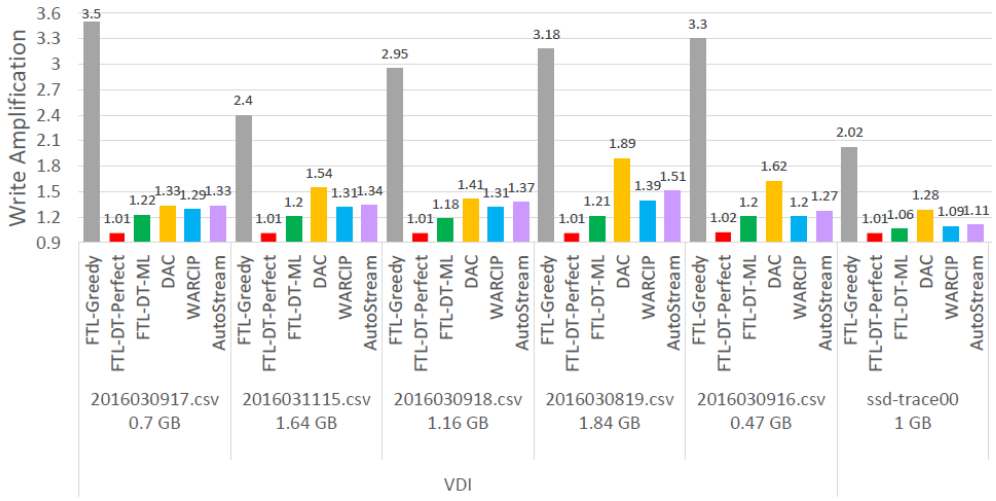


Figure 5.10: FTL comparison with baselines (VDI traces)

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.

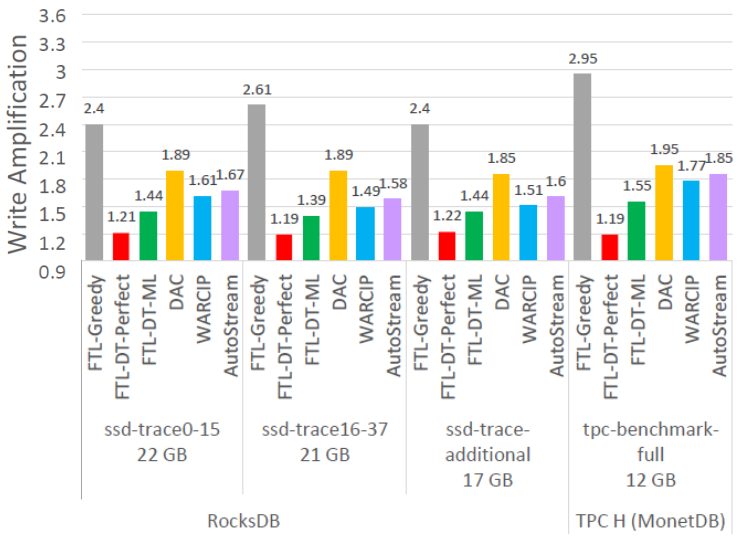


Figure 5.11: FTL comparison with baselines (RocksDB and TPC-H traces)

2. *Oracle-DT* : Has perfect knowledge of future death-times and uses the same placement policy as *ML-DT* .

3. Dynamic Data Clustering (DAC) [166]: 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 [167]: 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) [165]: 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.

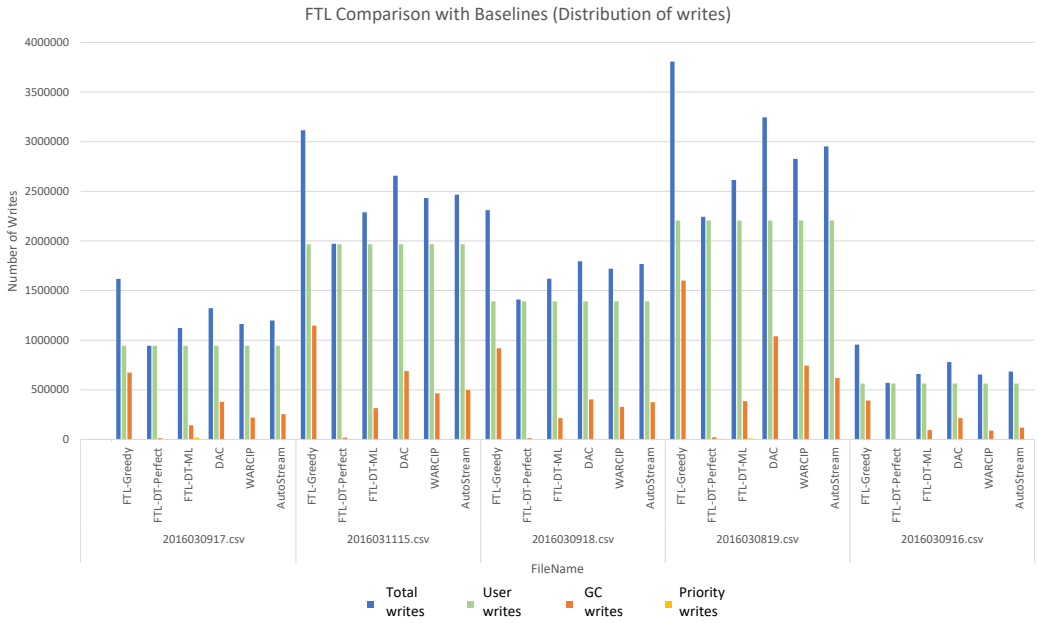


Figure 5.12: Distribution of writes comparison with baselines (VDI traces)

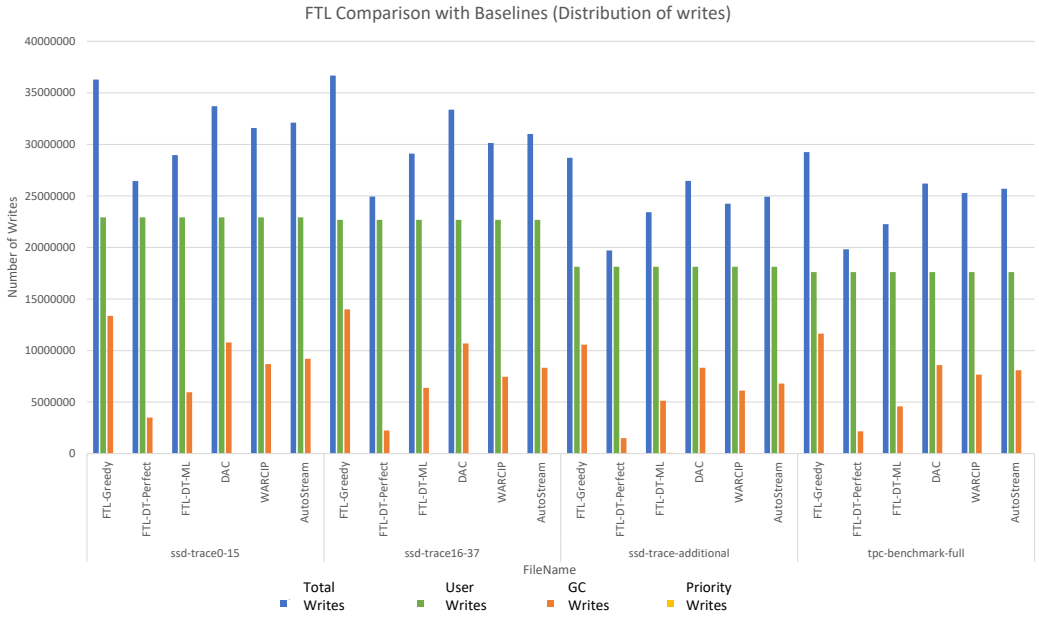


Figure 5.13: Distribution of writes comparison with baselines (RocksDB and TPC-H traces)

We present the comparative performance of our proposed *ML-DT* approach against the five baselines in Figures 5.10 and 5.11. For each FTL, we note the number of user-writes, GC-writes, and priority-writes (if applicable) to compute the WA and report the findngs in Figures 5.12 and 5.11. 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, 1,000, and 10,000 LBAs is also reported which shows 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. Figures 5.10 and 5.11 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. The distribution of total writes, user writes, GC writes and priority writes (if applicable) can be seen from Figures 5.12 and 5.13. 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 access 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.

### 5.6.4 Impact of number of open blocks

In this section, we study the impact of the number of open blocks available on WA. Figures 5.14, 5.15 and 5.16 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 (less than 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.

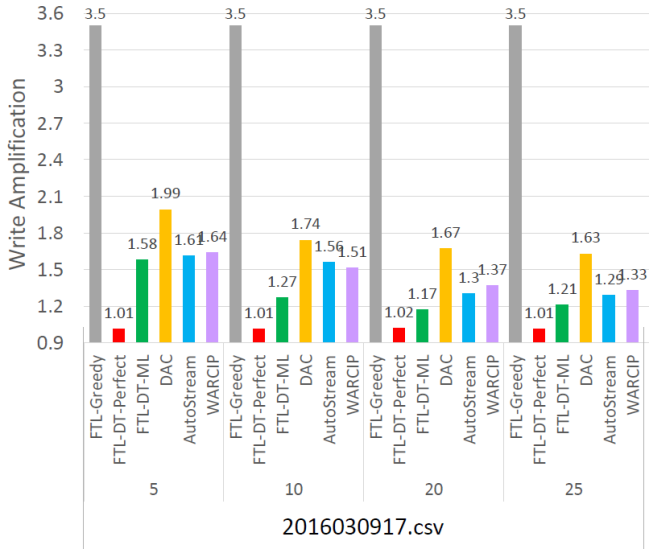


Figure 5.14: Impact of number of open blocks (VDI trace)

### 5.6.5 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

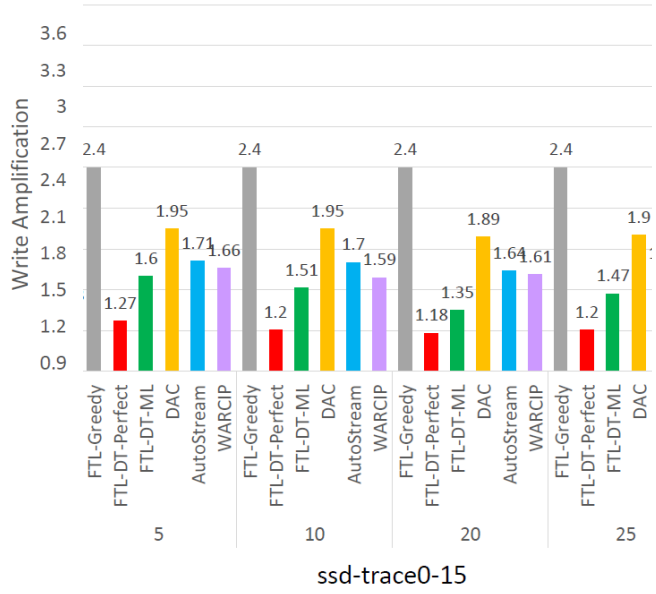


Figure 5.15: Impact of number of open blocks(RocksDB trace)

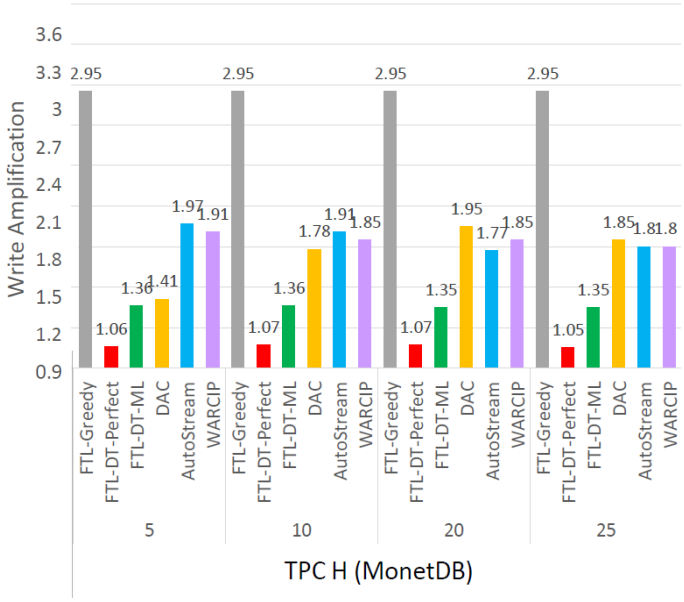


Figure 5.16: Impact of number of open blocks(TPC-H trace)

full resulting in the priority write overhead. The trend can be seen in Figures 5.17 and 5.18.

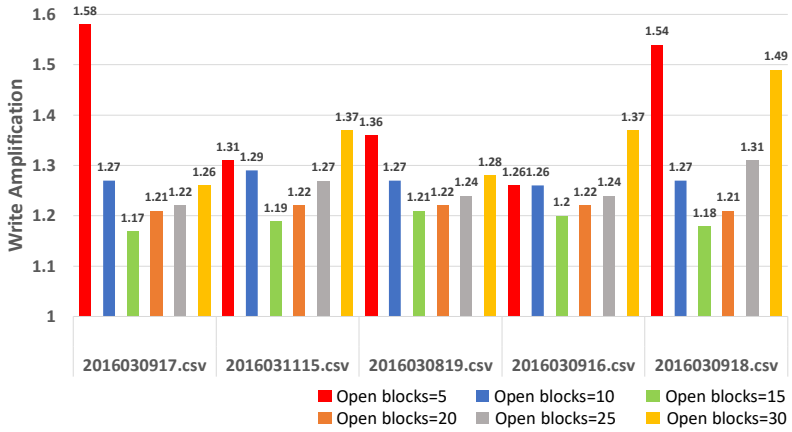


Figure 5.17: Sensitivity Study (VDI traces)

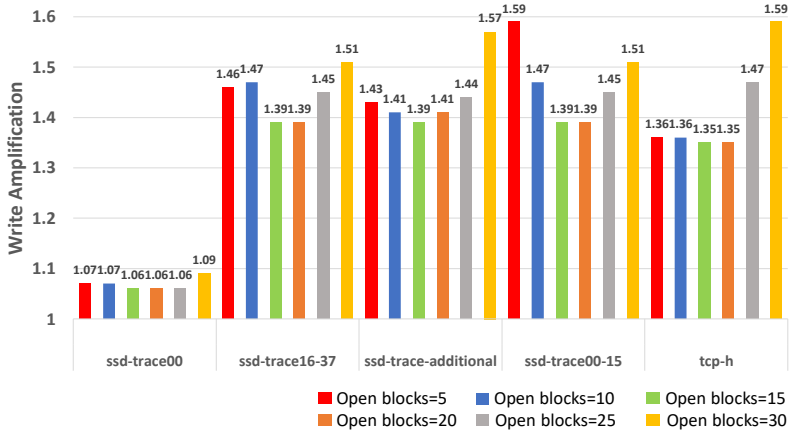


Figure 5.18: Sensitivity Study (RocksDB and TPC-H)

### 5.6.6 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 5.19 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). In this figure, 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 only 4% less than when training and evaluating the model on ssdtrace-16-37 trace file itself. 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 for the source trace. The number of prediction classes are kept the same in the two workloads is the 1-1 mapping is done is done sequentially. (For eg. Class 1 from 1st workload correspond to Class 1 of 2nd workload, Class 2 from 2nd workload correspond to Class 2 of 2nd workload, and so on. )

The overall effectiveness of Mapping Learning depends on the frequency distribution of data within each class in the two datasets. The results in Figure 5.19 show that our approach can be applied to diverse workloads, as long as they exhibit 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 for other workloads.

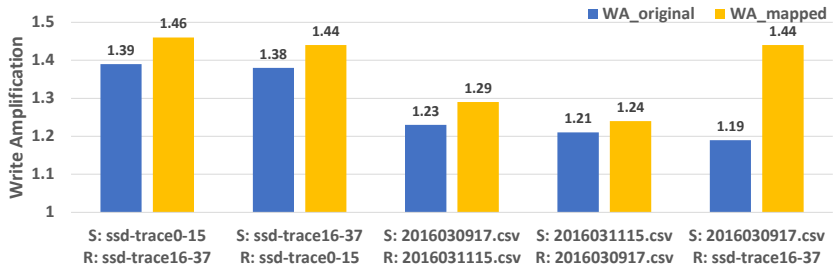


Figure 5.19: Mapping learning results using different Source (S) and Recipient (R) workloads

## 5.7 Conclusion

In this paper we introduced *ML-DT* , an ML based approach for reducing write amplification (WA) in log structured file systems by death-time prediction of logical block addresses (LBAs). We leveraged the time series nature of data in I/O accesses to train lightweight, yet powerful, 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 by death-time prediction of logical block addresses. Using the proposed data placement scheme, we present the design of *ML-DT* . We compare our approach with seven state-of-the-art data placement schemes and show that *ML-DT* achieves lowest WA by leveraging the learnt I/O write patterns from real-world storage workloads. Our approach results in up to 14% improvement in write amplification compared to the best baseline technique and generalizes better with increasing number of open blocks available. We provide insights on the type of workloads which receive most benefit using our approach. Finally, we present a mapping learning to test applicability of our approach to new unseen workloads and present a feasibility study to demonstrate the applicability of our work to unseen workload traces.

## 5.8 Publications

Research papers summarizing our project details was accepted at ACM International System and Storage conference, 2021 (Reducing write amplification in flash by death-time prediction of logical block addresses).

# Chapter 6

# Related Work

In this chapter, we provide related work on SSD failure prediction, neural network based prefetching techniques, and garbage collection methods.

## 6.1 SSD Failure Prediction

Anomaly detection techniques using both traditional machine learning [188] and deep learning techniques [189] have been successfully applied in various fields of research. Adewumi [190] provides a detailed review of deep learning-based techniques for fraud detection. A broad survey of deep anomaly detection (DAD) methods for cyber-intrusion detection is presented by Kwon [191]. An overview of DAD techniques for the Internet of Things (IoT) and big-data anomaly detection is introduced by Mohammadi and Mehdi [192]. Sensor networks anomaly detection has been reviewed by Ball [193]. The state-of-the-art deep learning-based techniques for video anomaly detection along with various categories have been presented in Kiran [194]. Other applications of anomaly detection include predicting failures in cloud sys-

tems [195], the medical domain [196], and self-driving vehicles [197] [198]. Zhang [77] introduced ATAD, a method of detecting anomalies in cloud systems, by training the model on one dataset and using transfer learning to use the model for another dataset. Our work contrasts in using a combination of several feature selection techniques to select the most relevant features for training the model for generating failure reasons.

In addition to anomaly detection applications, machine learning has been applied to improve the performance and efficiency of storage systems and SSDs [199, 142, 9, 200]. Failure of storage systems typically results in loss of data. In order to overcome the undesirable data loss, parity protection is implemented at a system level to improve system reliability [201]. However, studies have shown that parity protection provides benefits only when there is considerably low space utilization and low data access rates. Otherwise, it may result in higher write amplification and less efficient garbage collection with higher space utilization [202]. Even with parity protection, sudden failures of drives are undesirable, as they affect the overall reliability and performance of systems Ma et al. [63] studied the impact of RAID protection on disk failures and designed an original system, RAIDShield, for identifying the most vulnerable sections within disks. Considerable research has been done to improve the reliability of SSDs [203]. For example, Luo et al. [20] proposed a novel 3D NAND flash design with circuit-level and structure-level changes to NAND flash memory in order to improve the reliability of SSDs.

Traditional approaches for improving the reliability of SSDs focused on examining the failure trends within flash chips [204], [13], [59], [205], and determining various modes

of flash failures, such as power faults [204], program disturb errors [205], read disturb errors [13], including other errors [59]. Several studies measured the performance of SSDs under different workloads with different read/write characteristics. However, most of these studies were done in a controlled setting and using synthetic workloads, and hence, the results may not be directly applicable to real-world systems.

Earlier work on analyzing and predicting failures in storage systems focused on spinning disk drives [63, 70, 78, 206, 64, 62, 205, 65, 78, 206, 109]. Xu et al. [8] studied SSD failures focusing specifically on batch failures and repeat failures in drives and how human operators responded to them. Schroeder et al. [207] studied the reliability of DRAM and how it is affected by errors arising from external factors like memory utilization, chip density, temperature, memory technology, and DIMM age. Pinheiro et al. [62] collected disk information using SMART (Self-Monitoring and Reporting Technology) parameters to predict whether a disk drive is going to fail in the near future and used support vector machines (SVMs), clustering, and non-parametric statistical tests (rank-sum and reverse arrangements) to predict disk failures. Similar studies were done by Murray et al. [205], and they proposed an algorithm, mi-NB, based on multiple-instance learning framework and naïve Bayes classifier that was specifically designed to lower the number of false-positive cases.

In [79], the authors proposed a method for disk replacement using a predictive model that was trained on time series data collected from disks using SMART parameters. They used transfer learning and regularized greedy forest (RGF) classifier to predict future failures in disks with high accuracy. In [65], the author's used SMART information to characterize faulty disks

and used them to rank the disks based on their error-proneness in the near future. Hamerly et al. [78] used a mixture model of naïve Bayes sub-models that was trained using expectation-maximization. They used a naïve Bayes classifier to predict future disk failures and used features such as temperature, errors in addition to SMART parameters, etc. Zhang et al. [206] presented a system called "DeepView," that is specialized for localizing virtual hard disk failures using Lasso regression and tested their approach using the data collected from Microsoft Azure servers. However, most of these studies were limited to a small number of drives and using outdated trace data patterns because of the rapid increase in deployment of SSDs in real-world systems. Autoencoders have been shown recently to have shown promise in interpreting model predictions generated by DNN based models. Several techniques [117, 118] have been proposed to interpret the model decisions in multiple fields such as RNA sequencing [116], recommendation systems [115], and health monitoring systems [208].

However, due to the fundamentally different storage technologies, these prior results are not applicable to flash-based SSDs [209]. Furthermore, these prior studies were performed on a much smaller number of disk drives and hence were incompatible with ML techniques that require large training data sets. Pinheiro et al. [62] collected disk information using SMART parameters to predict whether a disk drive is going to fail in the near future and used support vector machines (SVMs), clustering, and non-parametric statistical tests to predict disk failures. Similar studies were done by Murray et al. [205], based on a modified naive Bayes classifier. Other Machine learning techniques (ML) [65, 78, 206, 109] have also been employed to predict disk failures. However, most of these studies were limited to a small number of drives

and using outdated trace data patterns because of the rapid increase in deployment of SSDs in real-world systems. Several studies have focused on providing statistics on long-term failure trends [8, 64, 72, 58, 64]. Meza [4] has explored SSD failure prediction based on a single feature (uncorrectable bit-error-rate), whereas our approach analyses a much more comprehensive set of 21 features. Schroeder [7] used supervised ML techniques (2-Class SVM and Random Forest) to predict sector failures. Alter [72] studied correlations between different workload conditions to study infant-mortality of SSDs within Google's data centers. We contribute over these works by proposing 1-class models improving prediction accuracy, providing adaptivity to previously unseen failures, and enabling interpretability of predictions.

## 6.2 Neural Network based Prefetching

Machine learning techniques have been applied to the prefetching problem in multiple domains such as web caching [210] and memory prefetching [142, 211]. [212] used a modified Greedy-Dual-Size-Frequency Caching Policy (GDSF) approach for efficient caching, taking into account frequency of access requests and an aging mechanism to deal with cache pollution. Researchers also examined Facebook photo caching patterns [213] and presented the correlation between content properties and the access patterns. The authors also demonstrated potential performance benefits using different eviction algorithms at both Edge and Origin layers.

In systems research, prefetch optimizations have also been proposed to improve performance [214] [215] [212]. Informed Prefetching and Caching (IPrC) [212] was proposed

for improving application response time by exploiting I/O and computation parallelism. In [215], the authors implement a file system that used a 2 level cache management strategy based on LRU-SP (Least-Recently-Used with Swapping and Placeholders) policy to allocate blocks to processes based on a controlled-aggressive policy. File access patterns have also been used to accelerate prefetching [216] using Partitioned Context Modeling (PCM) techniques. In [214], the authors presented an automatic application-specific file prefetching (AASFP) mechanism designed for improving the disk I/O performance of application programs based on file access patterns. Prior research in secondary storage prefetching mostly focused on disk drives [217] [218] [219] due to their popularity as a secondary storage device in the last two decades. In [217] the authors proposed an adaptive strip prefetching (ASP) scheme for striped disk arrays, which provides low prefetching cost and evicts prefetched data at the proper time by using differential feedback to maximize the hit rate of both prefetched data and cached data in a given cache management scheme. STEP – a Sequentiality and Thrashing dEtection based Prefetching scheme, was proposed based on an accurate cost-benefit analysis [219] of prefetch operations for Networked Storage Servers.

Data prefetching is vital to the performance of flash memory systems. Extensive research has been done to identify and exploit different types of correlation to improve prefetching. There have been studies [220], which showed several unanticipated aspects in the performance dynamics of SSD technology that must be addressed by system designers and data-intensive application users in order to effectively place it in the storage hierarchy. In [221], the authors introduced a multi-perspective reuse prediction, a technique that predicts the fu-

ture reuse of cache blocks using different types of features, and a bypass optimization. In [222] authors utilize instruction-based (PC) prediction to predict cache reuse distance, optimizing caching in the process. Reuse distance [223] has also been used as a prediction metric for efficient prefetching. In [224], the authors implemented a system called AMP ( Adaptive Multi-stream) used an Adaptive Asynchronous algorithm in a cache shared by multiple steady sequential streams to accelerate prefetching.

Some prefetching methods have been proposed for optimizing specific kinds of workloads. In [129], the authors present Tap - a storage cache optimized for sequential prefetching for improving the read-ahead cache hit rate and latency of system response. It dynamically adjusts prefetch cache size in order to maximize cache hit rate. Modha et al. [225] proposed another scheme that also used a dynamic cache size for sequential workloads but applicable to dynamic workloads as well comprised of multiple streams. Nilakant et al. [129] introduced PrefEdge, a prefetcher for graph algorithms that parallelize requests to maximize throughput from SSDs. PrefEdge combines the distribution of graph states between main memory and SSDs with a read-ahead algorithm to prefetch needed data in parallel. Data compression techniques have also been explored to accelerate prefetching [226].

Real-time workloads are typically not static, and special prefetchers have been introduced to handle such situations. Flashy prefetching proposed by [227] uses adaptive feedback-directed prefetching (FDP) techniques which can dynamically adapt to application needs based on accuracy, lateness, and pollution metrics. In [228], authors presented SARC (sequential prefetching in adaptive replacement cache) -a self-tuning, low overhead, simple to implement,

locally adaptive, novel cache management policy which dynamically and adaptively partitions the cache space amongst sequential and random streams so as to reduce the read misses. Adaptive Replacement Cache (ARC) [225], an online self-tuning cache management policy is another system that monitors the workload based on certain working rules and revises its characteristics automatically to improve caching.

Prior research work has also directly applied machine learning techniques to accelerate memory prefetching [142]. Prediction algorithms have been used to improve prefetching by leveraging spatial and temporal access patterns. Markov chains, in particular, have been shown to be promising [229]. Lynx, a system proposed by [141], uses an ML system based on Markov chains, which complement the Linux read-ahead prefetching system for both SSD performance model and new applications needs. In [229], the authors describe Markov predictor for memory prefetching, which acts as an interface between the on-chip and off-chip cache and works by prefetching multiple reference predictions from the memory subsystem. Other applications of machine learning techniques include [131], where the authors used a decision tree-based classifier to predict content to be prevented from entering the cache, thereby reducing unnecessary writes using a non-history-oriented approach. In [230], researchers designed a KM-Cluster-based pattern adaptive two levels prefetching mechanism for the last-level cache structure to support real-time big data management. Machine learning has also been used to predict certain contexts for increasing cache hit ratio. In intelligent cache [231], prefetch system includes a throttling scheme to monitor a cache hit rate context. Prefetch reads of additional data are only launched when the context is below a given threshold. [144] is another example

of a context-based prefetcher that uses semantic locality and reinforcement learning for memory prefetching. While previous work also utilized neural networks for determining prefetch candidates, they operate on very different datasets, as DRAM accesses differ significantly from I/O accesses. For instance, I/O accesses are not tagged with the source instruction for stream disambiguation, I/O accesses do not have a fixed size [232] and, in contrast to I/O, memory accesses are not intercepted by the OS.

Prior work on SSD prefetching utilized algorithmic approaches, typically using a data-range-table to detect usable strides and memory access streams [140]. Several variations of stride prefetchers have been proposed [233, 234] taking into account the spatial locality [234], feedback [235], and context [236]. However, as we showed in this work, algorithm-based prefetchers do not perform well on real-world applications due to their limited ability to learn complex patterns. The only prior research we are aware of that applies machine learning for prefetching in SSDs is based on Markov chains [131, 141], which we used as a baseline in this work. Finally, machine learning techniques have been applied to improve SSDs in other ways, for instance, by optimizing garbage collection [19], for predicting device failures [2, 237], for improving SSD virtualization [238], for managing SSDs in large clusters [239], and for improving the quality of service of SSDs [240]. These prior works are orthogonal to our work.

Artificial neural networks show great potential in accurate pattern prediction. It has a huge impact in image processing [241], audio processing [242], and natural language processing [243] . Recently, neural networks have been used in learning IO access patterns and have made great contributions in improving memory prefetching [142]. Prior research used deep

learning techniques for prefetching in DRAM, but they only accounted for the accuracy of the predictions. Our work, on the other hand, focuses on second flash storage and accounts for the timeliness of the predictions as well, and we present an analysis of how different parameters affect the performance.

## 6.3 Garbage Collection Optimization using Machine Learning

Garbage collection in flash devices in a well-researched area of storage systems [244], [19], [245] People have designed probabilistic and analytical models of SSD write performance which can compute how GC will affect performance of SSDs under certain conditions [246].

Several optimizations have been proposed to guarantee service response time in flash storage devices. Lazy-RTGC a real time real-time lazy garbage collection scheme proposed by [247] uses on-demand page-level address mappings, and partial garbage collection to guarantee system response time. DFTL [248] a modified FTL, which performs garbage collection in partial steps was proposed by [168]. It works by dividing the garbage collection of a single block into several chunks, thereby interleaving and hiding the garbage collection latency while servicing requests.

Work has also been proposed to reduce the garbage collection overhead in flash devices [249], [250] and [246]. In [249], the authors proposed a write pattern insensitive garbage collection technique based on the erasure interval, which is called EIGC (Erasure Interval-based Garbage Collection). Demand-based Flash Translation Layer (DFTL) [248] is another system which selectively caches page-level address mappings to reduce GC. Replacement policies such

as STGC (swap time-aware garbage collection) has been proposed which focuses on reducing the cleaning cost and improving the degree of wear-levelling. STGC calculates the cleaning index value of each block to select a victim block and the normalized value of the elapsed swap time of each valid page within the victim blocks to identify the hot valid page and cold valid page. A similar approach was proposed in [251] where the authors introduced two new GC algorithms dChoices GC algorithm, that selects d blocks uniformly at random and erases the block containing the least number of valid pages among the d selected blocks, and the Random++ GC algorithm, that repeatedly selects another block uniformly at random until it finds a block with a lower than average number of valid block. In [250], authors introduce FeGC a new garbage collection scheme that focuses on reducing garbage collection overhead by optimizing the number of erase operations for a flash device.

A workload adaptive flash translation layer (WAFTL) was proposed in [252] which can optimize garbage collection for dynamic workloads. WAFTL uses either page-level or block-level address mapping for normal data block based on access patterns. GC is also a well-studied problem in other areas of research. Pleco, a tool proposed by [253] used dependency information between users and cloud instances to remove unproductive instances in IaaS clouds. It constructed a weighted reference model based on application knowledge to find instances to garbage collect. MRJ [254], a MapReduce Java framework for multi-core architectures was used to auto tune the garbage collection process for Java MapReduce on multi-cores.

ML based techniques have also been proposed to optimize GC, although it is relatively a new field of research. In [255], the authors presented an efficient grey prediction model for

forecasting the future I/O workloads was proposed to determine the number of victim blocks that should be selected for evicting according to the predicted I/O workload. In [245] and [256] the authors propose a scheme to place data according to its predicted future temperature based on LSTMs. It also uses K-Means to do clustering and automatically dispatch similar "future temperature" data to the same NAND blocks.

Reinforcement learning has also been proposed to mitigate long tail latency problem of SSDs. In [245], authors introduced a reinforcement learning assisted garbage collection scheme which exploits fine grained GC to reduce the long-tail latency. Reinforcement learning has also been used to create an adaptive decision process [245].The process takes decisions regarding which garbage collector technique should be invoked and how it is applied based on information about the memory allocation behavior of currently running applications.

Prior works on reducing WA proposed greedy reclaiming policy [257, 258] 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 [169] have been introduced to expose multiple open blocks to applications. This enables applications to explore different placement policies for reducing WA. 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 [10] writes blocks into large segments in batches based on the write counts of an LBA divided by the block age. LOCS [259] 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 [163] automatically selects open blocks based on program counters in the Linux kernel. Extent-based identification (ETI) [260] 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) [164] also uses write frequency to allocate incoming writes to open blocks. It also takes into account the recency of 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 and thereby reduces WA over prior techniques.

Dynamic data Clustering (DAC) [166] assigns a temperature to each incoming write LBA and allocates segments based on different temperature levels. User-writes promote an LBA to a hotter segment while each GC-writes demote an LBA to a colder segment. Multi-Log [159] works similarly except that it keeps track of the update frequency to each LBA. It uses the update frequency to compute probability of assigning data to open blocks. AutoStream [165] 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 [170]. WARCIP [167] uses rewrite interval (i.e., 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 pages in a block. InferBIT [261] uses inferred the block invalidation time to minimize WA in log-structured storage by placing blocks with similar estimated BITs into the same group. In contrast to prior work, *ML-DT* leverages predicted death-time patterns

to assign writes to open blocks. We showed in Section 5.4 how *Oracle-DT* reduces WA over temperature-based techniques.

Novel FTL designs have also been proposed to address the high tail latency [262, 263] problem in flash systems due to GC overhead. Apart from flash memory, various work on reducing GC overhead has also been studied in virtual memory [264], file systems [10, 265, 266], RAID [267, 268] and distributed storage systems [269]. These prior works are tangentially related to our work.

# Chapter 7

# Conclusion

In this thesis, we presented three approaches of improving the response time, reliability and lifetime of flash based SSDs respectively. For improving the reliability of SSDs, we presented a machine learning based framework for predicting and interpreting drive failures. We showed that our proposed approach can effectively predict failures in SSDs with high accuracy and recall, thereby capturing all predicted failures. Furthermore, we introduced autoencoders to interpret the reasons for why the machine learning model flags a drive as likely to fail.

To reduce the response time of flash, we presented a DNN based prefetching framework that predicts future block accesses and preloads them into the main memory ahead of time. To achieve high performance, we addressed the challenges of prefetching in very large sparse address spaces, as well as prefetching in a timely manner by predicting ahead of time. We showed that our approach consistently outperforms the existing stride prefetchers by up to 800$\times$ and prior prefetching approaches based on Markov chains by up to 8$\times$ on multiple real-world applications running on data centers. Furthermore, we introduced an address mapping

learning technique to demonstrate the applicability of our approach to previously unseen SSD workloads.

We experimentally showed that our approach can reduce the number of extra writes required to store the data by 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.

# Chapter 8

# Acknowledgements

Pursuing a PhD has been a life changing journey for me, and I am deeply grateful for the support I have received from my family, mentors, and many, many, friends throughout. I am especially grateful to my parents, Sunil and Tapati Chakraborty for bearing all the hardships and sacrifices in life to give me a good life and education. I am also thankful my brother Indranil Chakraborty, for his inspiration and support.

I would like to thank my advisor, Heiner Litz, for leading me towards interesting and fruitful research directions. Heiner has inspired me in so many ways, particularly with his exceptional ability to recognize core research problems and by building a positive and collaborative culture in our research group. I have learned so much from our discussions about research and work life balance in academia. I would like to thank Jim Whitehead for his qualified advice and mentorship during my Masters, which helped me find my way at critical crossroads during graduate school. Jim knows exactly how to quickly identify the cruz of a problem, ask the right questions and give honest, wise advice. Special thanks to Ethan Miller, Adam Smith and Yang

Liu for their support and for being on my dissertation committee.

I am fortunate to have collaborated with brilliant colleagues in academia and industry on the work described in this dissertation. Thank you to Changho Choi, Vikas Sinha, Lokesh Jaliminche, and Arif Merchant for their valuable contributions to this work. I am proud of what we have accomplished together. I am grateful to the Memory Solutions Logic team at Samsung Semiconductor Inc. for hosting me for memorable summer research internships that inspired the topic of this dissertation. Thank you also to Stanford Pre Collegiate Summer Institutes, which provided me the opportunity to gain valuable teaching experience, and I discovered my passion for teaching.

Thank you also to all the members of the DataCenter Architecture Research Lab (RAD). Special thanks to Arif Merchant from Google for providing access to the Google traces and helpful feedback on the manuscript. We also thank the anonymous reviewers for their comments. My thesis work was generously supported by Samsung Semiconductor Inc. and NSF grant #1942754.

# BIBLIOGRAPHY


[1] Britesun, "[graph embedding techniques, applications, and performance: A survey." [Online]. Available: https://blog.csdn.net/qq_34807908/article/details/86595632

[2] I. Narayanan, D. Wang, M. Jeon, B. Sharma, L. Caulfield, A. Sivasubramaniam, B. Cutler, J. Liu, B. Khessib, and K. Vaid, "Ssd failures in datacenters: What? when? and why?" *Proceedings of the 9th ACM International on Systems and Storage Conference*, p. 7, 2016.

[3] J. Luo, T. ChuanJen, and K. Minhorng, "Ssd with sata and usb interfaces," Jun. 28 2011, uS Patent 7,970,978.

[4] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "A large-scale study of flash memory failures in the field," *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 1, pp. 177–190, 2015.

[5] D. R.-J. G.-J. Rydning, "The digitization of the world from edge to core," *Framingham: International Data Corporation*, 2018.

[6] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[7] B. Schroeder, R. Lagisetty, and A. Merchant, "Flash reliability in production: The expected and the unexpected," *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pp. 67–80, 2016.

[8] E. Xu, M. Zheng, F. Qin, Y. Xu, and J. Wu, "Lessons and actions: What we learned from 10k ssd-related storage system failures," *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pp. 961–976, 2019.

[9] C. Chakraborttii and H. Litz, "Learning i/o access patterns to improve prefetching in ssds," *ECML-PKDD*, 2020.

[10] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "Sfs: random write considered harmful in solid state drives." in *FAST*, vol. 12, 2012, pp. 1–16.

[11] Z. Whittaker, "Solid-state disk prices falling still more costly than hard disks," *Between the Lines. ZDNet.*, 2012.

[12] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, "Introduction to flash memory," *Proceedings of the IEEE*, vol. 91, no. 4, pp. 489–502, 2003.

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

[14] M. Kawamoto, "Hdd interface technologies," *Fujitsu scientific and technical journal*, vol. 42, no. 1, pp. 78–92, 2006.

[15] P. Cappelletti, C. Golla, P. Olivo, and E. Zanoni, *Flash memories*. Springer Science & Business Media, 2013.

[16] R. Chien, "Cyclic decoding procedures for bose-chaudhuri-hocquenghem codes," *IEEE Transactions on information theory*, vol. 10, no. 4, pp. 357–363, 1964.

[17] S. B. Wicker and V. K. Bhargava, *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[18] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of flash translation layer," *Journal of Systems Architecture*, vol. 55, no. 5-6, pp. 332–343, 2009.

[19] K. Smith, "Garbage collection," *SandForce, Flash Memory Summit, Santa Clara, CA*, pp. 1–9, 2011.

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

[21] W. Bux and I. Iliadis, "Performance of greedy garbage collection in flash-based solid-state drives," *Performance Evaluation*, vol. 67, no. 11, pp. 1172–1186, 2010.

[22] K.-H. Jang and T. H. Han, "Efficient garbage collection policy and block management method for nand flash memory," in *2010 2nd International Conference on Mechanical and Electronics Engineering*, vol. 1. IEEE, 2010, pp. V1–327.

[23] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 245–257.

[24] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[25] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.

[26] J. Zhang and M. Zulkernine, "Anomaly based network intrusion detection with unsupervised outlier detection," *2006 IEEE International Conference on Communications*, vol. 5, pp. 2388–2393, 2006.

[27] R. R. Sillito and R. B. Fisher, "Semi-supervised learning for anomalous trajectory detection." *BMVC*, vol. 1, pp. 035–1, 2008.

[28] H. Lu, Y. Li, S. Mu, D. Wang, H. Kim, and S. Serikawa, "Motor anomaly detection for unmanned aerial vehicles using reinforcement learning," *IEEE internet of things journal*, vol. 5, no. 4, pp. 2315–2322, 2017.

[29] H. B. Barlow, "Unsupervised learning," *Neural computation*, vol. 1, no. 3, pp. 295–311, 1989.

[30] O. Chapelle, B. Scholkopf, and A. Zien, "Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]," *IEEE Transactions on Neural Networks*, vol. 20, no. 3, pp. 542–542, 2009.

[31] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[32] R. E. Schapire, "The boosting approach to machine learning: An overview," in *Nonlinear estimation and classification*. Springer, 2003, pp. 149–171.

[33] "A comprehensive guide to boosting machine learning algorithms," https://www.edureka.co/blog/boosting-machine-learning/, accessed: 2018-09-30.

[34] N. V. Chawla, A. Lazarevic, L. O. Hall, and K. W. Bowyer, "Smoteboost: Improving prediction of the minority class in boosting," *European conference on principles of data mining and knowledge discovery*, pp. 107–119, 2003.

[35] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse, and A. Napolitano, "Rusboost: A hybrid approach to alleviating class imbalance," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 40, no. 1, pp. 185–197, 2009.

[36] G. Rätsch, T. Onoda, and K.-R. Müller, "Soft margins for adaboost," *Machine learning*, vol. 42, no. 3, pp. 287–320, 2001.

[37] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," *2008 Eighth IEEE International Conference on Data Mining*, pp. 413–422, 2008.

[38] A. K. Jain, J. Mao, and K. M. Mohiuddin, "Artificial neural networks: A tutorial," *Computer*, vol. 29, no. 3, pp. 31–44, 1996.

[39] B. B. Benuwa, Y. Z. Zhan, B. Ghansah, D. K. Wornyo, and F. Banaseka Kataka, "A review of deep machine learning," *International Journal of Engineering Research in Africa*, vol. 24, pp. 124–136, 2016.

[40] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," *Eleventh annual conference of the international speech communication association*, 2010.

[41] S. Hochreiter and J. Schmidhuber, "Lstm can solve hard long time lag problems," *Advances in neural information processing systems*, pp. 473–479, 1997.

[42] X. Qing and Y. Niu, "Hourly day-ahead solar irradiance prediction using weather forecasts by lstm," *Energy*, vol. 148, pp. 461–468, 2018.

[43] D. M. Nelson, A. C. Pereira, and R. A. de Oliveira, "Stock market's price movement prediction with lstm neural networks," in *2017 International joint conference on neural networks (IJCNN)*. IEEE, 2017, pp. 1419–1426.

[44] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," *IET*, 1999.

[45] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[46] Y. Wang, H. Yao, and S. Zhao, "Auto-encoder based dimensionality reduction," *Neurocomputing*, vol. 184, pp. 232–242, 2016.

[47] W. Wang, Y. Huang, Y. Wang, and L. Wang, "Generalized autoencoder: A neural network framework for dimensionality reduction," *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pp. 490–497, 2014.

[48] Y. Goldberg and O. Levy, "word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method," *arXiv preprint arXiv:1402.3722*, 2014.

[49] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, 2009.

[50] H. Leung and S. Haykin, "The complex backpropagation algorithm," *IEEE Transactions on signal processing*, vol. 39, no. 9, pp. 2101–2104, 1991.

[51] "What are word embeddings for text?" https://machinelearningmastery.com/what-are-word-embeddings/, accessed: 2018-09-30.

[52] C. Lea, R. Vidal, A. Reiter, and G. D. Hager, "Temporal convolutional networks: A unified approach to action segmentation," in *European Conference on Computer Vision*. Springer, 2016, pp. 47–54.

[53] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," *arXiv preprint arXiv:1404.2188*, 2014.

[54] R. Wan, S. Mei, J. Wang, M. Liu, and F. Yang, "Multivariate temporal convolutional network: A deep neural networks approach for multivariate time series forecasting," *Electronics*, vol. 8, no. 8, p. 876, 2019.

[55] Y. Wei, H. Xiao, H. Shi, Z. Jie, J. Feng, and T. S. Huang, "Revisiting dilated convolution: A simple approach for weakly-and semi-supervised semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7268–7277.

[56] X. Luo, W. Gan, L. Wang, Y. Chen, and E. Ma, "A deep learning prediction model for structural deformation based on temporal convolutional networks," *Computational Intelligence and Neuroscience*, vol. 2021, 2021.

[57] H. Riggs, S. Tufail, I. Parvez, and A. Sarwat, "Survey of solid state drives, characteristics, technology, and applications," *2020 SoutheastCon*, vol. 4, no. 4, pp. 1–6, 2020.

[58] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu, "Improving 3d nand flash memory lifetime by tolerating early retention loss and process variation," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 3, p. 37, 2018.

[59] J. H. Yun, J. H. Yoon, E. H. Nam, and S. L. Min, "An abstract fault model for nand flash memory," *IEEE Embedded Systems Letters*, vol. 4, no. 4, pp. 86–89, 2012.

[60] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi, "Differential raid: Rethinking raid for ssd reliability," *ACM Transactions on Storage (TOS)*, vol. 6, no. 2, pp. 1–22, 2010.

[61] E. S. Eleftheriou, R. Haas, X. Hu, and R. A. Pletka, "Reliability scheme using hybrid ssd/hdd replication with log structured management," Apr. 15 2014, uS Patent 8,700,949.

[62] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," *FAST*, 2007.

[63] A. Ma, R. Traylor, F. Douglis, M. Chamness, G. Lu, D. Sawyer, S. Chandra, and W. Hsu, "Raidshield: characterizing, monitoring, and proactively protecting against disk failures," *ACM Transactions on Storage (TOS)*, vol. 11, no. 4, p. 17, 2015.

[64] M. M. Botezatu, I. Giurgiu, J. Bogojeska, and D. Wiesmann, "Predicting disk replacement towards reliable data centers," *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 39–48, 2016.

[65] Y. Xu, K. Sui, R. Yao, H. Zhang, Q. Lin, Y. Dang, P. Li, K. Jiang, W. Zhang, J.-G. Lou *et al.*, "Improving service availability of cloud systems by predicting disk error," *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pp. 481–494, 2018.

[66] H. P. Belgal, N. Righos, I. Kalastirsky, J. J. Peterson, R. Shiner, and N. Mielke, "A new reliability model for post-cycling charge retention of flash memories," *2002 IEEE International Reliability Physics Symposium. Proceedings. 40th Annual (Cat. No. 02CH37320)*, pp. 7–20, 2002.

[67] Y. Cai, O. Mutlu, E. F. Haratsch, and K. Mai, "Program interference in mlc nand flash memory: Characterization, modeling, and mitigation," *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pp. 123–130, 2013.

[68] T.-S. Jung, Y.-J. Choi, K.-D. Suh, B.-H. Suh, J.-K. Kim, Y.-H. Lim, Y.-N. Koh, J.-W. Park, K.-J. Lee, J.-H. Park *et al.*, "A 3.3 v 128 mb multi-level nand flash memory for mass storage applications," *1996 IEEE International Solid-State Circuits Conference. Digest of TEchnical Papers, ISSCC*, pp. 32–33, 1996.

[69] J. Cooke, "The inconvenient truths of nand flash memory," *Flash Memory Summit*, vol. 3, no. 3, pp. 3–1, 2007.

[70] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you?" *FAST*, vol. 7, no. 1, pp. 1–16, 2007.

[71] R. Degraeve, F. Schuler, B. Kaczer, M. Lorenzini, D. Wellekens, P. Hendrickx, M. van Duuren, G. Dormans, J. Van Houdt, L. Haspeslagh *et al.*, "Analytical percolation model for predicting anomalous charge loss in flash memories," *IEEE Transactions on Electron Devices*, vol. 51, no. 9, pp. 1392–1400, 2004.

[72] J. Alter, J. Xue, A. Dimnaku, and E. Smirni, "Ssd failures in the field: symptoms, causes, and prediction models," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 75, 2019.

[73] A. P. Bradley, "The use of the area under the roc curve in the evaluation of machine learning algorithms," *Pattern Recognition*, pp. 1145–1159, 1997.

[74] A. Fernández, S. Garcia, F. Herrera, and N. V. Chawla, "Smote for learning from imbalanced data: progress and challenges, marking the 15-year anniversary," *Journal of artificial intelligence research*, vol. 61, pp. 863–905, 2018.

[75] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, 2006.

[76] B. Schroeder, A. Merchant, and R. Lagisetty, "Reliability of nand-based ssds: What field studies tell us," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1751–1769, 2017.

[77] X. Zhang, Q. Lin, Y. Xu, S. Qin, H. Zhang, B. Qiao, Y. Dang, X. Yang, Q. Cheng, M. Chintalapati *et al.*, "Cross-dataset time series anomaly detection for cloud systems," *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pp. 1063–1076, 2019.

[78] G. Hamerly, C. Elkan *et al.*, "Bayesian approaches to failure prediction for disk drives," *ICML*, vol. 1, pp. 202–209, 2001.

[79] F. Mahdisoltani, I. Stefanovici, and B. Schroeder, "Improving storage system reliability with proactive error prediction," *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, pp. 391–402, 2017.

[80] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[81] M. Lindenbaum, S. Markovich, D. Rusakov *et al.*, "Selective sampling for nearest neighbor classifiers," *Proceedings of The National Conference on Artificial Intelligence*, pp. 366–371, 1999.

[82] Y. Chen, X. S. Zhou, and T. S. Huang, "One-class svm for learning in image retrieval," *Image Processing, 2001. Proceedings. 2001 International Conference on*, vol. 1, pp. 34–37, 2001.

[83] M. Sabokrou, M. Fathy, and M. Hoseini, "Video anomaly detection and localisation based on the sparsity and reconstruction error of auto-encoder," *Electronics Letters*, vol. 52, no. 13, pp. 1122–1124, 2016.

[84] A. Guha, "Method and system for proactive drive replacement for high availability storage systems," May 13 2008, uS Patent 7,373,559.

[85] N. Sánchez-Maroño, A. Alonso-Betanzos, and M. Tombilla-Sanromán, "Filter methods for feature selection–a comparative study," *International Conference on Intelligent Data Engineering and Automated Learning*, pp. 178–187, 2007.

[86] T. N. Lal, O. Chapelle, J. Weston, and A. Elisseeff, "Embedded methods," in *Feature extraction*. Springer, 2006, pp. 137–165.

[87] L. Talavera, "An evaluation of filter and wrapper methods for feature selection in categorical clustering," *International Symposium on Intelligent Data Analysis*, pp. 440–451, 2005.

[88] A. Pearson, "The use of ranking formulae in r & d projects," *R&D Management*, vol. 2, no. 2, pp. 69–73, 1972.

[89] J. H. Zar, "Spearman rank correlation," *Encyclopedia of Biostatistics*, vol. 7, 2005.

[90] M. L. McHugh, "The chi-square test of independence," *Biochemia medica: Biochemia medica*, vol. 23, no. 2, pp. 143–149, 2013.

[91] K. J. Johnson and R. E. Synovec, "Pattern recognition of jet fuels: comprehensive gc× gc with anova-based feature selection and principal component analysis," *Chemometrics and Intelligent Laboratory Systems*, vol. 60, no. 1-2, pp. 225–237, 2002.

[92] P. M. Granitto, C. Furlanello, F. Biasioli, and F. Gasperi, "Recursive feature elimination with random forest for ptr-ms analysis of agroindustrial products," *Chemometrics and Intelligent Laboratory Systems*, vol. 83, no. 2, pp. 83–90, 2006.

[93] G. Louppe, L. Wehenkel, A. Sutera, and P. Geurts, "Understanding variable importances in forests of randomized trees," *Advances in neural information processing systems*, pp. 431–439, 2013.

[94] Y. Zhou, R. Jin, and S. C.-H. Hoi, "Exclusive lasso for multi-task feature selection," *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 988–995, 2010.

[95] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the royal statistical society: series B (statistical methodology)*, vol. 67, no. 2, pp. 301–320, 2005.

[96] Q. Gu, Z. Li, and J. Han, "Generalized fisher score for feature selection," *arXiv preprint arXiv:1202.3725*, 2012.

[97] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[98] C. Croux and C. Dehon, "Influence functions of the spearman and kendall correlation measures," *Statistical methods & applications*, vol. 19, no. 4, pp. 497–515, 2010.

[99] H. Zou and H. H. Zhang, "On the adaptive elastic-net with a diverging number of parameters," *Annals of statistics*, vol. 37, no. 4, p. 1733, 2009.

[100] A. I. McLeod, "Kendall rank correlation and mann-kendall trend test," *R Package Kendall*, 2005.

[101] M. B. Kursa, W. R. Rudnicki *et al.*, "Feature selection with the boruta package," *J Stat Softw*, vol. 36, no. 11, pp. 1–13, 2010.

[102] K. Yan and D. Zhang, "Feature selection and analysis on correlated gas sensor data with recursive feature elimination," *Sensors and Actuators B: Chemical*, vol. 212, pp. 353–363, 2015.

[103] Q. Cheng, P. K. Varshney, and M. K. Arora, "Logistic regression for feature selection and soft classification of remote sensing data," *IEEE Geoscience and Remote Sensing Letters*, vol. 3, no. 4, pp. 491–494, 2006.

[104] V. Fonti and E. Belitser, "Feature selection using lasso," *VU Amsterdam Research Paper in Business Analytics*, vol. 30, pp. 1–25, 2017.

[105] S. Paul and P. Drineas, "Feature selection for ridge regression with provable guarantees," *Neural computation*, vol. 28, no. 4, pp. 716–742, 2016.

[106] "Smoteboost," https://www.mathworks.com/matlabcentral/fileexchange/37311-smoteboost, accessed: 2020-01-14.

[107] D. Cook, *Practical machine learning with H2O: powerful, scalable techniques for deep learning and AI*. " O'Reilly Media, Inc.", 2016.

[108] Z. Zhang, "Improved adam optimizer for deep neural networks," in *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*. IEEE, 2018, pp. 1–2.

[109] F. Mahdisoltani, I. Stefanovici, and B. Schroeder, "Proactive error prediction to improve storage system reliability," *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pp. 391–402, 2017.

[110] T. Fawcett, "An introduction to roc analysis," *Pattern recognition letters*, vol. 27, no. 8, pp. 861–874, 2006.

[111] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[112] A. Reiss, G. Hendeby, and D. Stricker, "A novel confidence-based multiclass boosting algorithm for mobile physical activity monitoring," *Personal and Ubiquitous Computing*, vol. 19, no. 1, pp. 105–121, 2015.

[113] M. Sabhnani and G. Serpen, "Application of machine learning algorithms to kdd intrusion detection dataset within misuse detection context." in *MLMTA*, 2003, pp. 209–215.

[114] J. D. Echard, "Estimation of radar detection and false alarm probability," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 27, no. 2, pp. 255–260, 1991.

[115] V. Bellini, A. Schiavone, T. Di Noia, A. Ragone, and E. Di Sciascio, "Knowledge-aware autoencoders for explainable recommender systems," in *Proceedings of the 3rd Workshop on Deep Learning for Recommender Systems*, 2018, pp. 24–31.

[116] V. Svensson, A. Gayoso, N. Yosef, and L. Pachter, "Interpretable factor models of single-cell rna-seq via variational autoencoders," *Bioinformatics*, vol. 36, no. 11, pp. 3418–3421, 2020.

[117] S. Mishra, B. L. Sturm, and S. Dixon, "Local interpretable model-agnostic explanations for music content analysis." in *ISMIR*, 2017, pp. 537–543.

[118] S. M. Shankaranarayana and D. Runje, "Alime: Autoencoder based approach for local interpretability," in *International conference on intelligent data engineering and automated learning*. Springer, 2019, pp. 454–463.

[119] M. Jung and M. Kandemir, "Revisiting widely held ssd expectations and rethinking system-level implications," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1, pp. 203–216, 2013.

[120] Y. Cai, Y. Wu, and E. F. Haratsch, "Error correction code (ecc) selection using probability density functions of error correction capability in storage controllers with multiple error correction codes," Aug. 16 2016, uS Patent 9,419,655.

[121] C. Reche, L. Nevill, and T. Martin, "Error detection/correction based memory management," Nov. 13 2012, uS Patent 8,312,349.

[122] G. Moore, "Moore's law," *Electronics Magazine*, vol. 38, no. 8, p. 114, 1965.

[123] G. Wu and X. He, "Reducing ssd read latency via nand flash program and erase suspension." in *FAST*, vol. 12, 2012, pp. 10–10.

[124] V. Mohan, T. Siddiqua, S. Gurumurthi, and M. R. Stan, "How i learned to stop worrying and love flash endurance." *HotStorage*, vol. 10, pp. 3–3, 2010.

[125] H. Kim and U. Ramachandran, "Flashfire: Overcoming the performance bottleneck of flash storage technology," Georgia Institute of Technology, Tech. Rep., 2010.

[126] R. B. Da Zheng and A. S. Szalay, "A parallel page cache: Iops and caching for multicore systems," in *Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems*, 2012, pp. 5–5.

[127] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, pp. 1–32, 2013.

[128] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," *ACM SIGARCH Computer Architecture News*, vol. 19, no. 2, pp. 40–52, 1991.

[129] M. Li, E. Varki, S. Bhatia, and A. Merchant, "Tap: Table-based prefetching for storage caches." *FAST*, vol. 8, pp. 1–16, 2008.

[130] T. C. Mowry, A. K. Demke, O. Krieger *et al.*, "Automatic compiler-inserted i/o prefetching for out-of-core applications," in *OSDI*, vol. 96, 1996, pp. 3–17.

[131] R. Xu, X. Jin, L. Tao, S. Guo, Z. Xiang, and T. Tian, "An efficient resource-optimized learning prefetcher for solid state drives," *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 273–276, 2018.

[132] M. Nijim, Z. Zong, X. Qin, and Y. Nijim, "Multi-layer prefetching for hybrid storage systems: algorithms, models, and evaluations," in *2010 39th international conference on parallel processing workshops*. IEEE, 2010, pp. 44–49.

[133] M. Nijim, "Modelling speculative prefetching for hybrid storage systems," in *2010 IEEE Fifth International Conference on Networking, Architecture, and Storage*. IEEE, 2010, pp. 143–151.

[134] W. Fengguang, X. Hongsheng, and X. Chenfeng, "On the design of a new linux readahead framework," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 75–84, 2008.

[135] W.-S. Han, K.-Y. Whang, and Y.-S. Moon, "A formal framework for prefetching based on the type-level access pattern in object-relational dbmss," *IEEE Transactions on knowledge and data engineering*, vol. 17, no. 10, pp. 1436–1448, 2005.

[136] I. Averbouch, A. J. Birnbaum, J. T. Hsieh, and C.-L. K. Shum, "Automatic pattern-based operand prefetching," Feb. 10 2015, uS Patent 8,954,678.

[137] P. Mehra, "Samsung smartssd: Accelerating data-rich applications," *Flash Memory Summit*, 2019.

[138] S. Boboila and P. Desnoyers, "Performance models of flash-based solid-state drives for real workloads," in *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2011, pp. 1–6.

[139] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[140] A. Ki and A. E. Knowles, "Stride prefetching for the secondary data cache," *Journal of systems architecture*, vol. 46, no. 12, pp. 1093–1102, 2000.

[141] A. Laga, J. Boukhobza, M. Koskas, and F. Singhoff, "Lynx: A learning linux prefetching mechanism for ssd performance model," *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pp. 1–6, 2016.

[142] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," *arXiv preprint arXiv:1803.02329*, 2018.

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

[144] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 285–297, 2015.

[145] K. L. Chung, "Markov chains," *Springer-Verlag, New York*, 1967.

[146] J. R. Santos, R. R. Muntz, and B. Ribeiro-Neto, "Comparing random data allocation and data striping in multimedia servers," *ACM SIGMETRICS Performance Evaluation Review*, vol. 28, no. 1, pp. 44–55, 2000.

[147] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, and M. Sugawara, "Understanding storage traffic characteristics on enterprise virtual desktop infrastructure," in *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017, pp. 1–11.

[148] B. Xue, C. Fu, and Z. Shaobin, "A study on sentiment computing and classification of sina weibo with word2vec," in *2014 IEEE International Congress on Big Data*. IEEE, 2014, pp. 358–363.

[149] W. Liu, Y. Wen, Z. Yu, and M. Yang, "Large-margin softmax loss for convolutional neural networks." in *ICML*, vol. 2, no. 3, 2016, p. 7.

[150] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *international conference on machine learning*, 2016, pp. 1050–1059.

[151] J. Axboe, "Fio-flexible i/o tester synthetic benchmark," *URL https://github.com/axboe/fio (Accessed: 2015-06-13)*, 2005.

[152] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production windows servers," in *2008 IEEE International Symposium on Workload Characterization*. IEEE, 2008, pp. 119–128.

[153] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, pp. 1–23, 2008.

[154] "Msr cambridge traces," http://iotta.snia.org/traces/388.

[155] "Microsoft snia: Traces," http://iotta.snia.org/traces/4928.

[156] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[157] A. Tato and R. Nkambou, "Improving adam optimizer," *ICLR*, 2018.

[158] T. R. Puzak, "Analysis of cache replacement-algorithms," *Doctoral Dissertations Available from Proquest AAI8509594*, 1986.

[159] R. Stoica and A. Ailamaki, "Improving flash write performance by using update frequency," *Proceedings of the VLDB Endowment*, vol. 6, no. 9, pp. 733–744, 2013.

[160] Y. Oh, J. Choi, D. Lee, and S. H. Noh, "Improving performance and lifetime of the ssd raid-based host cache through a log-structured approach," *ACM SIGOPS Operating Systems Review*, vol. 48, no. 1, pp. 90–97, 2014.

[161] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2fs: A new file system for flash storage," in *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, 2015, pp. 273–286.

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

[163] T. Kim, S. S. Hahn, S. Lee, J. Hwang, J. Lee, and J. Kim, "Pcstream: automatic stream allocation using program contexts," in *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[164] K. Kremer and A. Brinkmann, "Fadac: A self-adapting data classifier for flash memory," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, 2019, pp. 167–178.

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

[166] M.-L. Chiang, P. C. Lee, and R.-C. Chang, "Using data clustering to improve cleaning performance for flash memory," *Software: Practice and Experience*, vol. 29, no. 3, pp. 267–290, 1999.

[167] J. Yang, S. Pei, and Q. Yang, "Warcip: Write amplification reduction by clustering i/o pages," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, 2019, pp. 155–166.

[168] R. Subramani, H. Swapnil, N. Thakur, B. Radhakrishnan, and K. Puttaiah, "Garbage collection algorithms for nand flash memory devices–an overview," *2013 European Modelling Symposium*, pp. 81–86, 2013.

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

[170] J. He, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The unwritten contract of solid state drives," in *Proceedings of the Twelfth European Conference on Computer Systems*. USA: European Conference on Computer Systems, 2017, pp. 127–144.

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

[172] G. Sun and S.-W. Jun, "Columnburst: a near-storage accelerator for memory-efficient database join queries," in *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2020, pp. 9–16.

[173] B. Athiwaratkun and J. W. Stokes, "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, 2017, pp. 2482–2486.

[174] Y. Manabe and B. Chakraborty, "A novel approach for estimation of optimal embedding parameters of nonlinear time series by structural learning of neural network," *Neurocomputing*, vol. 70, no. 7-9, pp. 1360–1371, 2007.

[175] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[176] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[177] T. J. Brazil, "Causal-convolution-a new method for the transient analysis of linear systems at microwave frequencies," *IEEE transactions on microwave theory and techniques*, vol. 43, no. 2, pp. 315–323, 1995.

[178] K. Yarrow, "Temporal dilation: the chronostasis illusion and spatial attention," *Attention and time*, pp. 163–176, 2010.

[179] S. Gold, A. Rangarajan *et al.*, "Softmax to softassign: Neural network algorithms for combinatorial optimization," *Journal of Artificial Neural Networks*, vol. 2, no. 4, pp. 381–399, 1996.

[180] S. Huang, N. Cai, P. P. Pacheco, S. Narrandes, Y. Wang, and W. Xu, "Applications of support vector machine (svm) learning in cancer genomics," *Cancer Genomics-Proteomics*, vol. 15, no. 1, pp. 41–51, 2018.

[181] B. Xu, X. Guo, Y. Ye, and J. Cheng, "An improved random forest classifier for text categorization." *JCP*, vol. 7, no. 12, pp. 2913–2920, 2012.

[182] C. Chen, B. Wu, M. Qiu, L. Wang, and J. Zhou, "A comprehensive analysis of information leakage in deep transfer learning," *arXiv preprint arXiv:2009.01989*, 2020.

[183] S. Wang, D. Zhou, X. Han, and T. Yoshimura, "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, 2017, pp. 1032–1037.

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

[185] B. Karlik and A. V. Olgac, "Performance analysis of various activation functions in generalized mlp architectures of neural networks," *International Journal of Artificial Intelligence and Expert Systems*, vol. 1, no. 4, pp. 111–122, 2011.

[186] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond," *arXiv preprint arXiv:1904.09237*, 2019.

[187] SIA, "Block io traces," http://iotta.snia.org/tracetypes/3, Dec 2001, accessed on 2021-01-11.

[188] M. A. Pimentel, D. A. Clifton, L. Clifton, and L. Tarassenko, "A review of novelty detection," *Signal Processing*, vol. 99, pp. 215–249, 2014.

[189] A. Pumsirirat and L. Yan, "Credit card fraud detection using deep learning based on auto-encoder and restricted boltzmann machine," *International Journal of advanced computer science and applications*, vol. 9, no. 1, pp. 18–25, 2018.

[190] A. O. Adewumi and A. A. Akinyelu, "A survey of machine-learning and nature-inspired based credit card fraud detection techniques," *International Journal of System Assurance Engineering and Management*, vol. 8, no. 2, pp. 937–953, 2017.

[191] C. Kwon, W. Liu, and I. Hwang, "Security analysis for cyber-physical systems against stealthy deception attacks," in *2013 American control conference*. IEEE, 2013, pp. 3344–3349.

[192] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani, "Deep learning for iot big data and streaming analytics: A survey," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 2923–2960, 2018.

[193] J. E. Ball, D. T. Anderson, and C. S. Chan, "Comprehensive survey of deep learning in remote sensing: theories, tools, and challenges for the community," *Journal of Applied Remote Sensing*, vol. 11, no. 4, p. 042609, 2017.

[194] B. R. Kiran, D. M. Thomas, and R. Parakkal, "An overview of deep learning based methods for unsupervised and semi-supervised anomaly detection in videos," *Journal of Imaging*, vol. 4, no. 2, p. 36, 2018.

[195] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "Prepare: Predictive performance anomaly prevention for virtualized cloud systems," in *2012 IEEE 32nd International Conference on Distributed Computing Systems*. IEEE, 2012, pp. 285–294.

[196] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," *Medical image analysis*, vol. 42, pp. 60–88, 2017.

[197] K. M. Ali Alheeti, A. Gruebler, and K. McDonald-Maier, "Intelligent intrusion detection of grey hole and rushing attacks in self-driving vehicular networks," *Computers*, vol. 5, no. 3, p. 16, 2016.

[198] K. M. A. Alheeti, A. Gruebler, K. D. McDonald-Maier, and A. Fernando, "Prediction of dos attacks in external communication for self-driving vehicles using a fuzzy petri net model," in *2016 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, 2016, pp. 502–503.

[199] H. Litz and M. Hashemi, "Machine learning for systems," *IEEE Micro*, vol. 40, no. 5, pp. 6–7, 2020.

[200] A. Klimovic, H. Litz, and C. Kozyrakis, "Selecta: Heterogeneous cloud storage configuration for data analytics," in *2018 USENIX Annual Technical Conference*, 2018, pp. 759–773.

[201] C. Li, D. Feng, Y. Hua, and F. Wang, "Improving raid performance using an endurable ssd cache," *2016 45th International Conference on Parallel Processing (ICPP)*, pp. 396–405, 2016.

[202] S. Moon and A. L. N. Reddy, "Dont let raid raid the lifetime of your ssd array," *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, 2013. [Online]. Available: https://www.usenix.org/conference/hotstorage13/workshop-program/presentation/Moon

[203] A. R. Olson, D. J. Langlois *et al.*, "Solid state drives data reliability and lifetime," *Imation White Paper*, pp. 1–27, 2008.

[204] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf, "Characterizing flash memory: anomalies, observations, and applications," *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 24–33, 2009.

[205] J. F. Murray, G. F. Hughes, and K. Kreutz-Delgado, "Machine learning methods for predicting failures in hard drives: A multiple-instance application," *Journal of Machine Learning Research*, vol. 6, no. May, pp. 783–816, 2005.

[206] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, M. Chintalapati, A. Krishnamurthy, and T. Anderson, "Deepview: Virtual disk failure diagnosis and pattern detection for azure," *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pp. 519–532, 2018.

[207] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: a large-scale field study," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1, pp. 193–204, 2009.

[208] M. Martinez-Garcia, Y. Zhang, J. Wan, and J. Mcginty, "Visually interpretable profile extraction with an autoencoder for health monitoring of industrial systems," in *2019 IEEE 4th International Conference on Advanced Robotics and Mechatronics (ICARM)*. IEEE, 2019, pp. 649–654.

[209] C. Chakraborttii, V. Sinha, and H. Litz, "Ssd qos improvements through machine learning," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 511–511.

[210] W. Ali, S. M. Shamsuddin, A. S. Ismail *et al.*, "A survey of web caching and prefetching," *Int. J. Advance. Soft Comput. Appl*, vol. 3, no. 1, pp. 18–44, 2011.

[211] Y. Zeng, "Long short term based memory hardware prefetcher," *MEMSYS*, 2017.

[212] D. M. Huizinga and S. Desai, "Implementation of informed prefetching and caching in linux," *Proceedings International Conference on Information Technology: Coding and Computing (Cat. No. PR00540)*, pp. 443–448, 2000.

[213] Q. Huang, K. Birman, R. Van Renesse, W. Lloyd, S. Kumar, and H. C. Li, "An analysis of facebook photo caching," *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 167–181, 2013.

[214] C.-K. Yang, T. Mitra, and T.-c. Chiueh, "A decoupled architecture for application-specific file prefetching." *USENIX Annual Technical Conference, FREENIX Track*, pp. 157–170, 2002.

[215] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 4, pp. 311–343, 1996.

[216] T. M. Kroeger and D. D. Long, "Design and implementation of a predictive file prefetching algorithm." *USENIX Annual Technical Conference, General Track*, pp. 105–118, 2001.

[217] S. H. Baek and K. H. Park, "Prefetching with adaptive cache culling for striped disk arrays," *The 2008 USENIX Annual Technical Conference*, pp. 363–376, 2008.

[218] S.-j. Ahn, H.-G. Lee, J.-H. Kim, Y.-b. Kim, S. Kim, Y.-i. Seo, and C.-h. Park, "Method of prefetching data in hard disk drive, recording medium including program to execute the method, and apparatus to perform the method," Jul. 12 2011, uS Patent 7,979,631.

[219] S. Liang, S. Jiang, and X. Zhang, "Step: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers," *27th International Conference on Distributed Computing Systems (ICDCS'07)*, pp. 64–64, 2007.

[220] R. Ye, W. Meng, and S. Wan, "Extending lifetime of ssd in raid5 systems through a reliable hierarchical cache," *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pp. 1–8, 2017.

[221] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 436–448, 2017.

[222] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache replacement based on reuse-distance prediction," *2007 25th International Conference on Computer Design*, pp. 245–250, 2007.

[223] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[224] B. S. Gill and L. A. D. Bathen, "Amp: Adaptive multi-stream prefetching in a shared cache." *FAST*, vol. 7, no. 5, pp. 185–198, 2007.

[225] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache." *FAST*, vol. 3, no. 2003, pp. 115–130, 2003.

[226] K. M. Curewitz, P. Krishnan, and J. S. Vitter, "Practical prefetching via data compression," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 257–266, 1993.

[227] A. J. Uppal, R. C. Chiang, and H. H. Huang, "Flashy prefetching for high-performance flash drives," *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–12, 2012.

[228] B. S. Gill and D. S. Modha, "Sarc: Sequential prefetching in adaptive replacement cache." *USENIX Annual Technical Conference, General Track*, pp. 293–308, 2005.

[229] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *ACM SIGARCH Computer Architecture News*, vol. 25, no. 2, pp. 252–263, 1997.

[230] S.-w. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, "Machine learning-based prefetch optimization for data center applications," *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, p. 56, 2009.

[231] S. K. K. Visvanathan and R. Ugale, "Intelligent cache pre-fetch," Oct. 9 2018, uS Patent 10,095,624.

[232] R. Pike, "Storage mechanism with variable block size," Mar. 13 2014, uS Patent App. 13/612,968.

[233] S. Kondguli and M. Huang, "T2: A highly accurate and energy efficient stride prefetcher," in *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 373–376.

[234] S. Iacobovici, S. Kadambi, and Y. C. Chou, "Multi-stride prefetcher with a recurring prefetch table," Feb. 3 2009, uS Patent 7,487,296.

[235] S. O. H. Y. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," *IEEE*, 2006.

[236] J. P. Bradford, H. F. Kossman, and T. J. Mullins, "Context switch instruction prefetching in multithreaded computer," Nov. 10 2009, uS Patent 7,617,499.

[237] T. O. Iwasaki, S. Ning, H. Yamazawa, C. Sun, S. Tanakamaru, and K. Takeuchi, "Machine learning prediction for 13x endurance enhancement in reram ssd system," in *2015 IEEE International Memory Workshop (IMW)*. IEEE, 2015, pp. 1–4.

[238] J.-E. Dartois, J. Boukhobza, A. Knefati, and O. Barais, "Investigating machine learning algorithms for modeling ssd i/o performance for container-based virtualization," *IEEE transactions on cloud computing*, 2019.

[239] B. Li, C. Deng, J. Yang, D. Lilja, B. Yuan, and D. Du, "Haml-ssd: A hardware accelerated hotness-aware machine learning based ssd management," in *38th IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2019*. Institute of Electrical and Electronics Engineers Inc., 2019, p. 8942140.

[240] C. Chakraborttii, V. Sinha, and H. Litz, "Ssd qos improvements through machine learning," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 511–511.

[241] J. Wan, D. Wang, S. C. H. Hoi, P. Wu, J. Zhu, Y. Zhang, and J. Li, "Deep learning for content-based image retrieval: A comprehensive study," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 157–166.

[242] H. Lee, P. Pham, Y. Largman, and A. Y. Ng, "Unsupervised feature learning for audio classification using convolutional deep belief networks," in *Advances in neural information processing systems*, 2009, pp. 1096–1104.

[243] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *ieee Computational intelligenCe magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[244] "Garbage collection in single-level cell nand flash memory," https://www.micron.com/media/client/global/Documents/Products/Technical%20Note/NAND%20Flash/tn2960_garbage_collection_slc_nand.ashx, accessed: 2018-09-30.

[245] W. Kang, D. Shin, and S. Yoo, "Reinforcement learning-assisted garbage collection to mitigate long-tail latency in ssd," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 134, 2017.

[246] P. Desnoyers, "Analytic modeling of ssd write performance," *Proceedings of the 5th Annual International Systems and Storage Conference*, p. 12, 2012.

[247] Q. Zhang, X. Li, L. Wang, T. Zhang, Y. Wang, and Z. Shao, "Lazy-rtgc: A real-time lazy garbage collection mechanism with jointly optimizing average and worst performance for nand flash memory storage systems," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 3, p. 43, 2015.

[248] A. Gupta, Y. Kim, and B. Urgaonkar, *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings*. ACM, 2009, vol. 44, no. 3.

[249] M. Lin and S. Chen, "Efficient and intelligent garbage collection policy for nand flash-based consumer electronics," *IEEE Transactions on Consumer Electronics*, vol. 59, no. 3, pp. 538–543, 2013.

[250] O. Kwon, K. Koh, J. Lee, and H. Bahn, "Fegc: An efficient garbage collection scheme for flash memory based storage systems," *Journal of Systems and Software*, vol. 84, no. 9, pp. 1507–1523, 2011.

[251] B. Van Houdt, "A mean field model for a class of garbage collection algorithms in flash-based solid state drives," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1, pp. 191–202, 2013.

[252] Q. Wei, B. Gong, S. Pathak, B. Veeravalli, L. Zeng, and K. Okada, "Waftl: A workload adaptive flash translation layer with data partition," *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–12, 2011.

[253] Z. Shen, C. C. Young, S. Zeng, K. Murthy, and K. Bai, "Identifying resources for cloud garbage collection," *2016 12th International Conference on Network and Service Management (CNSM)*, pp. 248–252, 2016.

[254] J. Singer, G. Kovoor, G. Brown, and M. Luján, "Garbage collection auto-tuning for java mapreduce on multi-cores," *ACM SIGPLAN Notices*, vol. 46, no. 11, pp. 109–118, 2011.

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

[256] M. Lin and S. Y. Chen, "Swap time-aware garbage collection policy for nand flash-based swap system," *Electronics Letters*, vol. 49, no. 24, pp. 1525–1526, 2013.

[257] L. Han, Y. Ryu, and K. Yim, "Cata: a garbage collection scheme for flash memory file systems," in *International Conference on Ubiquitous Intelligence and Computing*. Springer, 2006, pp. 103–112.

[258] L.-z. Han, Y. Ryu, T.-s. Chung, M. Lee, and S. Hong, "An intelligent garbage collection algorithm for flash memory storages," in *International Conference on Computational Science and Its Applications*. Springer, 2006, pp. 1019–1027.

[259] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "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*, 2014, pp. 1–14.

[260] M. Shafaei, P. Desnoyers, and J. Fitzpatrick, "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)*, 2016.

[261] Q. Wang, J. Li, P. P. Lee, G. Zhao, C. Shi, and L. Huang, "In search of optimal data placement for eliminating write amplification in log-structured storage," *arXiv e-prints*, vol. 1, no. 1, pp. arXiv–2104, 2021.

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

[263] A. Klimovic, H. Litz, and C. Kozyrakis, "Reflex: Remote flash==local flash," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 345–359, 2017.

[264] D. A. Moon, "Garbage collection in a large lisp system," in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, 1984, pp. 235–246.

[265] J. Wang and Y. Hu, "Wolf-a novel reordering write buffer to boost the performance of log-structured file systems." in *FAST*, 2002, pp. 47–60.

[266] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz, "Trash day: Coordinating garbage collection in distributed systems," in *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[267] S. Moon and A. N. Reddy, "Don't let {RAID} raid the lifetime of your {SSD} array," in *5th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 13)*, 2013.

[268] J. Menon, "A performance comparison of raid-5 and log-structured arrays," in *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*. IEEE, 1995, pp. 167–178.

[269] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum *et al.*, "The ramcloud storage system," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, pp. 1–55, 2015.