RL-Watchdog: A Fast and Predictable SSD Liveness Watchdog on Storage Systems

Jin Yong Ha, *Seoul National University;* Sangjin Lee, *Chung-Ang University;*
Heon Young Yeom, *Seoul National University;* Yongseok Son, *Chung-Ang University*

https://www.usenix.org/conference/atc24/presentation/ha

This paper is included in the Proceedings of the
2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the
2024 USENIX Annual Technical Conference
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

RL-Watchdog: A Fast and Predictable SSD Liveness Watchdog on Storage Systems

Jin Yong Ha
Seoul National University
South Korea

Sangjin Lee
Chung-Ang University
South Korea

Heon Young Yeom
Seoul National University
South Korea

Yongseok Son*
Chung-Ang University
South Korea

Abstract

This paper proposes a reinforcement learning-based watchdog (RLW) that examines solid-state drive (SSD) liveness or failures by faults (e.g., controller/power faults and high temperature) quickly, precisely, and online to minimize application data loss. To do this, we first provide a lightweight watchdog (LWW) to actively and lightly examine SSD liveness by issuing a liveness-dedicated command to the SSD. Second, we introduce a reinforcement learning-based timeout predictor (RLTP) which predicts the timeout of the dedicated command, enabling the detection of a failure point regardless of the SSD model. Finally, we propose fast failure notification (FFN) to immediately notify the applications of the failure to minimize their potential data loss. We implement RLW with three techniques in a Linux kernel 6.0.0 and evaluate it in a single SSD and RAID using realistic power fault injection. The experimental results reveal that RLW reduces the data loss by up to 96.7% compared with the existing scheme, and its accuracy in predicting failure points reaches up to 99.8%.

1 Introduction

Compared with hard disk drives, solid-state drives (SSDs) have higher performance, better reliability, and lower power consumption and thus have been widely adopted in various storage systems, such as enterprise storage systems, data centers, and cloud storage [9, 23, 40, 41, 43]. Accordingly, the reliability of SSD and storage systems has become critical; therefore, various fault tolerance mechanisms have been adopted, such as replication [28, 73], redundant array of inexpensive disks (RAID [14, 47]), or transaction processing [44, 62, 63, 68, 69].

Unfortunately, even if these mechanisms can recover written data or committed transactions, when a sudden SSD failure caused by various faults (e.g., SSD controller/power faults [36, 71], high SSD temperature [45, 59, 67], loose interconnects by vibration [18], and a faulty PCIe slot [33]) occurs, preventing data loss in running applications (i.e., the host system is alive) is still a challenge [6, 7, 32, 74]. For example, in the Linux kernel, when applications perform buffered writes, they can lose a significant amount of data in the page cache ranging from tens to hundreds of gigabytes when SSD cannot be available anymore due to the faults. This is because file information or metadata associated with the pages (i.e., user data) in the page cache can be lost when the SSD failure occurs. Figure 1 shows that the application can either recognize a failure later (e.g., XFS or F2FS) or fail to recognize the failure (e.g., EXT4 or raw RAID device) on a single SSD or RAID configuration. This reveals that the Linux kernel may not detect SSD failure quickly, even if high-end non-volatile memory express (NVMe) SSDs are adopted. As a result, this failure can cause applications to require reproducing data through an additional time-consuming computation.

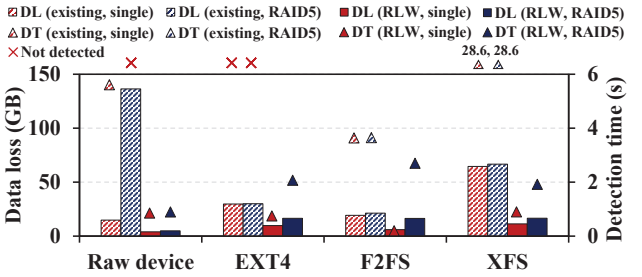


Figure 1: Application data loss (DL) and detection time (DT) upon an SSD failure (i.e., SSD power fault) in the Linux kernel (the command timeout is 1 second).

Specifically, according to our analysis, we observe three critical limitations of detecting and handling the SSD failure in the Linux storage stack as follows.

Loose-deterministic failure check: The current storage stack passively identifies that the SSD has failed only when a submitted command (e.g., I/O or admin command) is not completed within a given command timeout. Thus, failure detection can be significantly delayed until the command is submitted without a hard deterministic bound.

Fixed command timeout: The current storage stack employs a fixed timeout regardless of SSD models. Intuitively, a fixed small command timeout can decrease data loss, but

*Corresponding author: Yongseok Son (sysganda@cau.ac.kr).

Table 1: Categories and comparison with previous SSD failure studies.

Study	Kernel or application	Data loss mitigation	HW for realistic fault injection	ML prediction for future failure	ML prediction for command timeout	Offline or online learning
Ahmadian *et al.* [6, 7], Zheng *et al.* [71, 72]	Application		✓			
Tiger [24]	Application	✓				
Alter *et al.* [8], Chakraborttii *et al.* [12, 13] RUS_Ensemble [19], MVTRF [70]	Application			✓		Offline
Our study	Kernel	✓	✓		✓	**Online**

cause a false-positive detection, leading to an increased latency. Meanwhile, a large-fixed command may not incur the false-positive detection, however, can increase data loss. Determining and leveraging a "fixed" optimal timeout[1] can be effective temporarily. However, since this fixed value cannot cover all conditions (i.e., various SSD models and command types, and fluctuating workload intensity), it may not be considered a permanent solution.

Delayed failure notification: The current storage stack may not promptly notify running applications of the SSD failure. Specifically, the delayed notification is caused by a failure handling policy of each file system (e.g., some file systems notify the failure only after they fail to write critical metadata). Thus, during these behaviors, applications can accumulate data in the page cache until the failure is notified.

Previously, many studies have been conducted to investigate and predict SSD failures as presented in Table 1. As listed in the table, most studies detect or predict SSD failures at the application level, meanwhile, our study handles the SSD failure in the Linux kernel to detect it more quickly and it is also application-oblivious. Some studies [6, 7, 71, 72] have detected SSD failures directly using separate hardware to generate actual power faults, similar to our study. In contrast, we focus on mitigating the data loss by quickly detecting and precisely predicting the SSD failure points. Other studies [8, 12, 13, 19, 70] have predicted future SSD failures using log data from a large-scale SSD cluster and offline machine learning. Meanwhile, we focus on predicting the command latency by learning the current SSD states at run-time without offline pre-learning cost. Furthermore, we note that our study aims to detect actual failures that have occurred, meanwhile, other machine learning-based studies concentrate on predicting potential failures before they happen.

In this paper, we propose a novel reinforcement learning-based watchdog to monitor SSD liveness or failure called RL-watchdog (`RLW`). Our approach incorporates three techniques to quickly, precisely, and online detect SSD failures, thereby minimizing application data loss against the failures. Specifically, we first present a light-weighted watchdog (`LWW`) to monitor the SSD failures by periodically submitting a light-weighted liveness-monitoring command (`LWLC`) to the SSDs. It enables quick failure detection while minimizing interference from other I/O commands. Second, we introduce a reinforcement learning-based timeout predictor (`RLTP`) to predict the SSD command timeout online, regardless of device type. It enables the dynamic alteration of the command timeout at runtime while minimizing false-positive detection. Finally, we propose fast failure notification (`FFN`) which enables immediate SSD failure notification to applications to minimize data loss from upcoming write requests. By leveraging the abstraction provided by the virtual file system (VFS) layer and reusing existing failure code, it does not require application modification.

We implement `RLW` with all techniques in a Linux kernel 6.0.0. Then, we evaluate `RLW` using a single SSD and software RAID with various file systems including EXT4, XFS, and F2FS. We run micro/macro benchmarks and a real-world application (i.e., RocksDB) on two NVMe SSD models. For more realistic scenarios of SSD failure, we use a power control board (PCB) [7] that can inject a power fault to the NVMe SSD, independent of the system power. The experimental results reveal that `RLW` reduces the data loss by up to 96.7% compared with the existing scheme in the Linux kernel, and its accuracy in predicting the command timeout reaches up to 99.8%.

To the best of our knowledge, this paper is the first study to identify the issue of the detection scheme of NVMe SSD failure and introduce an online machine learning-based detector in the Linux kernel. Furthermore, we offer the source code at `https://github.com/OSopt/RL-Watchdog` to aid future studies in reducing data loss upon SSD failures.

2 Background and Motivation

This paper focuses on NVMe SSDs since they have increasingly been adopted in various storage systems [30, 34, 56, 58, 65, 66]. Despite their popularity and many advantages, NVMe SSD failure can occur frequently even if the system is alive [71]. For example, an SSD failure occurs when 1) the SSD controller has failed due to unrecoverable metadata loss or hardware damage, 2) the SSD power supply is unstable, 3) the SSD temperature is excessively high due to high performance and intensive workloads, or 4) the PCIe slot is worn-out, causing a disconnection between the kernel and SSD. We note that, when the system is alive and the SSD failure occurs, application data loss can be significant. This is because even if the SSD failure occurs, the application data can be accumulated continuously in the page cache until the failure is detected. Furthermore, the pages in the page cache

[1] We define the optimal timeout as the minimum command timeout that does not incur false-positive detection.

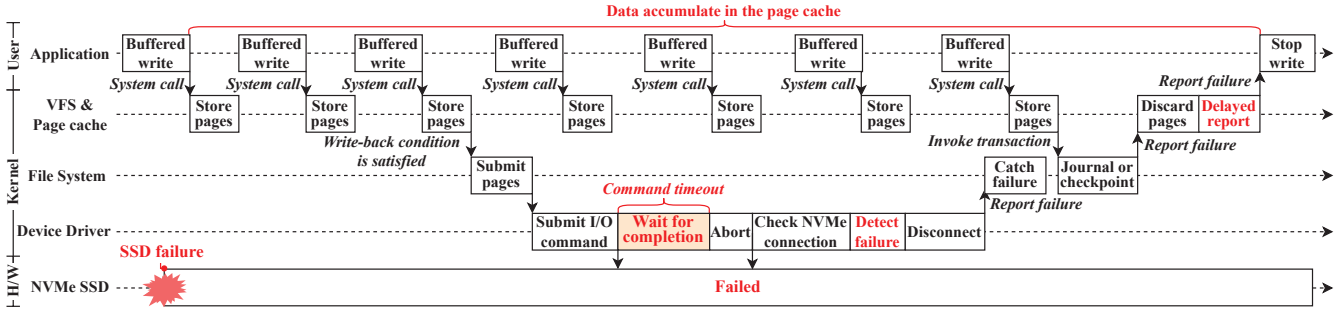


Figure 2: Example of the procedure for detecting NVMe SSD failure in the existing Linux kernel.

cannot be recovered since their corresponding files are lost, leading to user data loss. Accordingly, the current storage stack should catch the failure quickly to minimize data loss. However, according to our analysis, there are three limitations in the storage stack as follows.

2.1 Loose-deterministic Failure Check

SSD failure is passively checked via an I/O command: In the current Linux storage stack, the SSD failure can be detected only by checking whether a submitted command is completed or not within a given command timeout. For example, if the NVMe SSD fails to respond to the command within the specified timeout, the device driver checks the NVMe connection to determine whether communication with the NVMe SSD is available [11]. If it is unavailable, the device driver considers this situation as a failure. Then, it performs post-processing for the failure by disconnecting the NVMe connection and discarding the accumulated data in the page cache.

Unfortunately, this scheme may result in a significant delay due to a loose-deterministic time bound for checking the failure. Specifically, there are commonly two cases to submit I/O commands[2], such as the page cache write-back and transaction processing. However, they are performed when a specific condition is satisfied. For example, flushing the page cache and transaction processing are performed when `fsync()` is called in both cases, the ratio of dirty pages is high, and the commit/checkpoint interval has elapsed, respectively.

An example of application data loss: Figure 2 illustrates an example of checking SSD liveness or failure in the Linux kernel. As shown in the figure, even if an SSD failure occurs at an early point in time, it may remain undetected unless a command is issued. When the I/O command is issued via a satisfied write-back condition, the failure can be detected. Eventually, the kernel can only detect the SSD failure after the page cache is flushed (i.e., third buffered write), and the application data accumulated in the page cache after the failure will be lost. Consequently, because the command submission to SSD depends on various behaviors of the storage stack, a mechanism for examining SSD liveness with a strict-deterministic time bound becomes essential to minimize the data loss.

[2]We focus on the case of the write command for briefness and the target. The failure can be detected using other I/O or admin commands as well.

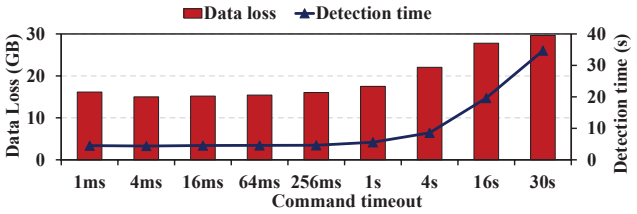
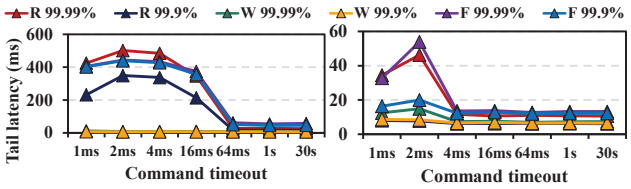


Figure 3: Data loss and failure detection time changes upon SSD failure according to command timeout.

Measuring data loss according to various command timeouts: Figure 3 shows data loss and failure detection time upon an SSD failure according to various command timeout configurations. In the figure, data loss and failure detection time decrease as command timeout decreases from 30 to 1 second which is the minimum value provided by the kernel. At the one-second command timeout, the amount of data loss caused by the SSD failure is 17.5 GB, and it indicates substantial data loss due to the slow detection time (i.e., 5.7s). For a more deep analysis, we configure the command timeouts to smaller values (e.g., 1 ms) via a small modification. However, data loss and detection time are still not improved. As explained, it is because of the loose-deterministic failure check performed by submitting I/O commands. This means that the detection of SSD failure is delayed until an I/O command is submitted, resulting in substantial data loss even with a very small command timeout. Consequently, the results show that the current scheme for examining SSD failure is not sufficient to mitigate data loss and detection time.

2.2 Fixed Command Timeout

Although a command is submitted with a strict-deterministic time bound for checking SSD failure, determining an optimal command timeout is still a challenge due to a trade-off between the amount of data loss and the false-positive effect. Specifically, a command timeout, which is larger than the actual command latency, can hinder prompt failure detection and result in larger data loss. Meanwhile, when the



(a) Samsung 980 (SSD A) (b) Samsung PM9A3 (SSD B)

Figure 4: Tail latency QoS according to the command timeouts (R: read, W: write, F: fsync).

command timeout is smaller than the actual command latency, false-positive detection occurs. As its adverse effect, the device driver issues abort commands as much as the number of expired commands, which are not completed within the timeout period, to abort each of them [57], resulting in a latency overhead.

To demonstrate the adverse effect of false-positive detection, we measure the latency of three I/O operations (i.e., read, write, and fsync) according to various command timeouts and SSD models as shown in Figure 4. In the case of SSD A, when the command timeout is smaller (i.e., from 1ms to 16ms), the tail latency of `read()` and `fsync()` system calls increases compared with larger command timeout cases (i.e., from 64ms to 30s). It is caused by the adverse effect of false-positive detection. Meanwhile, in the case of SSD B, the adverse effect occurs when the command timeout is less than 2ms. Furthermore, in both cases, the extent of the adverse effect can vary depending on the command type. For example, in the case of SSD B, when the command timeout is 2ms, the latency of the write operation is stable, meanwhile, read and fsync operations exhibit much higher latency. The results demonstrate that the command timeout should be dynamically determined according to the command type and SSD models instead of a fixed one.

2.3 Delayed Failure Notification

If a failure is quickly and precisely detected with an optimal timeout at a lower layer (i.e., device driver), the failure should be promptly notified to running applications to suspend the upcoming write operations. However, a failure detected by the lower layer is not immediately notified to the VFS layer, including the page cache; thus, the application can continue to perform its write operations.

For example, in the raw RAID cases, a failure is not notified because the device file is not removed but the RAID waits for its rebuild. For the file system cases, as presented in Figure 2, the XFS and F2FS file systems report a failure only if the critical metadata I/O has failed. Thus, only after the failure to write critical metadata (e.g., the journal superblock of XFS and the checkpoint of F2FS), the file systems can identify the failure and then stop their operations and notify the VFS layer of the failure. Specifically, in the case of EXT4, when the journal superblock write fails, EXT4 identifies the failure and remounts the device as a read-only mode. However, EXT4 does not transfer the failure to the VFS layer, and the application cannot catch the failure as shown in Figure 1. Thus, to catch the failure earlier in the VFS layer, a mechanism for fast notification is required.

2.4 Reinforcement Learning

Reinforcement learning (RL) is an algorithm that predicts the optimal action based on the current state and increases accuracy by providing negative or positive feedback based on the action results [25, 54, 61]. The feedback value reflected in the model is called a reward. A higher or lower reward value indicates a more or less accurate prediction, respectively. As a type of RL algorithm, Q-learning is a lightweight and model-free RL to learn the value of an action in a particular state [60].

This paper adopts Q-learning to predict the NVMe command latency online for two specific reasons as follows. First, to predict the NVMe command latency inside the Linux kernel at runtime, the cost of learning and prediction should be negligible to minimize interference with the target system performance. Q-learning is one of the low-cost online prediction methods [26, 27, 50]. Second, in most cases, the internal conditions and information (e.g., GC) of commodity SSDs are not open to applications (i.e., a black box). Thus, it is not easy to design a model to predict the command latency. Since Q-learning is a model-free learning algorithm, it can predict the command latency more precisely even if the SSD is a black box. For these reasons, we adopt Q-learning in our liveness examination system.

3 Design and Implementation

The goal of RL-watchdog (`RLW`) is to minimize application data loss by examining SSD liveness or failure quickly, precisely, and online. To develop `RLW`, we have to overcome the following research challenges.

- `RLW` should examine SSD liveness actively and strictly but should be lightweight to not incur much overhead to application I/O performance.
- `RLW` should precisely predict the command latency regardless of command types and SSD models without an offline pre-learning cost.
- `RLW` should quickly notify the application of a failure in an application-agnostic manner, regardless of failure post-processing of other kernel components.

3.1 Overview of RL-Watchdog (`RLW`)

To minimize application data loss, the key idea of `RLW` is to examine SSD liveness by leveraging lightweight and strict examination, an online learning technique, and a fast failure notification. Figure 5 shows the overall architecture and procedure of `RLW`.

`LWW` collects the SSD states (e.g., IOPS, in-flight I/Os, and average I/O size) from the block layer (①) and requests the

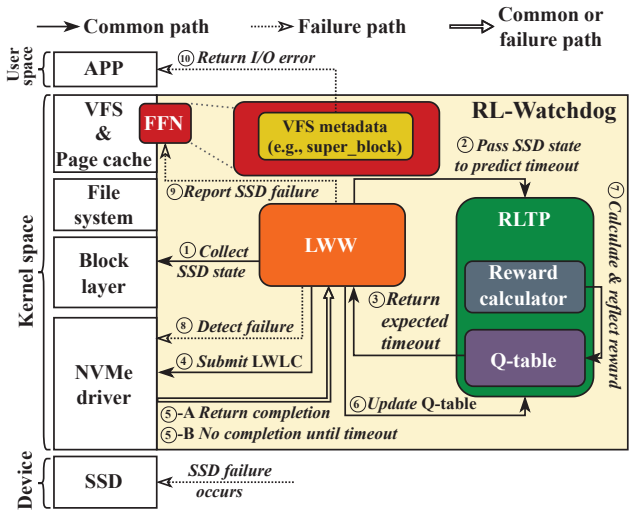


Figure 5: Overall architecture and procedure of RL-watchdog.

prediction of the command timeout value with states to RLTP (②). RLTP refers to the Q-table and returns a timeout value suitable for the given SSD states to LWW (③). Then, LWW sets the timeout value and issues the light-weighted liveness-monitoring command (LWLC) to the SSD (④). If the SSD is alive, LWLC is completed (⑤-A), and LWW requests to update the Q-table to RLTP (⑥). In RLTP, the reward is calculated through the difference between the measured LWLC latency and predicted timeout (⑦). If LWLC is not completed until the given timeout has elapsed (⑤-B), LWW checks whether an SSD failure has occurred or not (⑧). If a failure is detected by LWW, LWW reports the failure via FFN, specifically the VFS metadata (e.g., super_block) (⑨). Then, the application checks the VFS metadata resulting in the failure notification, and thus, the application can avoid upcoming writes immediately (⑩).

3.2 Light-Weighted Watchdog (LWW)

LWW monitors SSD liveness actively and quickly to determine whether an SSD is alive or not. To do this, LWW employs per-device watchdog threads (i.e., a one-to-one model) which periodically transfer LWLC which is a special NVMe command to the SSDs through the device driver. Without a response within the timeout interval, the driver aborts LWLC and checks the NVMe connection like the existing Linux kernel. In LWW, there are two key challenges to minimize the overhead of periodic monitoring: 1) devising LWLC to be a universal and lightweight command and 2) determining the optimal interval between issuing LWLCs to minimize performance interference. We describe how to handle these challenges as follows.

Light-weighted liveness-monitoring command (LWLC): It is a challenge that the periodic monitoring technique should be universal and lightweight. To handle this challenge, a potential approach defines a new NVMe command and implements it in the SSD firmware and NVMe device driver. However, modifying SSD firmware is practically impossible unless the manufacturer's assistance, and command processing of the latest SSDs is based on the hardware [46], making it more difficult. Furthermore, although the new NVMe command is defined, it is inapplicable to many commodity SSDs on existing storage systems. As an alternative approach, we can utilize an existing NVMe I/O command. However, it shares the NVMe I/O command queue with the normal I/O commands, resulting in a large interference with the normal I/O operations.

To handle this issue, we employ an admin command by using a spare opcode (e.g., 0xFF) that is not defined in the NVMe specification as an LWLC. This approach offers two key advantages. First, regardless of the SSD models, it can be applied to all commodity NVMe SSDs by handling the spare opcode to examine SSD liveness. Second, it requires minimal SSD internal resources. For example, when the command arrives at the SSD, the controller can only check the opcode and complete the command without any further actions. By doing so, we devise LWLC to be more universal, lightweight, and have minimal impact on normal I/O operations.

Heartbeat interval (HBI): As a second challenge, to minimize performance interference by LWW, it is important to set an interval between issuing LWLCs and understand the relationship between the interval and command timeout. Thus, we define the interval as HBI which is the time between the completion of LWLC and the re-submission of LWLC. An excessively short HBI increases the frequency of issuing LWLC, thereby detecting SSD failures early but reducing the normal I/O performance. Meanwhile, an excessively high HBI decreases the frequency of issuing LWLC, thereby less affecting the normal I/O performance but causing a delayed detection of SSD failures. Through the experiments, we observe an optimal HBI of 256 ms with a negative effect of less than 0.42% on performance. The detailed experimental results are explained in Section 4.7.

Procedure of LWW: Figure 6 and Procedure 1 describe how LWW works and interacts with RLTP. When LWW starts, it receives the list of SSDs to be monitored and HBI as parameters (line 1). Then, it creates per-device watchdog threads in an SSD list (①, lines 2–4). Each watchdog thread starts monitoring the SSD liveness until LWW is removed or detects SSD failure (line 7). First, each thread collects the current state (S) of its target SSD from the block layer (line 8). S consists of the I/O information of in-flight I/Os, IOPS, average I/O size, etc., which are features used in RLTP to learn and predict the command timeout (T). Since each thread utilizes this I/O information which is already collected in the existing block layer, there is no additional overhead caused by the collection. Then, each thread transfers the state to RLTP to get a predicted command timeout ($\hat{T}$) based on the state (②, line 9). Subsequently, each thread submits LWLC with $\hat{T}$ to its corresponding SSD (③, line 10), waits for LWLC completion, and stores its actual command latency (T_A) (④, lines 11–12).

If there is no response within $\hat{T}$ (i.e., the command is ex-

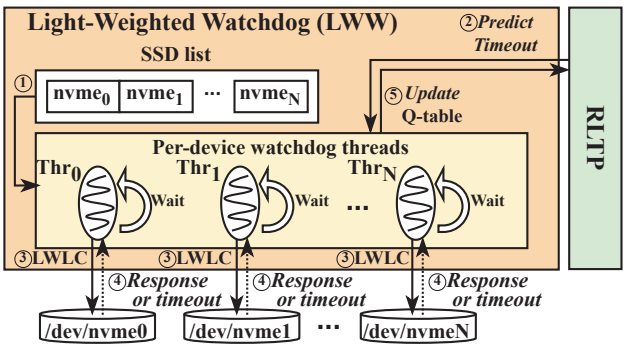


Figure 6: Procedure of LWW (Thr: thread).

Proc. 1 A C-like pseudo-code of SSD liveness examination in LWW

```
1:  function LWW((ssd_list, HBI))
2:      for ssd in ssd_list do                          ▷ Per-device watchdog thread
3:          create_thread(do_watchdog, ssd, HBI)
4:      end for
5:  end function
6:  function DO_WATCHDOG(target_ssd, HBI)
7:      while LWW not stopped do
8:          S ← State of target_ssd from the block layer;
9:          (T̂, Qptr) ← RLTP.Predict(S);
10:         Submit LWLC with timeout T̂;
11:         Wait for completion of LWLC;
12:         T_A ← latency of LWLC;
13:         if command is expired then
14:             if !is_alive(target_ssd) then
15:                 Stop the watchdog thread (target_ssd)    ▷ SSD failure
16:             end if
17:         end if
18:         if is_alive(target_ssd) then
19:             RLTP.UpdateQtable(prev_Qptr, Qptr, prev_T_A, prev_T̂)
20:             prev_T_A ← T_A                          ▷ Save previous values
21:             prev_T̂ ← T̂
22:             prev_Qptr ← Qptr
23:         end if
24:         wait(HBI)
25:     end while
26: end function
```

pired) (line 13), the NVMe device driver checks the NVMe connection (line 14). If a failure is found, the SSD connection is disconnected, and the watchdog thread of the target SSD is stopped (line 15). Then, LWW gets a failure code from the device driver and then reports the failure to FFN. Otherwise, each thread transfers its corresponding T_A and $\hat{T}$ to RLTP to update the Q-table (⑤, line 19). Then, it updates the previous latency, predicted timeout, and Q-value (i.e., $prev\_T_A$, $prev\_\hat{T}$, and $prev\_Qptr$) by the current T_A, $\hat{T}$, and $Qptr$, respectively, to use them for the next step (lines 20–22). Finally, each thread waits for the optimal HBI (line 24) and repeats the above procedure.

3.3 Reinforcement Learning Timeout Predictor (RLTP)

We propose RLTP to predict a command timeout (i.e., LWLC timeout) according to the current SSD state at runtime without an offline pre-learning process. We note that I/O command

Table 2: Correlation coefficient between the LWLC latency and the features of SSD states (FIO (GC): GC is invoked by FIO, W: write, R: read).

Features	Video server	File server	YCSB	FIO (GC)	FFSB
In-flight I/Os	0.06	0.13	0.63	0.03	-0.005
IOPS (W)	-0.23	-0.06	0.34	-0.01	0.023
Avg. size (W)	0.38	-0.76	0.51	-0.03	0.025
IOPS (R)	0.007	-0.005	-0.11	-0.01	0.001
Avg. size (R)	-0.002	0.002	0.04	-0.01	0.001

(e.g., read, write, and flush) latency prediction can be challenging and time-consuming work because of the complex internals of modern SSDs [29, 31]. However, instead of common I/O commands, RLW targets to predict the timeout of LWLC which induces the SSD controller to simply check the opcode as described in Section 3.2. Thus, predicting the LWLC timeout is much easier than that of other commands, as it is less affected by the SSD's complex internals. This design decision supports the viability of employing Q-learning [60] to predict the LWLC timeout. We match the Q-learning components of the action, state, and Q-value to the LWLC latency, current SSD states, and expected gain, respectively. A detailed explanation is described below.

Feature selection: To apply Q-learning, we select features for prediction among the various features of the current SSD state such as in-flight I/Os, the write/read IOPS, and the average write/read size. To do this, as shown in Table 2, we measure the correlation coefficient between the LWLC latency and the features under various workloads. The features of in-flight I/O, the write IOPS, and the average write I/O size are correlated with the LWLC latency. For example, when the number of in-flight I/O increases or write IOPS increases, since SSD may process a large number of pending I/O commands or normal I/O operations, respectively, it leads to an increase in the LWLC latency. Also, if in-flight I/Os are high and the write IOPS is low, LWW can infer that GC may be running inside the SSD [31]. Third, when the average write I/O size increases, the time of processing one command increases, resulting in increasing LWLC latency. Finally, we observe that both the read IOPS and the average read size have almost no relationship with the LWLC latency. Thus, we exclude them from the features for learning.

Through the correlation coefficient, we discover an interesting fact that we did not expect. The fact is that the correlation coefficients which we expect as positive values are observed as negative values in some workloads. For example, the correlation coefficient in the case of the average write size in the fileserver is lower than -0.5. This means that the LWLC latency increases even if the average write size decreases. The reason for this opposite result to the expectation is from the SSD power-saving mode [4]. For example, if few I/O requests occur, the SSD enters a low power mode to prevent power wastage. Thus, during the low power mode, the frequency of the controller inside the SSD is lowered, resulting in a

Table 3: Range of features to represent SSD state.

In-flight I/Os	Write IOPS	Avg. write Size	LWLC latency
< 13	< Max/256	< 8 KB	< 1 ms
>= 13	< Max/16	< 32 KB	< 4 ms
	>= Max/16	< 128 KB	< 16 ms
		>= 128 KB	>= 16 ms

Table 4: Reward according to the latency prediction results.

Prediction result	False positive (Underestimated)	Overestimated	Correct
Reward value	-1	-0.5	1

higher LWLC latency. Even if there can be this unexpected case, since RLTP continuously updates the values of the Q-table according to the current SSD state, RLTP can predict the LWLC timeout.

Design of Q-table and reward: If the values of the selected features are represented as individual integers, the size of the Q-table becomes unacceptably large, and learning slows down, making proper prediction impossible. Thus, we establish a range for each feature to design a Q-table with an appropriate size. To do this, we analyze the sensitivity against various combinations of each feature and its range based on the representative SSD states and choose them like the previous study [26] as shown in Table 3. The size of the resulting Q-table per SSD is 384 bytes when one entry size is 4 bytes. As a result, memory consumption is negligibly small.

Table 4 lists the determined reward according to the prediction results. As shown in the table, if the predicted timeout is shorter than the actual timeout (i.e., a false positive occurs), since it is an incorrect prediction result, the reward has the lowest value of -1. Meanwhile, if it is not a false positive but the predicted timeout is too long, we impose the penalty of -0.5 to give less penalty than the case of a false positive. If the predicted timeout and the actual one are within the same range, it is considered a correct prediction, and a value of 1 is used as a reward value. By doing so, we increase the prediction accuracy of the LWLC timeout according to the current SSD state.

Procedure of RLTP: We describe the procedure of RLTP in Figure 7 and Procedure 2. The figure presents a watchdog thread and table for each SSD (i.e., Thr_0:Q_0, Thr_1:Q_1, ... Thr_N:Q_N). As shown in the procedure, there are the main operations of Predict() and UpdateQtable() of RLTP. Predict() receives the SSD states to be used as features from LWW and predicts an LWLC timeout (①, line 1). Each SSD state is converted into its corresponding index to access the Q-table (line 2). The Q-table constitutes a four-dimensional array, but for simplicity, we depict it as three-dimensional except for in-flight I/O. After accessing the Q-table based on the index of each state, the entry with the highest Q-value for each latency range is found and transferred to LWW (②, lines 3–12). In the example in the figure, since the highest Q-value is 0.2 in the Q-table, thus, index 1 is returned. Additionally, the found Q-value ($Qptr$) is also returned for updating the

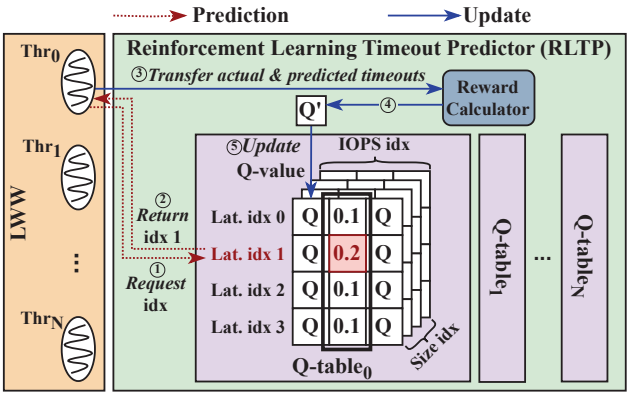


Figure 7: Procedure of RLTP (Thr: thread).

Proc. 2 A C-like pseudo-code of prediction and update in RLTP

```
1:  function PREDICT((iops, size, inflights))
2:      idx_o, idx_s, idx_i ← Get_Idx(iops, size, inflights)
3:      for idx_t = 0, 1, ... MAX_IOPS_RANGE − 1 do
4:          Q_value ← Q_table[idx_o][idx_s][idx_i][idx_t]
5:          if max < Q_value then
6:              max ← Q_value
7:              max_idx ← idx_t
8:          end if
9:      end for
10:     Qptr ← &max ▷ Get the address of the max Q-value for the update
11:     T̂ ← Get_T̂(max_idx)
12:     return (T̂, Qptr)
13: end function
14: function UPDATEQTABLE((Qptr, next_Qptr, T̂, T_A))
15:     if T̂ == T_A then
16:         reward ← 1                                  ▷ No false positive
17:     else
18:         if T̂ < T_A then
19:             reward ← −1                             ▷ False positive
20:         else
21:             reward ← −0.5                           ▷ No false positive
22:         end if
23:     end if
24:     Q_value ← ∗(Qptr)          ▷ Get the Q-value from the address
25:     next_Q_value ← ∗(next_Qptr)
26:     ∗(Qptr) ← Q_value + α(reward + γ ∗ next_Q_value − Q_value)
27: end function
```

Q-table later.

Next, UpdateQtable() updates the Q-values based on the reward. RLTP uses the actual command latency (T_A) and its predicted command timeout ($\hat{T}$) to calculate the reward (③, line 14). If the prediction is correct (i,e., T_A is the same as $\hat{T}$), a high reward of 1 is given (lines 15–16). If T_A is larger than $\hat{T}$ (i.e., a timeout occurs, but the SSD does not fail), it is determined as a false positive. This means that the LWLC timeout was predicted to be too short, and the lowest reward of -1 is given (lines 18–19), otherwise, -0.5 is given (line 21). Then, the predicted Q-value ($Qptr$) and the next predicted Q-value ($next\_Qptr$) are used to update the Q-value (lines 24–25). The new Q-value is calculated according to the Bellman equation [16] (④, line 26). Finally, the Q-value is updated in its location in the Q-table (e.g., the figure depicts an example of updating the Q-value from Q to Q' at size index 3, IOPS

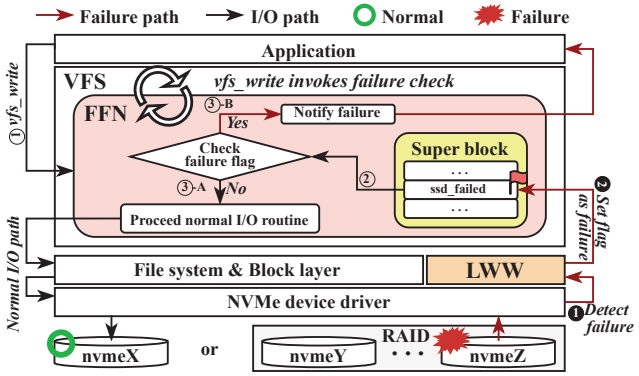


Figure 8: Procedure of FFN.

index 0, and latency index 0 (⑤)).

3.4 Fast Failure Notification (FFN)

Even though LWW can quickly detect an SSD failure event, the data loss can be still large unless the failure event is instantly notified to the upper layer. To minimize data loss further, we introduce the FFN mechanism by using the VFS layer to immediately notify applications of the failure detected by LWW. More specifically, we add a failure flag in an unused area of the VFS metadata (i.e., super_block) to notify the applications of the failure. If LWW detects the failure, it sets the flag to represent the failure. Thus, when starting I/O operations, the mechanism allows the applications to check the flag to determine whether the current SSD has failed. It is a simple but highly compatible and reliable method since the applications always access the VFS regardless of the file and storage configurations including the raw device, file system, single device, or RAID. Thus, this mechanism can quickly prevent applications from performing further write operations upon failure. FFN utilizes an existing failure code (i.e., EIO) to stop the application, requiring no application modification. However, if an SSD failure needs to be classified separately in the application, a new failure code should be defined and reflected in the application.

Procedure of FFN: Figure 8 shows the procedure of FFN. When an application starts the I/O operation via vfs_write(), the application checks whether a failure occurs via FFN (①,②). If there is no failure, the write operation of an application is processed normally through the VFS layer and storage stack (③-A). Otherwise, as soon as LWW detects the failure (❶), it immediately reports it to FFN by setting the failure flag (ssd_failed) in the super_block of the VFS layer (❷). Thus, the application can catch the failure of the current SSD and stop its write operation (③-B). By doing this, a failure can be notified to applications immediately after being detected by LWW.

3.5 Putting It All Together

Figure 9 depicts the timeline of RLW, where LWW periodically monitors the SSD liveness by submitting LWLC with its predicted timeout via RLTP. Specifically, in the first case, LWW receives the predicted timeout ($\hat{T}_1$) from RLTP and submits LWLC to the SSD. In this case, without failure, LWW completes LWLC normally within the given predicted timeout. Since there is no false positive, RLTP updates the Q-table with the reward of -0.5 (i.e., $T_{A1} < \hat{T}_1$). In the second case, even if the SSD is still alive, a command timeout occurs since the actual command latency (T_{A2}) is longer than the predicted timeout ($\hat{T}_2$) (i.e., a false positive), the device driver aborts the expired command and checks the NVMe connection, and RLTP updates the Q-table with the reward of -1.

In the third case, the SSD failure causes a command timeout. Thus, LWW detects the failure, notifies it via FFN, and stops monitoring for the failed SSD. Then, upcoming applications stop their write operations via FFN. We note that SSD failure is detected even if no I/O command is submitted to SSD because RLW can actively examine SSD liveness. The application loses only two buffered writes (i.e., the fourth and fifth) accumulated after SSD failure. The SSD failure is notified to the application immediately after the application submits the sixth buffered write, thereby further minimizing data loss. As a result, in contrast to the existing scheme, as shown in Figure 2, RLW can examine SSD liveness actively with a predicted timeout and quickly notify the application of the failure to minimize the application data loss.

4 Evaluation

We evaluate RLW by answering the following questions:

- How much is RLW effective on various workloads, various storage configurations, and different SSD models? (§4.1, §4.2, and §4.3)
- Is RLW effective even for various failure points? (§4.1)
- How much do the periods of fsync() impact the effectiveness of RLW? (§4.1)
- How much does each technique contribute to reducing data loss? (§4.4)
- How much RLTP predicts the command timeout well even in complex SSD internal operations (e.g., GC)? (§4.5)
- How much does false-positive detection impact the performance? (§4.6)
- How much overhead is caused by RLW? (§4.7)

Experimental setup: We use a server machine with Intel Xeon E5-2650 CPU (24 cores and 48 threads) with 160 GB DRAM. For storage, we employ two SSD models: Samsung 980 [2] (SSD A) and Samsung PM9A3 [3] (SSD B). We use SSD B unless stated otherwise. The write latency (us) / throughput (KIOPS) of 4 KB is 55.6 / 278 and 14.7 / 352 in SSD A and B, respectively. In addition, we use three SSDs of each model for all RAID configurations. We run Ubuntu 20.04.3 LTS with the Linux kernel 6.0.0. Unless stated otherwise, we set the command timeout to 1 second which is the shortest configurable timeout provided by kernel.

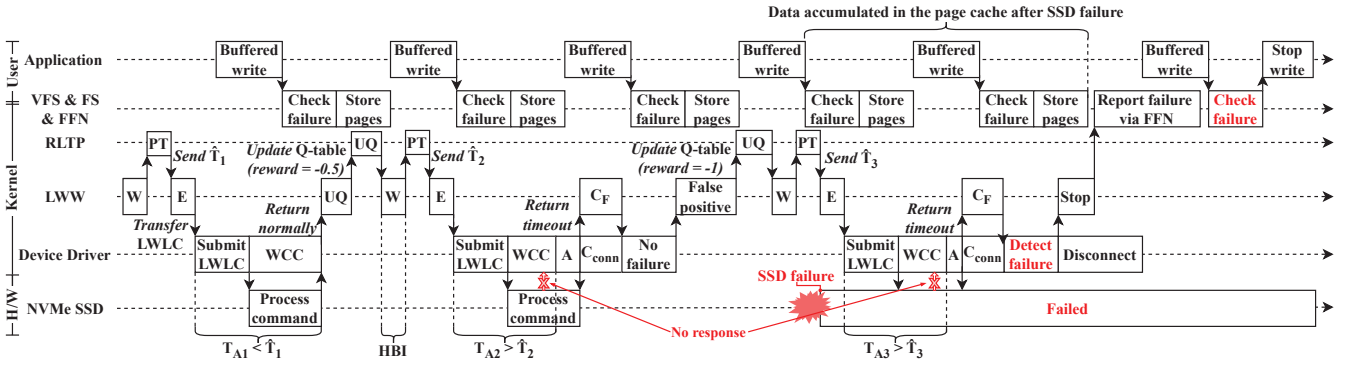


Figure 9: Timeline of RLW with three techniques (PT: predict timeout, W: wait HBI, E: examine, A: abort command, WCC: wait command completion, UQ: update Q-table, C_{conn}: check NVMe connection, C_F: check SSD failure, T_A: actual command latency, $\hat{T}$: predicted command timeout).

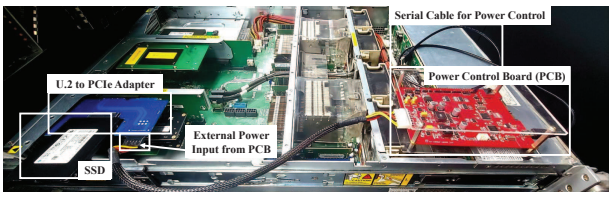


Figure 10: Experimental environment.

For a more comprehensive evaluation, we use 1us as a minimum timeout by slightly modifying the timeout management code in the block layer. To evaluate RLW on various storage configurations, we use a raw device and three common file systems (i.e., EXT4, F2FS, and XFS) under different software RAID configurations (i.e., RAID 0/1/5).

Injection of an SSD fault: To generate an actual SSD fault, we use a specially designed power control board (PCB) as depicted in Figure 10. The system with PCB controls the power supply of the SSD independently of the host system power, and the system can supply or cut off the power through serial communication. Moreover, we use adapters that convert the M.2 or U.2 form factor to PCIe, and these adapters support receiving external power from the PCB instead of a PCIe slot. Using these devices, we inject a fault to the NVMe SSDs at a fixed point (i.e., 2s) unless stated otherwise or at various points to generate the SSD failure. In the RAID evaluation, we select the number of SSDs to fail according to the RAID configuration because each RAID configuration has a different fault tolerance degree. For example, we inject the faults of 1, 3, and 2 SSDs for RAID 0, 1, and 5, respectively.

Workload: We use FIO [10] as a micro benchmark, filebench [55] and FFSB [49] as macro benchmarks, and RocksDB [1] as a real-world application with two benchmarks (i.e., DBBench and YCSB [5]). To evaluate the prediction accuracy during a GC procedure, we induce the GC procedure by performing random write twice as much as the device size from a clean state via FIO (notated as FIO (GC)). Since FIO (GC) continues to generate GC operations inside SSD, we believe that FIO (GC) is the most suitable workload to evaluate the prediction accuracy in the worst situation where SSD internal resources are highly utilized.

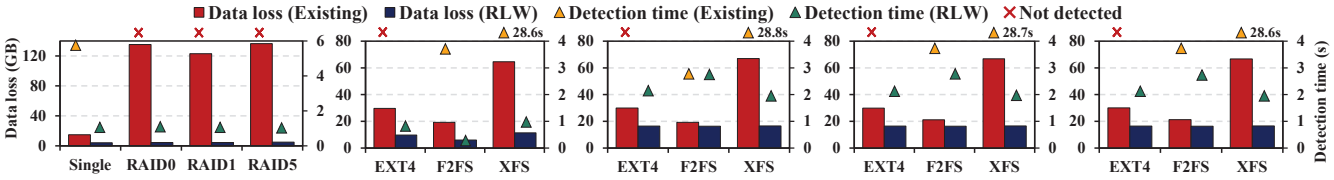
Measuring data loss and failure detection time: For FIO, which supports data verification, we can easily measure the data loss for all the cases by calculating the difference between "the amount of written data until an application detects the failure" and "the amount of verified data after supplying power again". However, the data loss on other benchmarks (e.g., filebench, DBBench, and YCSB) cannot be measured easily because they do not support data verification. Instead of adding verification logic to each benchmark which would be substantial work, we measure each amount written by an application (until the application detects the failure) in the existing scheme and RLW, and calculate the difference between them. The written amount difference is the same as the data loss difference. To get failure detection time, we measure the period from the point of failure injection to the point of failure detection on the application side.

4.1 Micro Benchmark

Data loss on various configurations: Figure 11 indicates how much RLW reduces the application data loss and the failure detection time upon SSD failures on various configurations when running random writes via FIO. We use eight threads using a 20 GB file per thread and a 4 KB request size.

For the raw device depicted in Figure 11a, RLW reduces the data loss by 72.9%, 96.7%, 96.4%, and 96.3%, on a single SSD and RAID 0/1/5, respectively. On the aspect of failure detection time, RLW reduces it by 86.7% compared with the existing scheme in the case of a single SSD. The red-colored 'X' (Not detected) flag indicates that the failure cannot be notified to the application even if the application terminates its execution (100 seconds). Therefore, we cannot measure the failure detection time in these cases of raw RAID configurations. Meanwhile, RLW enables the application to detect the failure within a similar time to that of the single SSD case, ranging from 0.73 to 0.76 seconds. These results imply that RLW successfully reduces data loss regardless of raw device storage configuration (i.e., a single device or RAID).

Figures 11b, 11c, 11d, and 11e depict the data loss and



(a) Raw device (b) Single SSD (c) RAID 0 (d) RAID 1 (e) RAID 5

Figure 11: Impact of RLW on data loss and failure detection time in SSD B.

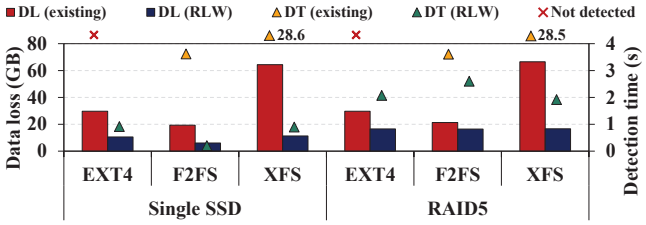


Figure 12: Impact of RLW on data loss and failure detection time in SSD A (DL: data loss, DT: detection time).

failure detection time on the three file systems under the single SSD and three RAID configurations using SSD B. RLW reduces the data loss by up to 82.4% and 75.2% on a single device and RAID cases, respectively. The failure detection time is reduced by up to 97.9% and 93.8% on a single device and RAID cases, respectively. Similar to raw RAID cases, the existing scheme on EXT4 does not notify the application of failure. Thus, the failure detection time cannot be measured in all cases with EXT4. For XFS, the existing scheme exhibits significantly high levels of data loss and failure detection time due to a relatively long journal flushing period of XFS (i.e., 30 seconds). Even though the results show that the impact of RLW depends on various factors, such as storage configuration and error handling policy of various layers, they demonstrate the effectiveness of RLW even in various file systems and storage configurations.

To demonstrate the effectiveness of RLW on various SSDs, we evaluate SSD A, as illustrated in Figure 12. As depicted in the figure, improvement degrees by RLW are similar to the results of SSD B. RLW reduces the data loss by up to 82.5% and 75% and detects the failure more quickly by up to 97.4% and 93.7% on a single device and RAID 5 cases, respectively, compared with the existing scheme. According to the results, we show that RLW is effective on both SSD models.

Data loss according to various failure points: To evaluate the impact of RLW on the various failure time points, ranging from 1 second to 40 seconds, we measure the data loss and detection time on F2FS with a single device as shown in Figure 13. In this evaluation, the application performs random writes to files with periodical file creation and deletion until the failure is detected. For example, when the failure occurs at 5 seconds and is detected at 8 seconds, the application performs the I/O operations for 8 seconds in total.

Overall, after a failure occurs, RLW detects the failure within time ranges from 0.5 to 2.1 seconds, meanwhile, the existing scheme detects the failures within time ranges from 2.8 to 7.7 seconds. It shows that RLW is more stable and faster than the existing scheme. However, data loss increases in RLW when the failure point is moved from 1 to 9 seconds, and this pattern repeats from 10/22/33 to 21/32/40 seconds. The rationale behind these results is that, as time elapses, the page cache becomes almost full. This means that even if a failure point is identified quickly, the loss of pages that are already stored in the page cache is inevitable. Also, the detection time in the existing scheme decreases at the specific time point (e.g., 9, 21, 32, and 40 seconds) since the flushing operations triggered by the almost full page cache can detect the failure relatively more quickly. After a file is deleted, the page cache is emptied and newly accumulated, resulting in the RLW being effective again. Consequently, RLW reduces the data loss regardless of failure points even if its effectiveness can be reduced.

Data loss according to various fsync periods: To understand the effect of `fsync()` periods in RLW on various file systems with RAID 0, we measure the data loss according to the periods. In Figure 14, for `fsync()` at every 128K write operations, the effectiveness of RLW is low because this short `fsync()` period flushes the accumulated pages frequently. Meanwhile, for 512K and more I/Os, the effectiveness of RLW significantly increases. These results demonstrate that adopting an optimal fsync period can be challenging, considering the trade-off between data loss and performance.

4.2 Macro Benchmark

To evaluate RLW in more realistic workloads, we measure the reduction of data loss and failure detection time using fileserver and videoserver in filebench and FFSB in F2FS on a single SSD and RAID 5. Figure 15 illustrates the data loss difference between the existing scheme and RLW and their failure detection time. For a single SSD, RLW significantly reduces the data loss by 300GB, 19GB, and 162GB for fileserver, videoserver, and FFSB, compared with the existing scheme, respectively. The rationale behind the reduced effectiveness of RLW on videoserver is that this workload has higher read rates with large sequential reads. Similarly, in RAID 5 configuration, RLW reduces the data loss by 296GB, 31GB, and 48GB for fileserver, videoserver, and FFSB, compared with the existing scheme, respectively. The overall impact of RLW is similar on both single device and RAID 5 cases. This result demonstrates that RLW can detect failures faster than the existing scheme, even in more realistic workloads.

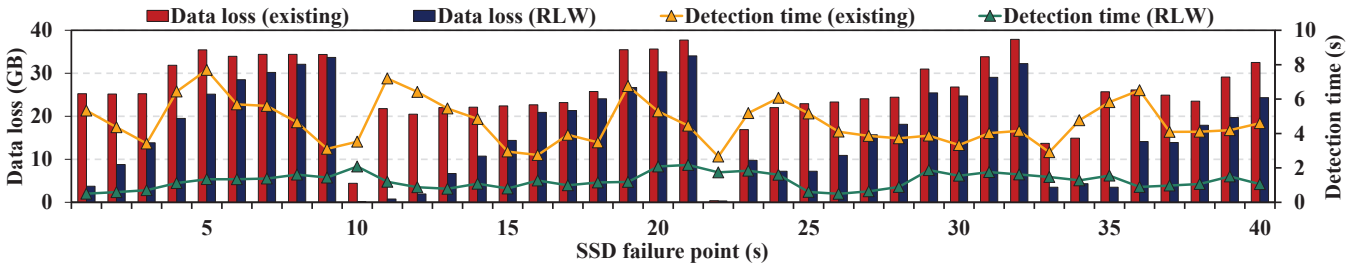


Figure 13: Data loss and failure detection time change according to different SSD failure points.

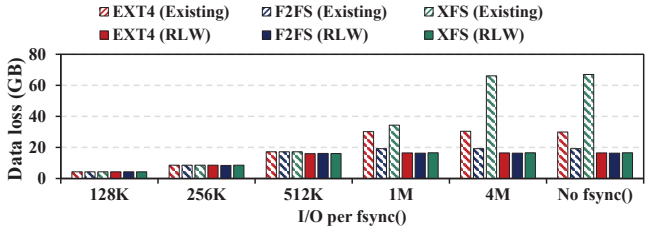


Figure 14: Data loss in various `fsync()` periods on the various file systems.

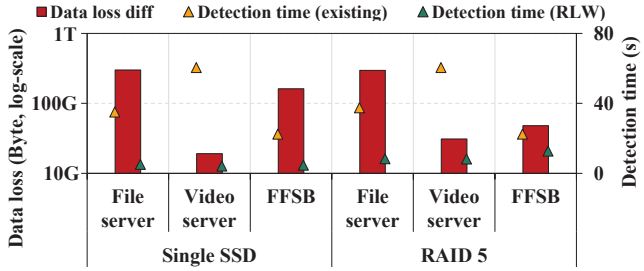


Figure 15: Data loss on macro benchmarks.

4.3 Real-world Application

To evaluate `RLW` with a real-world application, we employ a key-value store, RocksDB [1] with two benchmarks including DBBench and YCSB [5] as shown in Figure 16. We run the fill-random and update-only/workloadA for DBBench and YCSB, respectively. In the figure, `RLW` decreases the I/O loss by up to 400 thousand operations in both cases of a single SSD and RAID 5, compared with the existing scheme, respectively. Especially, the impact of `RLW` is large on DBbench since the write ratio is relatively higher at DBBench workload. Also, `RLW` reduces the detection time by up to 45.7% and 53.0% in the case of single SSD and RAID 5 compared with the existing scheme, respectively. These results demonstrate that `RLW` can be effective in real-world applications by minimizing the key-value data loss.

4.4 Impact of Individual Techniques

Figure 17 presents the reduction of data loss and failure detection time according to individual techniques for various storage configurations. `LWW` reduces the data loss and detection time by up to 57.6% and 84.4%, respectively, compared with the existing scheme. However, for raw RAID 0 and 5,

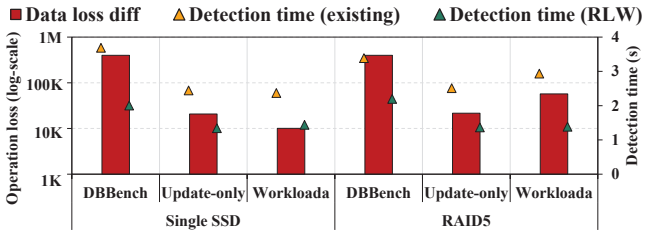


Figure 16: I/O loss in a real-world application (RocksDB).

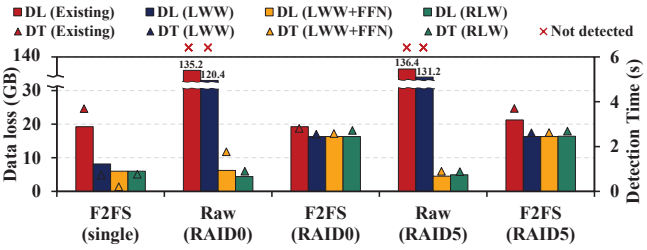
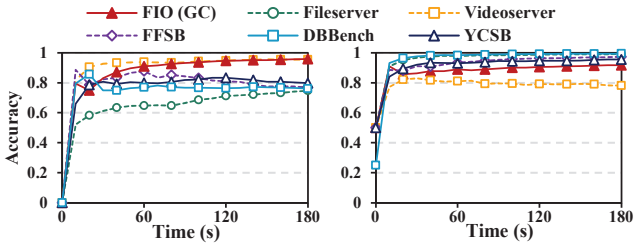


Figure 17: Impact of individual techniques (DL: data loss, DT: detection time).

`LWW` does not work well because the RAID layer blocks failure notifications so that the application cannot recognize any failure. In both cases, additionally applying `FFN` on `LWW` enables the application to recognize the failure and data loss is significantly reduced by up to 87.1%. The effectiveness of `FFN` is relatively lower on F2FS than raw RAID cases, however, data loss is reduced by up to 26.5%. Finally, when the `RLTP` is additionally applied, which is denoted as `RLW`, the results indicate that `RLTP` predicts the command timeout with minor overhead.

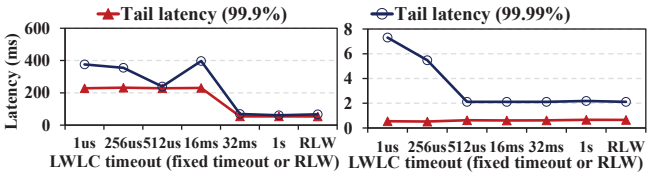
4.5 Prediction Accuracy

Figure 18 depicts the prediction accuracy of `RLTP` and its convergence point for two SSD models. The prediction accuracy reaches up to 99.8% for the DBBench workload. In most cases, the accuracy on SSD B is higher and converges faster than SSD A because the latency of SSD B is more stable and lower than that of SSD A. Additionally, the accuracy in most workloads converges within at least 120 seconds except for fileserver and videoserver on SSD A and B, respectively. Fileserver includes a relatively higher read I/Os ratio than other workloads. Thus, this read/write mixture pattern further



(a) Samsung 980 (SSD A) (b) Samsung PM9A3 (SSD B)

Figure 18: RLTP accuracy timeline on various workloads.



(a) Samsung 980 (SSD A) (b) Samusung PM9A3 (SSD B)

Figure 19: Tail latency QoS according to the timeout.

affects the performance of SSD A (due to the less stable latency) than that of SSD B. Meanwhile, videoserver includes a delay operation that further affects SSD B (with a higher throughput) compared to SSD A because this delay can omit more I/Os per unit of time as the throughput is higher.

4.6 Impact of False-Positive Failure Detection

Figure 19 depicts the performance impact of fixed LWLC timeout and predicted LWLC timeout via RLW on two SSD models to show the side effect of false positive detection by the fixed LWLC timeout and the importance of timeout prediction via RLTP. In the case of SSD A, as expected, the relatively large timeouts ranging from 32ms to 30s do not affect the tail latency. However, smaller timeouts ranging from 1us to 16ms increase the tail latency by up to 522% compared with RLW since the small timeouts incur the false-positive detection. Meanwhile, in the case of SSD B, the timeouts ranging from 1us to 256us increase the tail latency by up to 235% compared with RLW. Note that the timeouts incurring the false-positive detection can be different according to the SSD models (i.e., 16ms and 256us on SSD A and B, respectively).

On the other hand, RLW does not affect the tail latency on both SSD models because no false positive occurs while predicting the timeout at run-time even without a pre-learning job. Consequently, these results demonstrate that fixed small timeouts can affect the performance due to the false-positive detection, meanwhile, RLW less affects the performance by correctly predicting timeout based on current SSD states.

4.7 Performance Overhead of RLW

Table 5 lists the performance overhead of RLW. Write operations are performed while RLW submits LWLC with different HBI on both SSD models. When HBI is 1ms, the throughput

Table 5: Throughput according to different HBI values.

HBI (ms)	1	4	16	64	256	No RLW
SSD A (KIOPS)	184.9	209.1	241.4	268.2	275.9	278.5
SSD B (KIOPS)	346.3	351.9	346.9	350.8	346.7	352.7

Table 6: Computation overhead of RLTP.

I/O conditions	No I/O		Busy I/O	
	No RLTP	With RLTP	No RLTP	With RLTP
CPU utilization	0.03%	0.05%	14.06%	14.09%

drops by 33.3% on SSD A. As HBI increases, the throughput degradation becomes smaller. To minimize the side effects, we set HBI to 256ms in the evaluation, decreasing the throughput by only 0.42%. Meanwhile, interestingly, the throughput of SSD B is not affected by HBI. This means that the overhead of RLW is negligible on SSD B even if LWLC is submitted frequently. These results demonstrate that RLW is lightweight while quickly detecting SSD failure.

Furthermore, to show the computation overhead of RLTP, we measure the CPU utilization with or without RLTP on two different I/O conditions as shown in Table 6. RLW employs a sufficiently large fixed timeout (i.e., 1 second) when RLTP is not involved, and HBI is set to 1 ms to trigger RLTP frequently. As shown in the table, the CPU utilization in the case of a busy I/O condition increases by 14% compared with no I/O condition. However, the CPU utilization increased by RLTP is negligible in both cases. This result demonstrates that RLTP has lightweight computation.

5 Discussion

5.1 Position of RLW

RLW can be effective and collaborate with existing schemes in diverse SSD-based systems such as distributed and standalone systems. For example, RLW can enhance the SSD failure management in a distributed system with a redundancy scheme by detecting the failure within a replica node quickly which is an important issue as described in the previous studies (EAFR [35] and Ho *et al.* [20]). Furthermore, in a standalone system (e.g., fileserver and desktop) with a data-intensive workload [52] similar to our experimental environment, RLW can be also effective to mitigate the data loss.

5.2 Advantage of Kernel-based Approach

RLW adopts a kernel-based approach to leverage three advantages. First, the kernel-based approach is closer to the SSD than the application-based one, leading to faster failure detection. Second, it enables application-agnostic solutions without requiring application modifications, resulting in the easy utilization of RLW. Lastly, it does not require issuing a system call. Meanwhile, since an application-based approach requires frequent system calls, it can lead to high overhead. Therefore, we choose the kernel-based approach to make RLW more efficient.

6 Related Work

Detecting and handling SSD failures: To investigate SSD failures, some studies [6, 7, 71, 72] have adopted a special device as a PCB that can control the power supplied to the SSDs, as in our study. Ahmadian *et al.* [7] analyze the effect of various I/O patterns on data loss while controlling the power of a SATA SSD. Ahmadian *et al.* [6] and Zheng *et al.* [71, 72] classify various SSD failure types when power faults occur. Our work is similar to these studies [6, 7, 71, 72] in terms of analyzing SSD failure with the power control board to inject a more realistic fault. Meanwhile, we target reducing data loss at run-time.

Shi *et al.* [51] discover a bug that causes data loss when `sync()` and power faults occur. In addition, Jaffer *et al.* [22] evaluate the reliability of an SSD by classifying the data loss problems. Huang *et al.* [21] present the concept of a metastability failure, a point at which a large-scale cluster cannot be automatically returned to a normal state. In addition, Lu *et al.* [37] analyze and classify symptoms from the logs of storage clusters to the failures of NVMe SSDs.

Narayanan *et al.* [42] characterize and analyze the failures of millions of SSDs and the reliability of data centers. Furthermore, Mahdisoltani *et al.* [39] predict the sector failure of SSDs and propose use cases to mitigate the performance drop caused by failure handling. Our study aligns with these studies [21, 22, 37, 39, 42, 51] regarding investigating SSD failure problems. However, we focus on handling SSD failures instead of the discovery or classification of symptoms caused by SSD failures.

Kadekodi *et al.* [24] present Tiger which estimates the failure rate and dynamically configures RAID stripe by changing the ratio of parity devices to improve space efficiency and fault tolerance. Our study aligns with this study [24] in terms of reducing data loss caused by SSD failure. Even with an advanced RAID scheme, the data loss problem in the page cache still remains.

IronFS [48] and EIO [17] treat delayed error propagation. They inspire our study, meanwhile, `RLW` targets to notify applications of SSD failures using the VFS layer quickly. Chronos [15] and SafeTimer [38] have investigated to detect errors in distributed systems by heartbeat schemes between nodes. `RLW` can collaborate with them. For example, error detection schemes in distributed systems can propagate an SSD failure that occurred in a node to a master node more quickly with the assistance of `RLW`. Furthermore, the node with `RLW` can protect against application data loss from upcoming requests.

Prediction models in SSDs: Kang *et al.* [26] propose a GC scheduler to predict the idle time in an SSD through RL. Kurniawan *et al.* [31] build a deep-learning model to learn latency logs on various workloads and SSD models to predict I/O latency. Furthermore, LeaFTL [53] is a learning-based FTL that learns data access patterns via linear regression to reduce the mapping table size. Our study is inspired by these studies in terms of predicting latency and pattern in SSD using a learning model. In contrast, we focus on predicting SSD failure points and quickly addressing the SSD failure.

SSD failure prediction models: WEFR [64] presents a failure prediction algorithm regardless of the SSD manufacturer or model to select failure-related information. Alter *et al.* [8] analyze and classify failure cases and propose a failure prediction model through machine learning. Zhang *et al.* [70] analyze the characteristics of failed and normal SSDs through data center operation logs and propose MVTRF to predict failure types, times, and status. Chakraborttii *et al.* [12, 13] present a learning model to classify the failure type via the log of SSDs and predict their failure. Hao *et al.* [19] propose RUS_Ensemble learning to increase the true positive rate compared with SSD failure prediction models based on SMART information.

Our study is in line with these studies [8, 12, 13, 19, 64, 70] in terms of utilizing machine learning techniques to mitigate the impact of SSD failures. Specifically, they aim to predict a potential failure that does not occur using offline learning. Meanwhile, we focus on detecting the failure that has already occurred to minimize the application data loss via online learning instead of offline learning. We believe that our scheme can incorporate these prediction strategies to be more effective against failures.

7 Conclusion

This paper aims to minimize application data loss in a storage system upon an SSD failure. To this end, we propose RL-Watchdog (`RLW`) which examines SSD liveness or failures quickly, precisely, and online. Specifically, `RLW` first periodically monitors failures in a lightweight manner. Second, `RLW` predicts the failure point more precisely regardless of the SSD models without offline pre-learning. Finally, `RLW` suspends the storage system immediately to prevent further data loss. We implement `RLW` in a Linux kernel and evaluate it in various configurations using a power supply board to inject a realistic power fault. The evaluation results indicate that `RLW` reduces the data loss by up to 96.7%, and its prediction accuracy reaches 99.8%.

Acknowledgements

The authors thank the anonymous reviewers for all valuable comments. This work was supported in part by the National Research Foundation of Korea (NRF) (No. NRF-2022R1A4A5034130) and Korea Institute for Advancement of Technology (KIAT) (No. KIAT-P0012724) grant funded by the Korea Government (Corresponding Author: Yongseok Son).

References

[1] Rocksdb, 2024. URL: http://rocksdb.org/.

[2] Samsung 980 SSD, 2024. URL: https://semiconductor.samsung.com/consumer-storage/internal-ssd/980/.

[3] Samsung PM9A3 SSD, 2024. URL: https://semiconductor.samsung.com/ssd/datacenter-ssd/pm9a3/.

[4] Technology power features, 2024. URL: https://nvmexpress.org/resource/technology-power-features/.

[5] Yahoo cloud servicing benchmark, 2024. URL: https://github.com/brianfrankcooper/YCSB.

[6] Saba Ahmadian, Farhad Taheri, and Hossein Asadi. Evaluating reliability of ssd-based i/o caches in enterprise storage systems. *IEEE Transactions on Emerging Topics in Computing*, 9(4):1914–1929, 2021. doi:10.1109/TETC.2019.2945087.

[7] Saba Ahmadian, Farhad Taheri, Mehrshad Lotfi, Maryam Karimi, and Hossein Asadi. Investigating power outage effects on reliability of solid-state drives. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 207–212, 2018. doi:10.23919/DATE.2018.8342004.

[8] Jacob Alter, Ji Xue, Alma Dimnaku, and Evgenia Smirni. Ssd failures in the field: Symptoms, causes, and prediction models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3295500.3356172.

[9] David G. Andersen and Steven Swanson. Rethinking flash in the data center. *IEEE Micro*, 30(4):52–54, jul 2010. doi:10.1109/MM.2010.71.

[10] Jens Axboe. Flexible I/O Tester, 2024. URL: https://github.com/axboe/fio.

[11] Keith Busch. Linux nvme driver, 2013. URL: https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2013/20130812_PreConfD_Busch.pdf.

[12] Chandranil Chakraborttii and Heiner Litz. Explaining ssd failures using anomaly detection. In *Non-Volatile Memory Workshop*, volume 1, page 1, 2020. URL: http://nvmw.ucsd.edu/nvmw2021-program/nvmw2021-data/nvmw2021-paper44-final_version_your_extended_abstract.pdf.

[13] Chandranil Chakraborttii and Heiner Litz. Improving the accuracy, adaptability, and interpretability of ssd failure prediction models. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 120–133, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3419111.3421300.

[14] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, jun 1994. doi:10.1145/176979.176981.

[15] Y. Chen. Chronos: Finding timeout bugs in practical distributed systems by deep-priority fuzzing with transient delay. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 112–112, Los Alamitos, CA, USA, may 2024. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00109, doi:10.1109/SP54263.2024.00109.

[16] Jesse Clifton and Eric Laber. Q-learning: Theory and applications. *Annual Review of Statistics and Its Application*, 7(1):279–301, 2020. arXiv:https://doi.org/10.1146/annurev-statistics-031219-041220, doi:10.1146/annurev-statistics-031219-041220.

[17] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. Eio: error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, USA, 2008. USENIX Association.

[18] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Deepthi Srinivasan, Biswaranjan Panda, Andrew Baptist, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Trans. Storage*, 14(3), oct 2018. doi:10.1145/3242086.

[19] Wenwen Hao, Ben Niu, Yin Luo, Kangkang Liu, and Na Liu. Improving accuracy and adaptability of ssd failure prediction in hyper-scale data centers. *SIGMETRICS Perform. Eval. Rev.*, 49(4):99–104, jun 2022. doi:10.1145/3543146.3543169.

[20] J.-W. Ho, M. Wright, and S. K. Das. Fast detection of replica node attacks in mobile sensor networks using sequential analysis. In *IEEE INFOCOM 2009*, pages

1773–1781, 2009. doi:10.1109/INFCOM.2009.5062097.

[21] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/osdi22/presentation/huang-lexiang.

[22] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating file system reliability on solid state drives. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 783–798, Renton, WA, July 2019. USENIX Association. URL: https://www.usenix.org/conference/atc19/presentation/jaffer.

[23] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. Fusionraid: Achieving consistent low latency for commodity ssd arrays. In *FAST*, pages 355–370, 2021. URL: https://www.usenix.org/system/files/fast21-jiang.pdf.

[24] Saurabh Kadekodi, Francisco Maturana, Sanjith Athlur, Arif Merchant, K. V. Rashmi, and Gregory R. Ganger. Tiger: Disk-Adaptive redundancy without placement restrictions. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 413–429, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/osdi22/presentation/kadekodi.

[25] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996. doi:10.1613/jair.301.

[26] Wonkyung Kang, Dongkun Shin, and Sungjoo Yoo. Reinforcement learning-assisted garbage collection to mitigate long-tail latency in ssd. *ACM Trans. Embed. Comput. Syst.*, 16(5s), sep 2017. doi:10.1145/3126537.

[27] Wonkyung Kang and Sungjoo Yoo. q -value prediction for reinforcement learning assisted garbage collection to reduce long tail latency in ssd. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2240–2253, 2020. doi:10.1109/TCAD.2019.2962781.

[28] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 201–217, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3373376.3378496.

[29] Joonsung Kim, Pyeongsu Park, Jaehyung Ahn, Jihun Kim, Jong Kim, and Jangwoo Kim. Ssdcheck: Timely and accurate prediction of irregular behaviors in black-box ssds. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 455–468, 2018. doi:10.1109/MICRO.2018.00044.

[30] Donghun Koo, Jaehwan Lee, Jialin Liu, Eun-Kyu Byun, Jae-Hyuck Kwak, Glenn K. Lockwood, Soonwook Hwang, Katie Antypas, Kesheng Wu, and Hyeonsang Eom. An empirical study of i/o separation for burst buffers in hpc systems. *Journal of Parallel and Distributed Computing*, 148:96–108, 2021. URL: https://www.sciencedirect.com/science/article/pii/S0743731520303907, doi:10.1016/j.jpdc.2020.10.007.

[31] Daniar H Kurniawan, Levent Toksoz, Anirudh Badam, Tim Emami, Sandeep Madireddy, Robert B Ross, Henry Hoffmann, and Haryadi S Gunawi. Ionet: Towards an open machine learning training ground for i/o performance prediction. *Technical Report2021*, 2021.

[32] Hyeon Gyu Lee, Juwon Lee, Minwook Kim, Donghwa Shin, Sungjin Lee, Bryan S. Kim, Eunji Lee, and Sang Lyul Min. Spartanssd: a reliable ssd under capacitance constraints. In *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, 2021. doi:10.1109/ISLPED52811.2021.9502476.

[33] Scott Lee, Eriksson Chuang, William Chang, Jerry Syue, and Cooper Li. Problem of the slot connector model extraction by de-embedding methodology. In *2020 International Symposium on Electromagnetic Compatibility - EMC EUROPE*, pages 1–4, 2020. doi:10.1109/EMCEUROPE48519.2020.9245687.

[34] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. Leapio: Efficient and portable virtual nvme storage on arm socs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 591–605, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3373376.3378531.

[35] Yuhua Lin and Haiying Shen. Eafr: An energy-efficient adaptive file replication system in data-intensive clusters. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):1017–1030, 2017. doi:10.1109/TPDS.2016.2613989.

[36] Kirill D. Liubavin, Dmitriy A. Furletov, Andrey V. Novikov, Konstantin S. Kurenkov, Vaagn A. Oganesyan, and Oleg A. Kalistratov. Design of a fully-autonomous low-power axi4 firewall for pci-express nvme ssd. In *2022 Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*, pages 166–169, 2022. doi:10.1109/ElConRus54750.2022.9755734.

[37] Ruiming Lu, Erci Xu, Yiming Zhang, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Minglu Li, and Jiesheng Wu. NVMe SSD failures in the field: the Fail-Stop and the Fail-Slow. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1005–1020, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/atc22/presentation/lu.

[38] Sixiang Ma and Yang Wang. Accurate timeout detection despite arbitrary processing delays. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 467–480, Boston, MA, July 2018. USENIX Association. URL: https://www.usenix.org/conference/atc18/presentation/ma-sixiang.

[39] Farzaneh Mahdisoltani, Ioan Stefanovici, and Bianca Schroeder. Proactive error prediction to improve storage system reliability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 391–402, Santa Clara, CA, July 2017. USENIX Association. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/mahdisoltani.

[40] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. A study of ssd reliability in large scale enterprise storage deployments. In *FAST*, pages 137–149, 2020. URL: https://www.usenix.org/system/files/fast20-maneas.pdf.

[41] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. Operational characteristics of SSDs in enterprise storage systems: A Large-Scale field study. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 165–180, Santa Clara, CA, February 2022. USENIX Association. URL: https://www.usenix.org/conference/fast22/presentation/maneas.

[42] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. Ssd failures in datacenters: What? when? and why? In *Proceedings of the 9th ACM International on Systems and Storage Conference*, SYSTOR '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2928275.2928278.

[43] Yuanjiang Ni, Ji Jiang, Dejun Jiang, Xiaosong Ma, Jin Xiong, and Yuangang Wang. S-rac: Ssd friendly caching for data center workloads. In *Proceedings of the 9th ACM International on Systems and Storage Conference*, SYSTOR '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2928275.2928284.

[44] N K Nivetha and D Vijayakumar. Modeling fuzzy based replication strategy to improve data availabiity in cloud datacenter. In *2016 International Conference on Computing Technologies and Intelligent Data Engineering (ICCTIDE'16)*, pages 1–6, 2016. doi:10.1109/ICCTIDE.2016.7725322.

[45] Gyuyoung Park and Myoungsoo Jung. Automatic-ssd: Full hardware automation over new memory for high performance and energy efficient pcie storage cards. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3400302.3415653.

[46] Gyuyoung Park and Myoungsoo Jung. Automatic-ssd: Full hardware automation over new memory for high performance and energy efficient pcie storage cards. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3400302.3415653.

[47] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, page 109–116, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/50202.50214.

[48] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 206–220, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1095810.1095830.

[49] Jose Santos. Flexible file system benchmark, 2024. URL: https://sourceforge.net/projects/ffsb/.

[50] Julian Schrittwieser, Thomas Hubert, Amol Mandhane, Mohammadamin Barekatain, Ioannis Antonoglou, and David Silver. Online and offline reinforcement learning by planning with a learned model. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 27580–27591. Curran Associates, Inc., 2021. URL: https://proceedings.neurips.cc/paper_files/paper/2021/file/e8258e5140317ff36c7f8225a3bf9590-Paper.pdf.

[51] Yiliang Shi, Danny V. Murillo, Simeng Wang, Jinrui Cao, and Mai Zheng. A command-level study of linux kernel bugs. In *2017 International Conference on Computing, Networking and Communications (ICNC)*, pages 798–802, 2017. doi:10.1109/ICCNC.2017.7876233.

[52] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, page 8, USA, 2010. USENIX Association.

[53] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. Leaftl: A learning-based flash translation layer for solid-state drives. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 442–456, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3575693.3575744.

[54] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[55] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016. URL: https://www.usenix.org/system/files/login/articles/login_spring16_02_tarasov.pdf.

[56] Mahdi Torabzadehkashi, Ali Heydarigorji, Siavash Rezaei, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. Accelerating hpc applications using computational storage devices. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1878–1885, 2019. doi:10.1109/HPCC/SmartCity/DSS.2019.00259.

[57] Linus Torvalds. Linux kernel, 2024. URL: https://github.com/torvalds/linux/blob/v6.0/drivers/nvme/host/pci.c.

[58] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark Silberstein. Slashing the disaggregation tax in heterogeneous data centers with fractos. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 352–367, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3492321.3519569.

[59] Yi Wang, Mingxu Zhang, Xuan Yang, and Tao Li. A thermal-aware physical space reallocation for open-channel ssd with 3-d flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):617–627, 2019. doi:10.1109/TCAD.2018.2821442.

[60] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992. doi:10.1007/BF00992698.

[61] Marco A Wiering and Martijn Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12(3):729, 2012.

[62] Chentao Wu and Xubin He. Gsr: A global stripe-based redistribution approach to accelerate raid-5 scaling. In *2012 41st International Conference on Parallel Processing*, pages 460–469, 2012. doi:10.1109/ICPP.2012.32.

[63] Chentao Wu, Xubin He, Guanying Wu, Shenggang Wan, Xiaohua Liu, Qiang Cao, and Changsheng Xie. Hdp code: A horizontal-diagonal parity code to optimize i/o load balancing in raid-6. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 209–220, 2011. doi:10.1109/DSN.2011.5958220.

[64] Fan Xu, Shujie Han, Patrick P. C. Lee, Yi Liu, Cheng He, and Jiongzhou Liu. General feature selection for failure prediction in large-scale ssd deployment. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 263–270, 2021. doi:10.1109/DSN48987.2021.00039.

[65] Ji Zhang, Ke Zhou, Ping Huang, Xubin He, Ming Xie, Bin Cheng, Yongguang Ji, and Yinhu Wang. Minority disk failure prediction based on transfer learning in large data centers of heterogeneous disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2155–2169, 2020. doi:10.1109/TPDS.2020.2985346.

[66] Jianquan Zhang, Dan Feng, Jianlin Gao, Wei Tong, Jingning Liu, Yu Hua, Yang Gao, Caihua Fang, Wen Xia, Feiling Fu, and Yaqing Li. Application-aware and

software-defined ssd scheme for tencent large-scale storage system. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 482–490, 2016. doi:10.1109/ICPADS.2016.0071.

[67] Jie Zhang, Mustafa Shihab, and Myoungsoo Jung. Power, energy, and thermal considerations in SSD-Based I/O acceleration. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, June 2014. USENIX Association. URL: https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/zhang.

[68] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. *SIGARCH Comput. Archit. News*, 43(1):3–18, mar 2015. doi:10.1145/2786763.2694370.

[69] Yuchao Zhang, Junchen Jiang, Ke Xu, Xiaohui Nie, Martin J. Reed, Haiyang Wang, Guang Yao, Miao Zhang, and Kai Chen. Bds: A centralized near-optimal overlay network for inter-datacenter data replication. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3190508.3190519.

[70] Yuqi Zhang, Wenwen Hao, Ben Niu, Kangkang Liu, Shuyang Wang, Na Liu, Xing He, Yongwong Gwon, and Chankyu Koh. Multi-view feature-based SSD failure prediction: What, when, and why. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 409–424, Santa Clara, CA, February 2023. USENIX Association. URL: https://www.usenix.org/conference/fast23/presentation/zhang.

[71] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the robustness of ssds under power fault. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 271–284, 2013. URL: https://www.usenix.org/system/files/conference/fast13/fast13-final80.pdf.

[72] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W. Zhao, and Elizabeth S. Yang. Reliability analysis of ssds under power fault. *ACM Trans. Comput. Syst.*, 34(4), nov 2016. doi:10.1145/2992782.

[73] Siyuan Zhou and Shuai Mu. Fault-Tolerant replication with Pull-Based consensus in MongoDB. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 687–703. USENIX Association, April 2021. URL: https://www.usenix.org/conference/nsdi21/presentation/zhou.

[74] You Zhou, Qiulin Wu, Fei Wu, Hong Jiang, Jian Zhou, and Changsheng Xie. Remap-ssd: Safely and efficiently exploiting ssd address remapping to eliminate duplicate writes. In *FAST*, pages 187–202, 2021. URL: https://www.usenix.org/system/files/fast21-zhou.pdf.