Telomere: Real-Time NAND Flash Storage

KATHERINE MISSIMER, MANOS ATHANASSOULIS, and RICHARD WEST,
Boston University

Modern solid-state disks achieve high data transfer rates due to their massive internal parallelism. However, out-of-place updates for flash memory incur garbage collection costs when valid data needs to be copied during space reclamation. The root cause of this extra cost is that solid-state disks are not always able to accurately determine data lifetime and group together data that expires before the space needs to be reclaimed. Real-time systems found in autonomous vehicles, industrial control systems, and assembly-line robots store data from hundreds of sensors and often have predictable data lifetimes. These systems require guaranteed high storage bandwidth for read and write operations by mission-critical real-time tasks. In this article, we depart from the traditional block device interface to guarantee the high throughput needed to process large volumes of data. Using data lifetime information from the application layer, our proposed real-time design, called *Telomere*, is able to intelligently lay out data in NAND flash memory and eliminate valid page copies during garbage collection. Telomere's real-time admission control is able to guarantee tasks their required read and write operations within their periods. Under randomly generated tasksets containing 500 tasks, Telomere achieves 30% higher throughput with a 5% storage cost compared to pre-existing techniques.

CCS Concepts: • **Computer systems organization → Real-time systems;**

Additional Key Words and Phrases: Real-time storage, SSD, flash translation layer

ACM Reference format:
Katherine Missimer, Manos Athanassoulis, and Richard West. 2022. Telomere: Real-Time NAND Flash Storage. *ACM Trans. Embedd. Comput. Syst.* 21, 1, Article 10 (January 2022), 24 pages.
https://doi.org/10.1145/3479157

1 INTRODUCTION

Autonomous vehicles, such as driverless cars and unmanned aerial vehicles, are becoming increasingly important in our society. Unmanned aerial vehicles are used to deliver equipment to medical facilities in times of crisis [40] and monitor public areas [5]. Autonomous vehicles rely on real-time processing of sensor data to perform collision avoidance, path planning, object detection, 3D scene reconstruction, simultaneous localization and mapping (SLAM), and other tasks. 3D cameras, laser range finders, radar, sonar and inertial sensors provide numerous data streams that need to

This material is based upon work supported by the National Science Foundation under grants 2007707 and 1850202. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.
Authors' address: K. Missimer, M. Athanassoulis, and R. West, Boston University, Boston, USA; emails: kzhao@cs.bu.edu, mathan@bu.edu, richwest@cs.bu.edu.

1539-9087/2022/01-ART10 $15.00
https://doi.org/10.1145/3479157

be stored and retrieved periodically with timing guarantees. Google's self-driving car has been reported to generate on the order of 1 GB/s of data from its various onboard sensors [3]. Tesla recently announced moving toward a "pure vision" approach, relying on eight cameras and 12 ultrasonic sensors for autopilot. To accomplish mission objectives, these vehicles need to store, retrieve, and process a vast amount of parallel data streams within specific time bounds. For example, Georgia Tech's Autonomous Mini Rally Car races around a track at 30 km/hour, calculating its trajectory 60 times every second [1]. This would require sensor data to be read in and processed every 16 msec. Assuming eight 4K 8-bit cameras streaming in 60 frames per second, we can calculate about 63 MB of camera data produced every 16 msec.

NAND flash memory has desirable characteristics for real-time information storage and retrieval, such as non-volatility, shock resistance, low power consumption, and fast access time [26, 37]. However, NAND flash memory management is complex because in-place updates are not possible. Once a memory location is written to, it must be *erased* before it can be written to again. Traditional **solid-state disks (SSDs)** designed for general or high-performance computing suffer from unpredictability such as high tail latency and are unable to meet real-time deadlines [22]. A large body of literature exists that provides throughput guarantees [14, 16, 38, 44]. However, related work that provides the strict timing guarantees of a real-time system is scarce despite the increasing demand for real-time sensor data storage. NAND flash memory also has a limited lifetime [29], as repeated writes and erasures of the same physical block *wears* it, rendering it eventually not usable. Thus, strategic placement of data in flash is important to reduce **garbage collection (GC)** overhead and increase device lifetime. Tesla recently announced failure in the eMMC flash memory card in their vehicles due to logging, which caused the flash memory to wear out too quickly [36]. Inefficient data storage in flash memory leads to poor performance, long latency, and short device lifetime [15, 17, 20, 22, 46].

GC incurs substantial overhead in flash memory when data with different expiration times are stored together in the same flash block. When a block needs to be reclaimed, the valid pages in that block have to be copied to another block. This is why traditional **flash translation layers (FTLs)** often have a tunable trade-off between storage and throughput utilization, as seen in Figure 1. Storage utilization here refers to how much of the available capacity is exposed as usable capacity to the user (the more capacity exposed to the user the higher the utilization), and bandwidth (or throughput) utilization refers to how close to the maximum device bandwidth a device can operate. Traditional flash designs copy valid pages during GC resulting in a write amplification factor (*WAF*) greater than 1, which directly affects throughput utilization. *WAF* is the ratio of the number of page writes on flash to the number of page write requests from the device driver. The higher the *WAF* value, the lower the throughput utilization. The *WAF* value can be decreased by increasing *over-provisioning*, or the amount of physical storage space that, instead of being available to the user, is used to facilitate CG. Higher over-provisioning will decrease *WAF* at the expense of lower storage utilization. If the SSD is able to figure out how to store data such that all the pages in a block become invalid before it needs to be reclaimed, CG overhead would only be a block erasure, and it will not require any over-provisioning. This is the approach taken by multi-stream SSDs [17, 23, 45], where data with similar lifetime is assigned to a stream, and the FTL groups in the same physical blocks data from a single stream.

State-of-the-art SSD devices support a block I/O interface and do not have a way to receive application-specific knowledge needed to make the most informed decisions for data layout. Instead, they must infer data hotness, which often cannot be accurately predicted. We call this class *drive-managed* devices. A new class of devices expose a limited percentage of the internals of the SSD to the host-level software to enable fine-grained control over device operations from the host. We call this class *host-managed* devices. A host-managed approach, however, cannot provide a

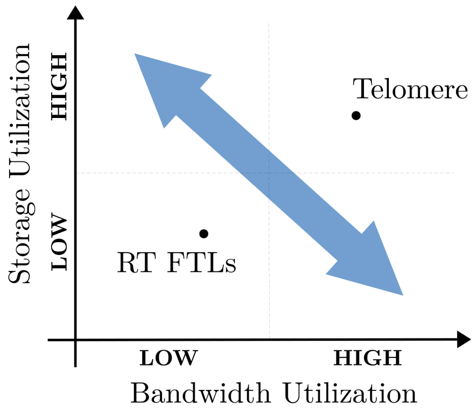

Fig. 1. In traditional NAND flash devices, there is a tunable trade-off between storage and throughput utilization. Real-time FTL (RT FTL) designs typically use redundancy to provide timing guarantees, thus suffering from lower storage and throughput utilization [28, 34, 38].

real-time solution when wear-leveling is handled in the device. Our design bridges this gap where the application provides knowledge of the data lifetime and the throughput requirements and the drive guarantees predictable high throughput and storage utilization.

The problem: Current real-time approaches under-utilize throughput and storage. Real-time FTLs typically use redundancy to provide timing guarantees [28, 34, 38]. Writing parity data results in lower throughput and storage utilization. Another technique often used in real-time FTL designs to reduce latency is partial GC [34, 47]. However, these designs do not exploit the parallelism in an SSD, thus resulting in low throughput.

Our approach: Telomere. Based on the preceding information, this article presents the design of a drive-managed SSD system called *Telomere*, which allows data lifetime information to be passed from the application to the device. This is beneficial to real-time systems featuring numerous high bandwidth sensors (e.g., cameras) that must store, retrieve, and process data according to throughput requirements, and which must replace stale data once it has expired. Telomere carefully selects which data page to collocate in order to minimize GC overhead. Our throughput admission control guarantees that tasks meet their deadlines and admits task sets with high read and write throughput.

To capture real-time applications, we assume a real-time model where each application defines a set of periodic tasks with a pre-determined data lifetime per task. For each task, these parameters are passed to the **operating system (OS)** through the *open*() system call, then admission control tests determine whether the task is accepted. If the system cannot guarantee the throughput and lifetime requirements of a task, the *open*() syscall fails for that particular task.

Contributions. Our work offers the following contributions:

- A new interface that provides the drive with information about the lifetime of the data.
- Two block allocation algorithms for partitioning blocks among different tasks:
  - *Single Task Placement (SiP)* that allocates each task to different blocks, and
  - *Shared Task Placement (SharP)* that identifies which tasks exhibit similar data lifetime to collocate blocks with similar access patterns.
- A throughput-aware admission control to guarantee performance.

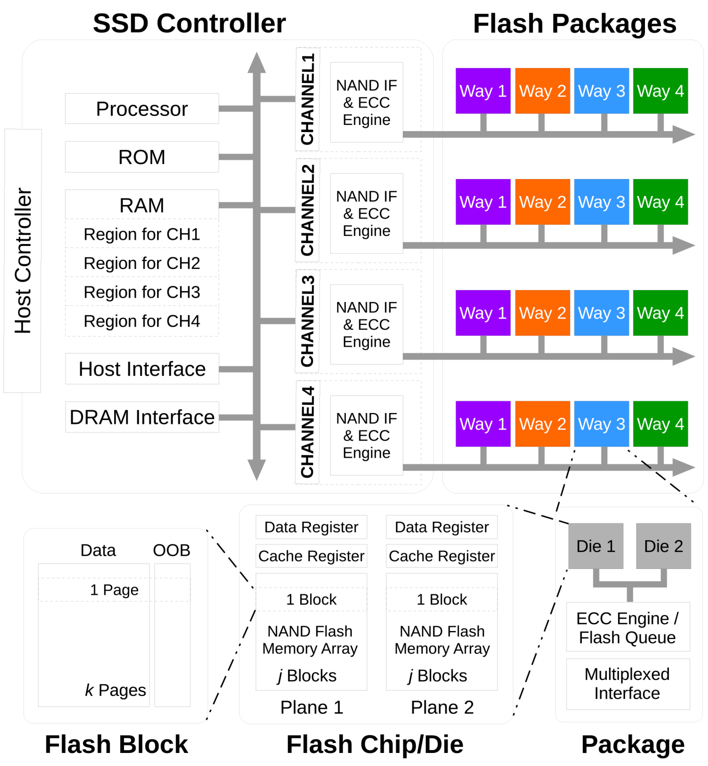

Fig. 2. SSD internal architecture.

- We both simulate and experiment on real hardware using an OpenSSD Cosmos board that allows us to treat an SSD as a white box.

2 BACKGROUND

This section explains the internals of NAND flash memory and describes flash properties such as GC, over-provisioning, and granularity to improve read and write throughput.

2.1 NAND Flash Memory

The internal structure of flash storage is significantly different from that of traditional mechanical hard drives. As shown in Figure 2, the smallest read and write unit in flash devices is a page. A page used to be standardized at 512 and 2,048 bytes [25]. However, recently much larger page sizes have been seen ranging from 4 to 32 KB [27]. In addition to data, a page also contains some extra bytes for an out of band (OOB) area, which is used to store bookkeeping information (e.g., **error correction code (ECC)**) for the corresponding page. Data in NAND flash memory cannot be overwritten; instead, a *block* of pages must first be erased before a *page* is eligible for reuse. When SSD initially became available, a flash block contained 32 or 64 pages [25]. Current flash blocks in SSDs can range from 128 to 512 pages [27]. A 4-MB block, for example, can contain 512 pages with each page containing 8 KB. Multiple blocks form a plane, and typically two to four planes form a die. The flash die, also known as a chip, is the smallest unit that can independently execute commands or report status. Typically, up to 16 flash chips form a flash package.

A flash package exists on a specific *way* on a specific *channel*. There are usually four to eight ways on a channel. The ways, also called *banks*, on a channel share a common flash bus and an internal data bus. The way arbiter in the channel controller grants access to the shared buses. This is called *way interleaving* or *package-level parallelism*. There are usually 4 to 8 channels in a consumer SSD [21, 30], although high-end enterprise SSDs such as Flashtec 3016 and 3032 have controllers with 16 and 32 channels, respectively. Each channel contains its own NAND interface block and ECC block, so it can operate independently. This is called *channel striping* or *channel-level parallelism*. Way interleaving and channel striping are the two main methods of parallelization that modern flash controllers support [10].

2.2 Garbage Collection

In a page-level mapping FTL, every logical page number is mapped to a physical page number. When a logical page gets updated, the physical page containing the old data gets marked as invalid and the new, updated content is written to a different physical page. When GC is triggered, a block is selected to be erased. All valid pages in that block are copied to a clean block, and the mappings are updated.

The need to reclaim space in NAND flash memory results in potentially unacceptable worst-case performance for a real-time system. When free space becomes limited, GC selects a block to reclaim. The valid pages in the selected block are copied to another block, and the selected block is erased. In the worst case, only one invalid page out of P pages in a block is reclaimed. Therefore, if a write request triggers GC, it could be blocked waiting for one block erasure and $P - 1$ read and write operations to copy the valid pages.

To mitigate GC costs and efficiently perform wear-leveling, extra physical blocks are reserved to migrate valid data from blocks that are being reclaimed. These extra blocks make up the over-provisioning space in the SSD. With over-provisioning, the logical address space becomes a fraction of the physical address space in the SSD. Let λ be the device utilization:

$$\lambda = \frac{\text{logical address space}}{\text{physical address space}}. \tag{1}$$

Another frequently used metric for measuring the extra FTL capacity is defined as the ratio between the extra capacity and the user capacity, denoted as α:

$$\alpha = \frac{\text{extra capacity}}{\text{user capacity}} = \frac{1}{\lambda} - 1. \tag{2}$$

2.3 Access Granularity

When the read and write granularity of a request is more than one flash page, the FTL could stripe the request across different parallel units to increase read and write throughput. Although write requests with granularity of one page could be buffered and written to flash memory in parallel, the worst-case scenario for read requests is that all the pages that need to be read exist on a single flash die and cannot be performed in parallel. Therefore, the read bandwidth can be achieved when the read and write granularity equals the number of parallel units in the SSD.

2.4 Drive-Managed vs. Host-Managed Design

To support the block I/O interface, traditional *drive-managed* SSDs include an FTL, which performs address mapping, GC, wear-leveling, and error correction [24]. However, these SSDs suffer from shortcomings such as log-on-log [46], high tail- [15], and unpredictable I/O latency [2, 11].

Figure 3(a) shows the FTL for a drive-managed design. The I/O interface does not provide the drive with any knowledge of the lifetime of the data being stored. Without this application-specific

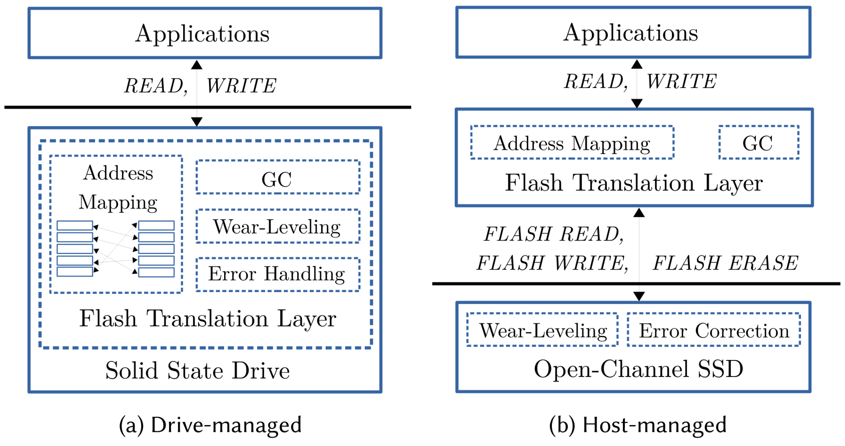

Fig. 3. In drive-managed designs, the SSD includes a layer of indirection (the FTL). In host-managed designs, the mapping and GC is managed by a software FTL layer in the OS or in the application layer.

knowledge, drive-managed SSDs can only infer update frequencies to try to store hot (recently updated) and cold data in different blocks. Storing hot and cold data together increases GC overhead. This is caused by writing hot data to a new block, invalidating old pages of the updated data in a victim block, copying valid pages for cold data in the victim block to the new block, and then erasing the victim block. The copying of the valid data would be eliminated if it were not located in the same block as hot data. Prior work [17, 23, 45] shows that writing data with similar hotness or lifetime to the same flash blocks results in improved performance.

To overcome the shortcomings of drive-managed SSDs, the Open-Channel SSD community has been pushing for *host-managed* devices [8]. For example, approaches such as LightNVM [9] and Zoned Namespaces [7, 42] expose the SSD internals to host-level software, which is able to control data placement and I/O scheduling. Figure 3(b) shows a host-managed design, where the mapping and GC is handled by a software FTL in the OS, or in the application itself.

Currently, the Open-Channel SSD community is developing a standard as part of the NVMe 2.0 specification, to allow an SSD to expose a logical address namespace using zones [7]. In this new interface, applications can intelligently decide where to place the data in zones based on their knowledge of data hotness. Samsung showed that by passing information from the application down to the drive to specify data hotness, they were able to improve the update throughput by 56% [17]. The increase in throughput is attributed to lower valid page copies and GC overheads. However, with host-managed devices, wear-leveling [29, 32] is still implemented in the device, where it is able to track bad blocks and perform error correction. This leads to movement of data and erasure of blocks outside the control of the software FTL or the application layer. In turn, this adds timing unpredictability to the performance of applications, which is undesirable in a real-time system.

3 TELOMERE DESIGN

Our proposed design, called *Telomere*, is a drive-managed design, where data lifetime is passed down to the FTL, enabling it to efficiently store data such that GC overhead is only a block erasure. It is a NAND flash storage system targeted for sensor data in real-time systems. As shown in Figure 4, Telomere stripes data across parallel units in the SSD and eliminates write amplification

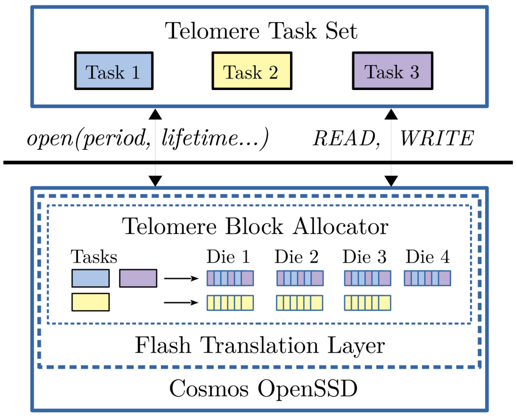

Fig. 4. A Telomere-compliant application defines a set of periodic tasks with a pre-determined data lifetime per task. These parameters are passed to the OS on the *open*() syscall, and admission control tests determine if the task is accepted. Then, Telomere places data with similar expiration times together in the same block so that GC overhead is minimized.

by using a block expiration time.[1] Data has a lifetime value and each block is associated with a future timestamp at which point all the data inside that block will become invalid. *Telomere places data with similar expiration times together in the same block so that GC overhead is minimized.* It eliminates the need to copy valid pages to a new block to reclaim the invalid pages in that block. Instead, pages with similar expiration times and update frequencies are stored together and the block is reclaimed when all the pages become invalid. Thus, GC in Telomere is *simply a block erasure.*

3.1 Task Model

Let $\{\tau_1, \tau_2, \ldots, \tau_n\}$ be a set of n periodic tasks on the FTL performing read and write requests. Each task τ_i is associated with a real-time CPU task δ_i with a capacity and a period. On *open*(), the user specifies a task with a read period T_i^r, a write period T_i^w, the maximum number of flash pages read r_i and written w_i during their respective periods, and a data lifetime l_i^w. Table 1 contains all the symbol definitions. The read and write periods are multiples of the period of δ_i, the associated real-time CPU task. The data lifetime is the number of write periods after the data is written during which the data is valid. After l_i^w periods, the data expires and is no longer accessible. A periodic task releases jobs at regular intervals based on its period. Each job has a request time and a deadline. We assume the deadline equals the period. Note that a task in our model corresponds to opening a file. A CPU task can open multiple files, which would result in multiple tasks in our model. Our task model is similar to the one used in Partitioned Real-Time FTL [28].

During the *open*() syscall, the admission control executes to ensure there are enough resources for the new task τ_i to run. The admission control consists of two parts. First, we present the storage admission control to guarantee the availability of free pages to write the data from task τ_i. Second, we provide the throughput admission control to guarantee the read and write throughput requirement of task τ_i.

[1]The name *Telomere* is inspired by the biological structures that cap the ends of chromosomes that are truncated during cell division. Over time, the telomere ends become shorter; when they get too short, the cell can no longer divide and "expires" or dies.

Table 1. Telomere Symbol Definitions

Symbol	Definition
λ	Device utilization: logical over physical address space
α	Over-provisioning: extra capacity over user capacity
τ_i	A periodic task
T_i^r	Read period for τ_i
r_i	Number of pages read in one read period
T_i^w	Write period for τ_i
w_i	Number of pages written in one write period
l_i^w	Number of write periods before the data expires
g	Read and write page granularity
P	Number of pages in a flash block
S_{pages}	Number of pages in the SSD
S_{blocks}	Number of blocks in the SSD
S_{chips}	Number of flash chips in the SSD
t_r	Flash page read latency
t_w	Flash page write latency
t_e	Flash block erasure latency

4 STORAGE ADMISSION CONTROL

To guarantee that there are enough free pages available to write new data, we need to bound the number of blocks used. To simplify the problem, we assume that the number of free pages needed by a task is available at the beginning of each period. In addition, to ensure that the data can be read regardless of when the CPU task is scheduled, we add an extra period to the expiration time. For a task τ_i with write period T_i^w, we can calculate the latest write request start time s and the time those pages will expire e at an time t as follows:

$$s = \lfloor t/T_i^w \rfloor \cdot T_i^w, \tag{3}$$

$$e = s + l_i^w \cdot T_i^w + T_i^w. \tag{4}$$

We present two block allocation algorithms to bound the number of blocks needed for a task set. The Single Task Placement (SiP) allocates different blocks to each task. The Shared Task Placement (SharP) groups together tasks with similar data lifetime and shares blocks among the group of tasks.

4.1 Single Task Placement (SiP)

SiP allocates independent blocks to store the incoming write jobs from each task. For example, assume that each block contains eight pages. Let task τ_1 request one page write every two time units that expires after three periods, and let task τ_2 request two pages of data written every three time units that expire after two periods.

To calculate the number of blocks to allocate to a task, we need to compute how much valid data the task stores and how many extra pages are stored before the block containing the oldest data expires. A block expires when all of its pages become invalid.

For example, in Figure 5, the maximum number of valid pages stored by task τ_1 at any point in time is four pages. Let w_i be the number of pages written in one period, and l_i^w be the number of periods before the data expires. $K(\tau_i)$ is the maximum number of valid pages stored by τ_i defined

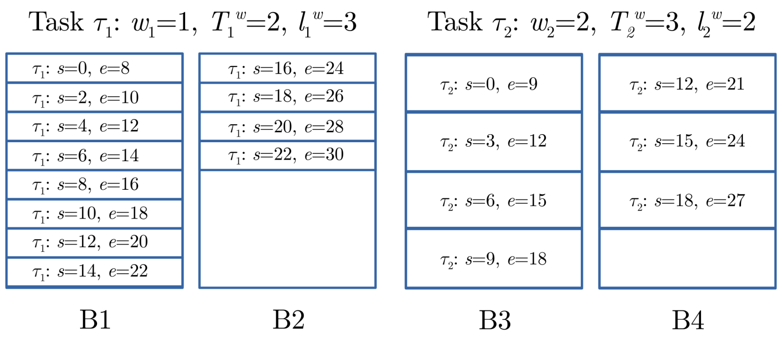

Fig. 5. Telomere SiP: a block only contains data written by one task. τ_1 writes data to blocks $B1$ and $B2$. τ_2 writes data to blocks $B3$ and $B4$. No block contains data from both tasks.

as follows:

$$K(\tau_i) = w_i \cdot (l_i^w + 1). \tag{5}$$

A block is alive if it contains valid data. For the block storing the oldest data to expire, τ_1 will write at most another block of data. For example, in Figure 5, task τ_1 first writes four valid pages, but for block $B1$ to expire, it will write at most another block, or eight pages, of data. At $t = 22$, all the data in $B1$ are expired, so $B1$ can be erased. The maximum number of pages that will be written during the time of a block erasure is $(w_i \cdot \lceil t_e/T_i^w \rceil)$, where t_e is the time it takes to perform a block erasure. Let P be the number of pages in a block. The total number of pages needed by a task τ_i under SiP is therefore $(K(\tau_i) + P + w_i \cdot \lceil t_e/T_i^w \rceil)$. Since a block is allocated to a task and is not shared among tasks, SiP takes the ceiling of the total number of pages over P. We assume in the following equation that the number of pages written in one period is less than the size of a flash block ($w_i < P$). When $w_i \geq P$, the pages in blocks that store data from a single request will either be all valid or all invalid. This is the trivial case, and those blocks are simply added to the total number of blocks. $H_1(\tau_i)$ is the number of blocks needed by task τ_i with read and write granularity $g = 1$. It is defined as follows:

$$\begin{aligned} H_1(\tau_i) &= \left\lceil \frac{K(\tau_i) + P + w_i \cdot \lceil t_e/T_i^w \rceil}{P} \right\rceil \\ &= \left\lceil \frac{K(\tau_i) + w_i \cdot \lceil t_e/T_i^w \rceil}{P} \right\rceil + 1. \end{aligned} \tag{6}$$

Equation (6) assumes a read and write request size of one page. Oftentimes, requests of multiple pages are striped across flash chips to improve throughput. We extend the calculation to a granularity size of g pages. Similar to our previous assumption when $g = 1$, we assume in the following equation that the number of pages written in one period is less than the granularity multiplied by the size of a flash block ($w_i < g{\cdot}P$). $H_g(\tau_i)$ is the number of blocks needed by task τ_i defined as follows:

$$H_g(\tau_i) = g \cdot \left(\left\lceil \frac{K(\tau_i) + w_i \cdot \lceil t_e/T_i^w \rceil}{g{\cdot}P} \right\rceil + 1 \right). \tag{7}$$

Without the preceding assumption on w_i, K, K' and H_g are defined as follows:

$$K(\tau_i) = [w_i \% (g{\cdot}P)] \cdot (l_i^w + 1), \tag{8}$$

$$K'(\tau_i) = g \cdot \left\lfloor \frac{w_i}{g{\cdot}P} \right\rfloor \cdot (l_i^w + 1), \tag{9}$$

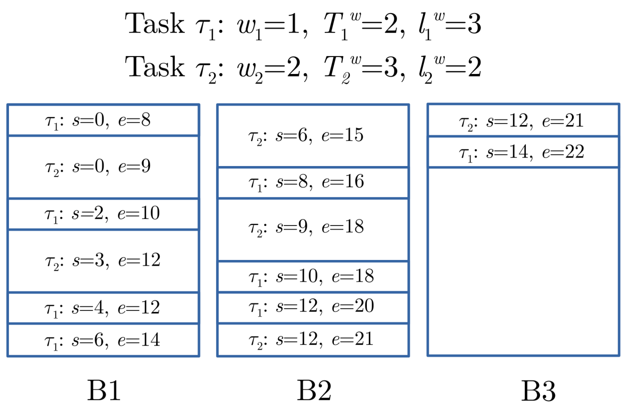

Fig. 6. Telomere SharP: a block contains data written by multiple tasks. Blocks $B1$, $B2$, and $B3$ are shared among tasks τ_1 and τ_2.

$$H_g(\tau_i) = g \cdot \left(\left\lceil \frac{K(\tau_i) + w_i \cdot \lceil t_e/T_i^w \rceil}{g{\cdot}P} \right\rceil + 1 \right) + K'(\tau_i). \tag{10}$$

The total number of blocks needed for a task set, D, is the sum of the blocks needed for each task. Let S_{blocks} be the total number of physical blocks. The storage admission control guarantees that the SSD logical space ($S_{blocks} \cdot \lambda$) can store data from all the tasks.

$$D = \sum_{i=0}^{m} H_g(\tau_i) \leq S_{blocks} \cdot \lambda \tag{11}$$

The benefit of SiP is the simplicity in its implementation, and, as we show in Section 6, its storage cost is comparable to the more complicated SharP algorithm when the size of the task set is small.

4.2 Shared Task Placement (SharP)

SharP uses the observation that since a block is erased when all the data pages expire, data with similar start and expiration times should be stored together. However, in the case that mixing has no benefit, then SharP falls back to a SiP-like partitioning that each task would use different blocks to store data. Figure 6 is an example of when it is more efficient to share blocks among the write requests of tasks τ_1 and τ_2 compared to allocating independent blocks for each task. When a block stores write requests from both tasks, only three blocks are needed compared to four blocks needed in SiP.

To determine the number of blocks to allocate to a set of tasks $\tau_i, \ldots, \tau_j$, SharP first computes the percentage of a block that will be written to by each task. For example, in Figure 6, task τ_1 writes to three to four pages of a block and task τ_2 will write to four to five pages of a block. SharP determines how much of a block each task writes to based on the task's throughput requirement ($\frac{w_i}{T_i^w}$) compared to the total write throughput requirement of the set of tasks. $Q(\tau_i, \{\tau_h \ldots \tau_k\})$ is the fraction of write throughput requirement of τ_i over the write throughput requirement of the set of tasks $\{\tau_h \ldots \tau_k\}$. It is defined as follows:

$$Q(\tau_i, \{\tau_h \ldots \tau_k\}) = \frac{w_i/T_i^w}{\sum_{\tau_j \in \{\tau_h \ldots \tau_k\}} (w_j/T_j^w)}. \tag{12}$$

Since each task can no longer write to all the pages in a block, the denominator in Equation (7) is multiplied by $Q(\tau_i, \{\tau_h \dots \tau_k\})$. Again, we assume that $w_i < g{\cdot}P$. $J(\tau_j, \{\tau_i \dots \tau_k\})$ is number of blocks needed by task τ_j while sharing blocks with other tasks in the set. It is defined as follows:

$$J(\tau_j, \{\tau_i \dots \tau_k\}) = g \cdot \left(\left\lceil \frac{K(\tau_j) + w_j \cdot \lceil t_e / T_j^w \rceil}{g{\cdot}P{\cdot}Q(\tau_j, \{\tau_i \dots \tau_k\})} \right\rceil + 1 \right). \tag{13}$$

The total number of blocks needed by the set of tasks is the maximum $J(\tau_j, \{\tau_i \dots \tau_k\})$ of all the tasks.

$$H_g(\{\tau_i \dots \tau_k\}) = max_{\forall \tau_j} J(\tau_j, \{\tau_i \dots \tau_k\})^{2} \tag{14}$$

Again, without the assumption on w_i, the equations are as follows:

$$Q(\tau_i, \{\tau_h \dots \tau_k\}) = \frac{(w_i \% (g{\cdot}P))/T_i}{\sum_{\tau_j \in \{\tau_h \dots \tau_k\}} (w_j \% (g{\cdot}P))/T_j}, \tag{15}$$

$$J(\tau_j, \{\tau_i \dots \tau_k\}) = g \cdot \left(\left\lceil \frac{K(\tau_j) + w_j \cdot \lceil t_e / T_j^w \rceil}{g{\cdot}P{\cdot}Q(\tau_j, \{\tau_i \dots \tau_k\}, g)} \right\rceil + 1 \right) + K'(\tau_i). \tag{16}$$

SharP partitions the tasks in a task set such that each partition will use separate blocks to store data. First, the tasks are sorted by the amount of time the data is alive, $(T_i^w \cdot (l_i^w + 1))$. The algorithm keeps track of a current partition and determines whether each task should be added to the current partition. The pseudocode is shown in Algorithm 1. For each task τ_1, the algorithm calculates the number of blocks needed if τ_1 is added to the current partition ($opt1$) and if τ_1 is excluded from the current partition—that is, whether τ_1 should be added to the next partition with the next task τ_2 ($opt2$) or if τ_1 should be its own partition and not be grouped with any other tasks ($opt3$).

The following example highlights why we need to consider task τ_2 in Algorithm 1. Assume there are eight pages in a block, time to erase a block is 2.7 msec, $g = 1$, and the current partition is $[\tau_0]$, where $\tau_0 = (w_0 = 4, T_0^w = 9, l_0^w = 9)$. Let $\tau_1 = (w_1 = 3, T_1^w = 14, l_1^w = 7)$ and $\tau_2 = (w_2 = 7, T_2^w = 16, l_2^w = 7)$, where the period is in milliseconds. To determine whether or not τ_1 should be added to the current partition, we calculate $opt1 = H([\tau_0, \tau_1]) + H(\tau_2) = 20$, $opt2 = H([\tau_0]) + H([\tau_1, \tau_2]) = 19$, and $opt3 = H([\tau_0]) + H(\tau_1) + H(\tau_2) = 21$. Here, we find that $opt2$ provides the partition with the smallest number of blocks, so τ_1 is not added to τ_0. If we did not consider task τ_2 in Algorithm 1, we would have only calculated $H([\tau_0, \tau_1]) = 11$ and $H([\tau_0]) + H(\tau_1) = 12$, and ended up adding τ_1 to the partition containing τ_0.

5 THROUGHPUT ADMISSION CONTROL

In addition to the storage admission control, we also need to guarantee the read and write throughput requirement of the tasks. Due to out-of-place updates in NAND flash, the read and write throughput can fluctuate due to interfering GC activities.

A read request with granularity g can be performed in parallel when $g > 1$. The read capacity C_i^r for task τ_i is as follows:

$$C_i^r = \left\lceil \frac{r_i}{g} \right\rceil {\cdot} t_r, \tag{17}$$

where t_r is the time it takes to read a flash page.

[2]Note that $H_g(\{\tau_i\})$ of a list consisting of a single task is equivalent to $H_g(\tau_i)$ in Equation (7).

ALGORITHM 1: Telomere SharP partitions tasks such that each partition will use separate blocks to store data.

```
procedure TelomereSharP($tasks$)
    Sort $tasks$ by increasing $(T^w \cdot (l^w + 1))$
    $idx = 0$
    $i = 0$
    $tlen = tasks.length$
    while $i < tlen - 1$ do
        $\tau_1 = tasks[i]$
        $\tau_2 = tasks[i + 1]$
        $opt1 = H(tasks[idx : i + 1]) + H(\tau_2)$
        $opt2 = H(tasks[idx : i]) + H([\tau_1, \tau_2])$
        $opt3 = H(tasks[idx : i]) + H(\tau_1) + H(\tau_2)$
        $minBlks = min(opt1, opt2, opt3)$
        if $opt1 == minBlks$ then
            $i = i + 1$
        else
            Add $tasks[idx : i]$ to partitions
            if $opt2 == minBlks$ then
                $idx = i$
                $i = i + 1$
            else
                Add $[\tau_1]$ to partitions
                $idx = i + 1$
                $i = i + 2$
            end if
        end if
    end while
    $opt1 = H(tasks[idx : tlen - 1])$
    $opt2 = H(tasks[idx : tlen - 1]) + H(tasks[tlen - 1])$
    if $opt1 < opt2$ then
        Add $tasks[idx : tlen]$ to partitions
    else
        Add $tasks[idx : tlen - 1]$ to partitions
        Add $[tasks[tlen - 1]]$ to partitions
    end if
end procedure
```

A write request is first written to a buffer and later written to flash. Since there are no in-place updates in flash memory, a write request consisting of multiple pages can be distributed to different flash chips. The write capacity C_i^w for task τ_i is as follows:

$$C_i^w = \left\lceil \frac{w_i}{S_{chips}} \right\rceil \cdot t_w, \tag{18}$$

where t_w is the time it takes to write a flash page.

Every write task has a corresponding GC task:

$$C_i^g = \left\lceil \frac{t_e}{S_{chips}} \right\rceil$$
$$T_i^g = \begin{cases} T_i^w / \left\lceil \frac{w_i}{P} \right\rceil, & \text{if } w_i > P \\ T_i^w \cdot \left\lfloor \frac{P}{w_i} \right\rfloor, & \text{otherwise} \end{cases} \tag{19}$$

where t_e is the latency for block erasure.

A schedulability test is invoked to ensure that all the read and write requests are schedulable. We use **Earliest Deadline First (EDF)**. The largest non-preemptive period is the longest flash operation that takes place on write flash chips, which is a block erasure. The feasibility of the real-time write requests can be verified by the following equation derived from Theorem 2 in Baker's Stack Resource Policy work [4], which presents a sufficient condition of the schedulability of tasks that have non-preemptible portions under EDF:

$$\frac{t_e}{min(T)} + \sum_{i=1}^{n} \left(\frac{C_i^r}{T_i^r} + \frac{C_i^w}{T_i^w} + \frac{C_i^g}{T_i^g} \right) \leq 1, \tag{20}$$

where $min(T)$ is the minimum period in all T_i^r, T_i^w, and T_i^g.

6 EVALUATION

The experimental evaluation consists of three sections: simulation-based schedulability tests, event-driven simulation to measure GC overheads, and hardware experiments on the OpenSSD Cosmos board. The admission control simulation shows that Telomere has a higher feasible throughput utilization with an additional storage cost. The event-driven simulation presents Telomere's low GC overhead. Hardware experiments with the OpenSSD Cosmos board show that Telomere is able to maintain high and predictable throughput.

6.1 Admission Control Simulations

6.1.1 SSD Parameters. We assume the following parameters for an SSD with 1 TB of storage. There are 16 flash channels, each containing 4 flash chips. There are 4,096 blocks per flash chip and 128 pages in a block, and each page is 32 KB. The average page read time is 0.2 msec, the average page write time is 0.7 msec, and the average block erasure time is 2.7 msec. The read bandwidth is 100 MB/s, the write bandwidth is 41.4 MB/s, and the block erasure bandwidth is 1.4 GB/s [16].

6.1.2 Data Generation Parameters. Fifty random task sets were generated with two varying total utilizations, U^s for storage and U^b for bandwidth, using the UUniFast algorithm [6] with values ranging from 0.01 to 1.0 in 0.01 increments. Each task set contains 500 tasks. Each task τ_i has a storage utilization U_i^s and a bandwidth utilization U_i^b such that $(\sum_{\forall i} U_i^s = U^s)$ and $(\sum_{\forall i} U_i^b = U^b)$. The read period T_i^r and write period T_i^w for each task is calculated based on the number of pages read and written per period. Data is striped across all flash chips, so the read and write granularity is 64. Let B_{max}^R be the read bandwidth, and let B_{max}^W be the write bandwidth. We generate reads and writes up to half of their respective bandwidths.

$$T_i^r = \left\lfloor \frac{r_i}{U_i^b \cdot (B_{max}^R/2)} \right\rfloor \tag{21}$$
$$T_i^w = \left\lfloor \frac{w_i}{U_i^b \cdot (B_{max}^W/2)} \right\rfloor \tag{22}$$

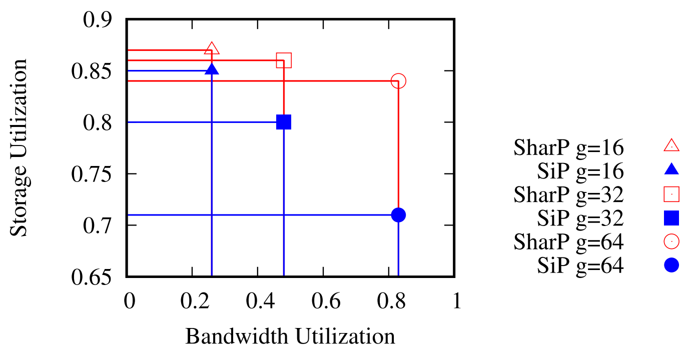

Fig. 7. Storage and throughput utilization at which 100% of the task sets are schedulable with different read and write granularity. As granularity increases, both throughput utilization and storage cost increases. The storage cost of Telomere SharP grows slower than Telomere SiP.

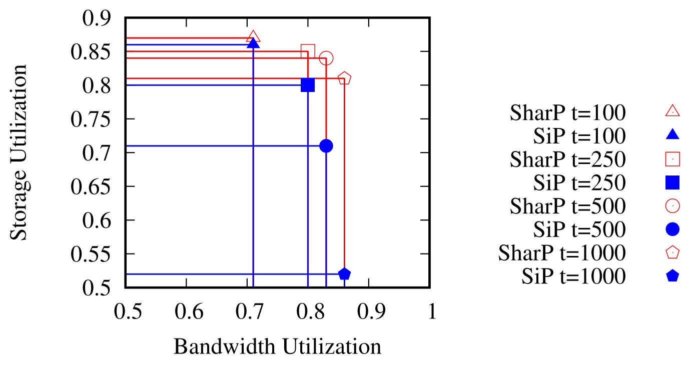

Fig. 8. Storage and throughput utilization at which 100% of the task sets are schedulable with varying number of tasks in a task set. The number of tasks in a task set significantly affects the storage cost of Telomere SiP since each task is assigned independent blocks.

The lifetime of the data depends on the storage utilization of the task and the number of pages the task writes to guarantee that the SSD has the capacity to store all the valid data.

$$l_i^w = \left\lfloor \frac{U_i^s \cdot S_{pages}}{w_i} \right\rfloor \tag{23}$$

6.1.3 Telomere SiP and SharP. The storage cost of Telomere SiP and SharP depends on the read and write granularity and the number of tasks in a task set. In Figures 7 and 8, we show the maximum storage utilization out of a 1-TB drive with device utilization $\lambda = 0.90$ as we vary these parameters.

Figure 7 shows how the read and write granularity affects the storage cost and throughput of Telomere SiP and SharP. When granularity increases, more pages are striped across flash chips. This means that multiple pages can be read and written in parallel, which leads to increased throughput. The striping of data also results in increased storage cost for Telomere SiP and SharP. Figure 7 shows the maximum storage and throughput utilization at which 100% of the task sets are schedulable. We do not plot the gradient from 100% to 0% schedulable because it is very small (within 1% to 2% storage or throughput utilization). Note that in our simulation, there are 64 flash chips, so there will not be any throughput or storage cost increase for granularity $g > 64$.

Figure 8 shows the storage cost of Telomere SiP and SharP as the number of tasks in each task set varies. We plot the storage and throughput utilization at which 100% of the task sets are schedulable for task sets with 100, 250, 500, and 1,000 tasks. As expected, the storage cost increases as the number of tasks increases. However, the storage cost of Telomere SharP grows at a much slower rate than that of Telomere SiP. The slight decrease in throughput utilization when the size of the task set decreases is due to the shorter periods in the generated tasks, which results in a larger value for the first term ($\frac{t_e}{min(T)}$) in Equation (20).

6.1.4 Comparison to Previous Work. We compare Telomere to previous real-time FTL: the **Worst-case and Average-case joint Optimization for Garbage Collection (WAO-GC)** [47]. We do not compare with Partitioned Real-Time FTL [28] since writes and GC are partitioned onto one-fourth of the flash chips to provide low and predictable read latency. Thus, the write bandwidth is significantly lower than the methods compared to. Due to the limited number of real-time FTLs to compare against, we also compare Telomere to non-real-time work by Stoica and Ailamaki [41] on improving flash write performance using update frequency. Their work partitions pages into sets with update frequencies that decrease in powers of 2. When a page becomes cold, it moves to a set with a lower update frequency. Similarly, when it becomes hotter, it moves to a set with a higher update frequency. Each set is stored as a log structure, and this algorithm is called *MultiLog data placement.* Stoica and Ailamaki [41] also provide algorithms for estimating page update frequency, including an *Oracle* algorithm that knows the exact page update frequency, and thus has the lowest GC overhead. We compare Telomere to MultiLog-Oracle. In addition, we apply bank reservation [16] to provide throughput admission control based on the average observed *WAF* value of MultiLog-Oracle for two different workloads. The work of Stoica and Ailamaki [41] is selected for comparison because the motivation is similar to ours in that data with similar update frequencies should be placed together. Real-time FTLs often sacrifice bandwidth for predictability. We show that Telomere achieves better throughput under certain over-provisioning levels even compared to non-real-time methods.

6.1.5 Comparison to WAO-GC. WAO-GC [47] builds upon the partial GC technique. In addition to real-time bounds, WAO-GC shows that it is able to achieve better average-case performance than Guarantee FTL [13] and Real-Time FTL [34] by using over-provisioning to delay GC. WAO-GC derives a maximum λ value using SSD parameters to guarantee that a page write will only be blocked by one partial GC step. With our SSD parameters, the upper bound of λ for WAO-GC is 0.74.

WAO-GC uses page-level mapping. Thus, the storage admission control needs can be calculated as a function of its over-provisioning. Specifically, the number of pages needed by tasks to store data has to be less than λ times the total number of physical flash pages.

$$\sum_{i=0}^{m}[w_i \cdot (l_i^w + 1)] \leq \lambda \cdot S_{pages} \tag{24}$$

Figure 9 shows the maximum throughput and storage utilizations at which 100% of the task sets are schedulable at different over-provisioning levels. When over-provisioning $\alpha = 0.10$, 10% of the physical space reserved for wear-leveling for each method, which is why in Figure 9(a) task sets that need more than 10% of the storage space are not schedulable. Specific wear-leveling techniques are outside the scope of this work, but we reserve the same capacity of the SSD for wear-leveling purposes to make a fair comparison of the different methods with flash endurance taken into account. Note that because Telomere does not copy valid pages during GC, it needs less over-provisioning than other methods that need to perform wear-leveling during GC to achieve

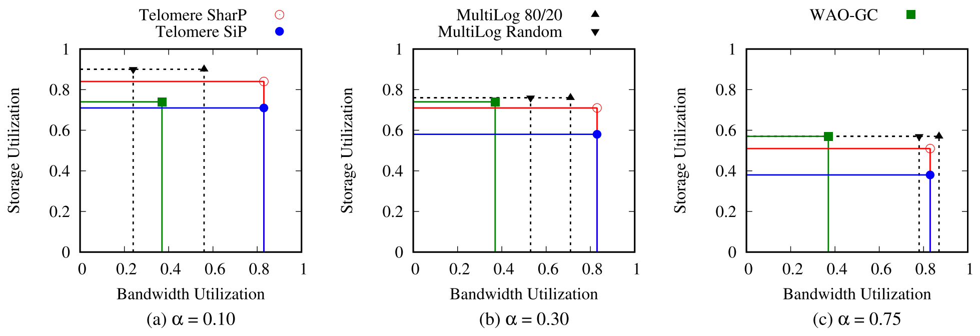

Fig. 9. Storage and throughput admission control. Lines indicate the maximum utilization at which 100% of the task sets are schedulable. MultiLog-Oracle method is plotted in dotted lines because the bank reservation throughput admission control [16] is not real time and the *WAF* is the average observed value, not the worst case.

the same flash endurance. We can see that the storage cost of Telomere SiP is 19% and Telomere SharP is 6%. Note that the storage utilization of WAO-GC cannot be higher 0.74 since WAO-GC requires a certain amount of over-provisioning to guarantee that a page write will only trigger one partial GC step. This is why at $\alpha = 0.10$, WAO-GC rejects more task sets than Telomere SharP even with Telomere's extra storage cost.

Figure 9 also shows the percentage of task sets schedulable under the throughput admission control with different over-provisioning values. The Telomere throughput admission control is calculated with Equation (20). WAO-GC does not provide admission control for multiple flash chips. It only guarantees that a write request will be blocked by no more than a partial GC step. We use the admission control in Equation (20) with the following write and GC capacities:

$$C_i^w = \left\lceil \frac{w_i}{S_{chips}} \right\rceil \cdot (t_w + max(t_e, t_r + t_w)), \tag{25}$$

$$C_i^g = 0. \tag{26}$$

The schedulability simulation shows that WAO-GC starts rejecting task sets at 38% throughput utilization, whereas Telomere SiP and SharP are 100% schedulable at 83% throughput utilization.

6.1.6 Comparison to MultiLog-Oracle. The MultiLog-Oracle algorithm can be integrated into a page-level mapping FTL, so the storage admission control can be calculated using Equation (24). For the throughput admission control, because MultiLog-Oracle is not real time, we will rely on the observed *WAF* value for two different workloads and use bank reservation [16] to provide a throughput admission control on average. Banks are dynamically partitioned to service read and write requests and perform GC based on the read and write throughput specified. Unlike Telomere's throughput admission control, bank reservation [16] does not guarantee that each task will be able to perform its specified number of page reads and writes within its period. It only guarantees that a total read and write throughput from all the tasks in the task set can be sustained on average.

When over-provisioning $\alpha = 0.10$, MultiLog-Oracle has an average write amplification factor experienced with a random workload at $WAF = 5.0$. Since our random workload generated may be different from theirs, we also use their lower average write amplification factor value for a skewed Zipf 80/20 distribution at $WAF = 2.0$. Table 2 summarizes the *WAF* for MultiLog-Oracle at different over-provisioning levels. Note that these *WAF* values are the average estimated under

Table 2. MultiLog-Oracle *WAF* for Different Over-Provisioning Levels [41]

α	Random	Zipf 80/20
0.10	$WAF = 5.00$	$WAF = 2.00$
0.30	$WAF = 2.20$	$WAF = 1.50$
0.75	$WAF = 1.25$	$WAF = 1.13$

empirical observation. They are not the worst-case *WAF* values. Therefore, we plot the MultiLog-Oracle method in dotted lines.

Figure 9(b) and (c) show the storage/throughput trade-off for MultiLog-Oracle at different levels of over-provisioning. Lower storage utilization in MultiLog-Oracle leads to higher throughput. The inverse relationship between throughput and storage utilization is because to achieve higher throughput, the write amplification factor has to be lower, which means more over-provisioning, which leads to lower storage utilization. Telomere SharP does not suffer from a decrease in throughput when storage utilization increases.

For over-provisioning $\alpha = 0.75$, Figure 9(c) shows that MultiLog-Oracle for a workload with a skewed Zipf 80/20 distribution has an observed *WAF* that admits a higher throughput using bank reservation compared to Telomere. This is because bank reservation [16] only guarantees that a total read and write throughput from all the tasks in the task set can be sustained on average, whereas Telomere's throughput admission control guarantees that each task will perform its specified number of page reads and writes within its period.

MultiLog-Oracle, like traditional FTLs, has a tunable storage/throughput trade-off. It can achieve either high storage utilization with low throughput utilization, as seen in Figure 9(a), or high throughput utilization with low storage utilization, as seen in Figure 9(c). Telomere, as seen in Figure 9(a), can achieve both high throughput and storage utilization.

6.2 Wear-Leveling Experiments

We also measured the effectiveness of wear-leveling for Telomere and Pagemap under different levels of over-provisioning using the health binning wear-leveling technique [33]. Pletka et al. [32] measured **raw bit error rates (RBER)** of the worst page obtained from the blocks of a real consumer-level 16-nm MLC flash chip and found that it is a function of the nominal endurance, defined as the program/erase cycle normalized to the manufacturer-specified block endurance. The wear of a 2D flash block of advanced age can be accurately modeled using the following log-log model:

$$log_{10}(W_b) = x_b + y_b \cdot log_{10}(E(b)), \tag{27}$$

where $E(b)$ denotes the program/erase cycle of block b normalized to the manufacturer-specified block endurance, and x_b and y_b are parameters obtained from large-scale characterization and are distinct for every block [32]. In the health binning wear-leveling technique, the health of a block is determined by Equation (27). The hottest data is placed in the healthiest blocks, and cold data is placed in less healthy blocks.

We run the event-driven simulator for Telomere and Pagemap under (1) no wear-leveling, where the free blocks queue is a First-In-First-Out queue, and (2) RBER-balancing wear-leveling, where the free blocks queue is sorted by the RBER of the blocks. Telomere is also run with health binning. The page-map FTL is not run with health binning because WAO-GC page-map FTL does not co-locate data with similar hotness. WAO-GC's partial GC constraints restrict valid pages of a victim block to be written to the same block as incoming write requests, regardless of data hotness. Thus, health binning cannot be used since it only works with FTLs that collocate together data with

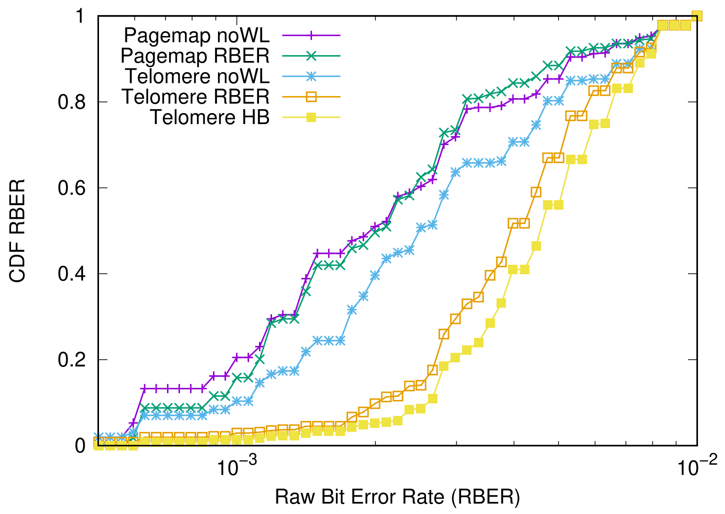

Fig. 10. CDF of the measured RBER at the end of a simulation run for a task set at over-provisioning $\lambda = 0.10$.

similar update frequency. For the simulation, we generate 20 task sets for each λ value from 0.70 to 0.95 in increments of 0.05. Each task set contains 10 tasks that are generated with relative standard deviation $v = 70$. The simulation runs until 2% of the blocks reach a wear value of $W_b = 10^{-2}$, which is the RBER threshold for declaring a block dead [48]. Pletka and Tomic [33] observed that once a small percentage of the blocks have been retired as they reach the error correction capability of the ECC, write amplification jumps abruptly, and the performance of the device drops suddenly. Thus, we end our simulation when 2% of the total number of flash blocks get retired.

Figure 10 shows the **Cumulative Distribution Function (CDF)** of the measured RBER at end of the simulation run for a task set with $\lambda = 0.10$ for Telomere and Pagemap with different wear-leveling algorithms. We tested the health binning algorithm (HB), RBER-balancing wear-leveling algorithm (RBER), and no wear-leveling with a First-In-First-Out free blocks queue (noWL). In the ideal case, the blocks would all wear out at the same time. In the graph, the CDF for the RBER would be 0% for RBER less than 10^{-2} and then 100% at 10^{-2}. Telomere HB and Telomere RBER are both more effective at wear-leveling than Telomere noWL, Pagemap RBER (page-map FTL with RBER wear-leveling algorithm), and Pagemap noWL (page-map FTL with no wear-leveling algorithm).

6.3 Hardware Experiments

We use the OpenSSD Cosmos board [39], pictured in Figure 11, to implement three FTLs: Telomere, WAO-GC, and page-map FTL. As shown in Figure 12, the board is connected via an external PCIe cable to a PC with an ASRock Z68 PRO3-M Motherboard and a 3.10-GHz Intel Core i3-2100 CPU running the Quest real-time operating system [49].

The Cosmos board includes the Zynq-7000 with dual ARM Cortex-A9 and NEON DSP co-processor for each core. The internal structure of a Zynq-7000 SoC has two components: the processing system (PS) and the **programmable logic (PL)**. The processing system component includes the dual-core ARM processor, the memory interfaces, and the I/O peripherals. The PL component includes the FPGA fabric. The flash storage controller is synthesized in the PL, and the FTL firmware is running on the ARM Cortex-A9.

The OpenSSD Cosmos board has two small outline dual in-line memory modules (SO-DIMMs), each containing Micron Technology's MLC NAND flash (with part number

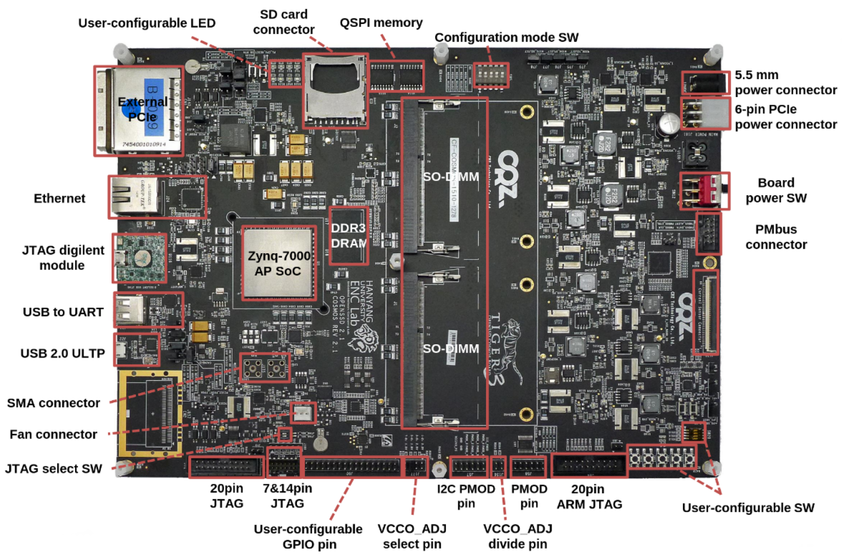

Fig. 11. OpenSSD Cosmos board [39].

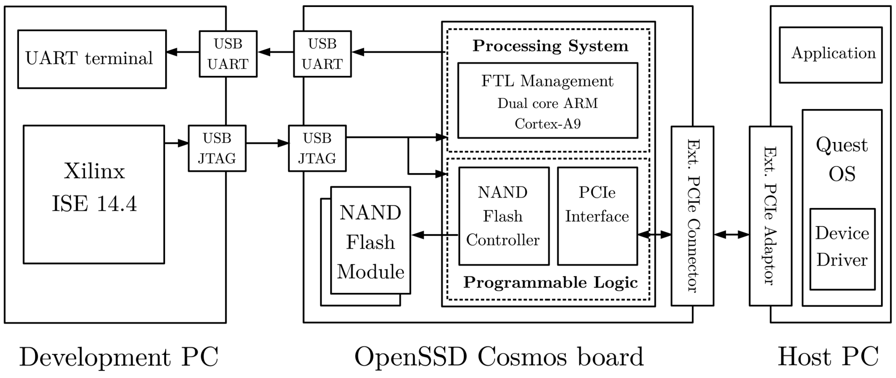

Fig. 12. OpenSSD Cosmos board setup.

MT29F256G08CMCABH2). A block contains 256 pages, and a page is 8 KB. The FTL sends commands to way controllers directly; however, it cannot access the channel controller including the way arbiter, page buffer, and the BCH ECC engine. The way arbiter grants permission in a round-robin manner for the way controllers to use the common flash bus or the internal data bus to access the page buffer. The page buffer stores 2 KB of data, 60 bytes of ECC parity, and 90 reserved bytes. Since a flash page is 8 KB, data transfer between the page buffer and the encoder/decoder occurs four times for each flash page.

The FTL is implemented in the ARM Cortex-A9 and sends commands to the way controllers directly. To perform a page write, when the way arbiter grants access to the internal data bus, the command is issued and data is moved from DRAM to the page buffer. The data is then transferred to the ECC encoder that calculates the parity and transfers data and parity to the page buffer. Then, data is transferred to the way controller and finally to the NAND flash when the way arbiter grants access to the common flash bus. To perform a page read, data arrive from the way controller and

Table 3. Tasks Running on the FTL Implementations on the OpenSSD Cosmos Board

	r_i	T_i^r	w_i	T_i^w	l_i^w
τ_0	16	20	16	80	250
τ_1	8	20	16	40	33
τ_2	16	20	16	40	330
τ_3	0	0	16	40	990
τ_4	0	0	11	40	1,980
τ_5	0	0	5	40	1,980

are transferred to the ECC decoder. If there are errors in the data, the ECC decoder corrects the data and transfers the data to the page buffer. Data is then transferred to DRAM.

The device driver splits up each read or write request into flash page-size requests and inserts them into a request circular buffer. The FTL retrieves the requests and orders them according to EDF and starts handling the request if the flash chip is not busy. A flash page read request will occur on a physical page that exists on a specific flash chip. A flash page write request is assigned to a flash chip depending on whether there is a current flash block assigned for that task. If a block is assigned, the page write will occur on a specific flash chip. Otherwise, the flash page write request could occur on any available flash chip. In our FTL implementation, we use page-level mapping and keep track of a free blocks list, which is used when a partition of tasks require a new block. When a flash block expires (i.e., all the pages in the block become invalid), a block erasure is triggered and the block is added to the free blocks list.

Table 3 shows the parameters of the read and write tasks. We ran an experiment with six write tasks and four read tasks. The task set is synthetic with parameters that maximized the read and write throughputs for the OpenSSD Cosmos board and highlighted the advantage of SharP partitioning. As Figure 8 shows, when the number of tasks is small, the maximum storage utilization for SharP is only slightly higher than SiP. We selected parameters for this task set to show the difference between SiP and SharP. We ran the experiment with 21 blocks per flash chip. The limit on the number of blocks per flash chip is due to time constraints. The bookkeeping takes place on the FTL with metadata per page access, logging latency to verify that timing guarantees are met. This hardware experiment is meant to demonstrate the effectiveness of the different partitioning algorithms running on real hardware. It is not a case study of a real application. However, we believe it shows the storage utilization advantage of the SharP partitioning algorithm and the throughput guarantee based on our analysis. For varying task sets with different parameters, please see our simulation results in Section 6.1.

We test four methods: Telomere SiP, Telomere SharP, WAO-GC, and page-map FTL. Figures 13 and 14 show the four methods, three of which run at different write throughputs due to the rejected tasks. Telomere SharP and Pagemap start out with the highest write throughput, running all six tasks at 1,800 pages per second. Telomere SiP's storage admission control rejects τ_5, so it runs at a lower throughput of 1,680 pages per second. WAO-GC rejects τ_3, τ_4, and τ_5 due to insufficient logical address space and write throughput, so it runs at 1,000 pages per second. Telomere SiP, SharP, and WAO-GC are all able to maintain the throughput accepted by their admission control when read requests arrive at around $t = 97$ as shown in Figure 14. The non-real-time page-map FTL, however, is unable to maintain the write throughput and read throughput as the number of pages written per second drops and both read throughput and write throughput fluctuate.

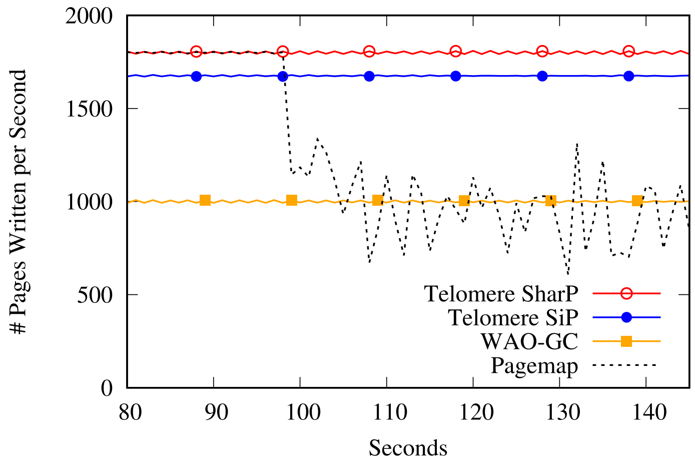

Fig. 13. Write throughput for the task set in Table 3 for Telomere SiP and SharP, WAO-GC, and page-map FTL. Telomere SiP and WAO-GC run at a lower write throughput because their admission control could not accept the entire task set.

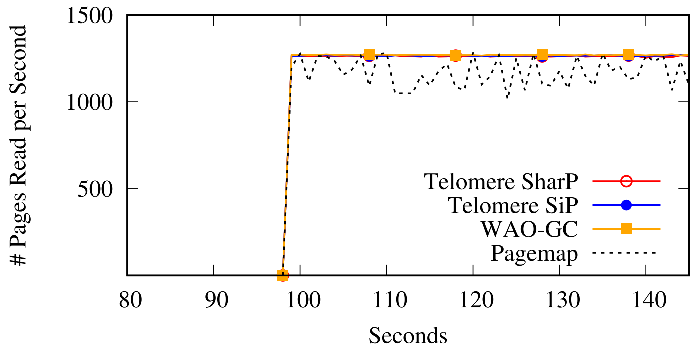

Fig. 14. Read throughput for the task set in Table 3 for Telomere SiP and SharP, WAO-GC, and page-map FTL. After read requests arrive at around $t = 97$, Pagemap is not able to maintain its read and write throughput due to GC overhead.

7 RELATED WORK

A large body of work focuses on reducing write amplification by storing together data with similar update frequencies. Rosenblum and Ousterhout [35] first pointed out that categorizing data as hot or cold reduces cleaning overhead in a log-structured file system. This was followed by works that tuned LFS on flash systems [19, 43]. Other research that detects data temperature can be found in flash wear-leveling works such as Dynamic Age Clustering [12] as well as data placement algorithms based on update frequency [31, 41]. However, drive-managed SSDs that provide the traditional block I/O interface suffer from many shortcomings [2, 11, 15], including log-on-log issues [46]. Solutions to these problems fall into two main categories: host-managed designs [7, 8, 18, 22] and some form of information sharing between the host and the FTL [17, 23, 45]. The latter is the approach we have taken with Telomere.

Multi-stream SSDs show how write amplification can be reduced by assigning data with different update frequencies to different streams, which are stored at a different physical location [17, 23, 45]. Multi-streamed SSDs [17] and WARM [23] assign different stream IDs to different types of data (index files, log files, sstables, etc.), however, files of the same type may contain data with different lifetime. AutoStream [45] automatically assigns stream IDs to data. Their experiments show that *WAF* gets close to 1 when there is a large data lifetime difference with 4 streams of data. However, with 16 streams of data with different lifetimes, AutoStream takes time to differentiate blocks and some requests are mixed into 1 stream, resulting in *WAF* above 2. AutoStream does not disclose crucial details including the over-provisioning, making it hard to compare against.

Previous work on reducing flash latency includes dividing GC for reclaiming a flash block into partial steps that are distributed among host write requests [13]. There are a few real-time FTLs that build upon the partial GC technique [34, 47]. However, these FTLs do not exploit the high level of parallelism that exist in SSDs. Other work for exploiting redundancy for predictable read performance include Flash on Rails [38] and PaRT-FTL [28]. Flash on Rails [38] uses two SSDs, one performing write requests and the other performing read requests, and periodically exchanges the SSDs and synchronizes data. Although this design provides high read throughput, it can suffer from unpredictable write performance since it has no direct control of GC activities. PaRT-FTL [28] uses a RAID design to rebuild read requests so that reads are never blocked by writes or GC. However, because reads and writes are partitioned onto different parallel units in flash, the write bandwidth is much lower.

8 CONCLUSION

The out-of-place property of flash memory requires GC to be performed when reclaiming a block. When blocks contain data with different lifetime, GC can incur long latency and cause throughput to drop and fluctuate. However, by intelligently placing data such that all the pages in a block being reclaimed are invalid, we can minimize the GC overhead to simply a block erasure. We present a new interface that provides the drive with information about the lifetime of the data. Our results show that Telomere's real-time admission control is able to guarantee tasks their required read and write operations within their periods. Under randomly generated tasksets containing 500 tasks, Telomere achieves 30% higher throughput with a 5% storage cost compared to pre-existing techniques.

ACKNOWLEDGMENTS

Special thanks to Craig Einstein for the name of Telomere.

REFERENCES

[1] Evan Ackerman. 2016. Autonomous Mini Rally Car Teaches Itself to Powerslide. Retrieved November 20, 2021 from https://spectrum.ieee.org/cars-that-think/transportation/self-driving/autonomous-mini-rally-car-teaches-itself-to-powerslide.

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigraphy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference (ATC'08).*

[3] Amara D. Angelica. 2013. Google's Self-Driving Car Gathers Nearly 1 GB/sec. Retrieved November 20, 2021 from http://www.kurzweilai.net/googles-self-driving-car-gathers-nearly-1-gbsec/.

[4] Theodore P. Baker. 1990. A stack-based resource allocation policy for realtime processes. In *Proceedings of the Real-Time Systems Symposium (RTSS'90).*

[5] Miriam Berger. 2020. How the world's beaches are readying for a summer of social distancing. *Washington Post.* Retrieved November 20, 2021 from https://www.washingtonpost.com/world/2020/06/27/how-worlds-beaches-are-readying-summer-social-distancing/.

[6] Enrico Bini and Giorgio C. Buttazzo. 2005. Measuring the performance of schedulability tests. *Real-Time Systems* 30, 1-2 (2005), 129–154.

[7] Matias Bjorling. 2019. From open-channel SSDs to zoned namespaces. In *Proceedings of the Linux Storage and Filesystems Conference (Vault'19).*

[8] Matias Bjorling, Philippe Bonnet, Luc Bouganim, and Niv Dayan. 2013. The necessary death of the block device interface. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR'13)*.

[9] Matias Bjorling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux open-channel SSD subsystem. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'17)*.

[10] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal parallelism of flash memory-based solid-state drives. *ACM Transactions on Storage* 12, 3 (2016), Article 13, 39 pages.

[11] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*.

[12] Mei-Ling Chiang and Ruei-Chuan Chang. 1999. Cleaning policies in mobile computers using flash memory. *Journal of System Software* 48, 3 (1999), 213–231.

[13] Siddharth Choudhuri and Tony Givargis. 2008. Deterministic service guarantees for NAND flash using partial block cleaning. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*.

[14] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. 2015. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*.

[15] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. 2016. The tail at store: A revelation from millions of hours of disk and SSD deployments. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'16)*.

[16] Sheng-Min Huang and Li-Pin Chang. 2018. Providing SLO compliance on NVMe SSDs through parallelism reservation. *ACM Transactions on Design Automation of Electronic Systems* 23, 3 (2018), Article 28, 26 pages.

[17] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The multi-streamed solid-state drive. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'14)*.

[18] Yangwook Kang, Jingpei Yang, and Ethan L. Miller. 2011. SCM: An efficient interface for storage class memories. In *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies (MSST'11)*.

[19] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. 1995. A flash-memory based file system. In *Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems*. 155–164.

[20] Jaeho Kim, Donghee Lee, and Sam H. Noh. 2015. Towards SLO complying SSDs through OPS isolation. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'15)*.

[21] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, and Sang-Won Lee. 2011. Fast, energy efficient scan inside flash memory SSDs. In *Proceedings of the International Workshop on Accelerating Data Management Systems*.

[22] Youyou Lu, Jiwu Shu, and Weimin Zheng. 2013. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'13)*.

[23] Yixin Luo, Yu Cai, Saugata Ghose, Jongmoo Choi, and Onur Mutlu. 2015. WARM: Improving NAND flash memory lifetime with write-hotness aware retention management. In *Proceedings of the Conference on Mass Storage Systems and Technology (MSST'15)*.

[24] Dongzhe Ma, Jianhua Feng, and Guoliang Li. 2014. A survey of address translation technologies for flash memories. *ACM Computing Surveys* 46, 3 (2014), 36.

[25] Micron. 2005. *Micron Technical Report: Small-Block vs. Large-Block NAND Flash Devices*. Technical Report TN-29-07. Micron.

[26] Micron. 2017. Micron Reveals Critical Technologies for Autonomous Vehicles. Retrieved November 20, 2021 from https://investors.micron.com/news-releases/news-release-details/micron-reveals-critical-technologies-autonomous-vehicles.

[27] Micron. 2018. NAND Flash Die—128Gb Die: X8 300mm MLC MT29F128G08CBECB. Retrieved November 20, 2021 from https://prod.micron.com/media/documents/products/data-sheet/nand-flash/die/l95b_die_128gb_nand.pdf.

[28] Katherine Missimer and Richard West. 2018. Partitioned real-time NAND flash storage. In *Proceedings of the Real-Time Systems Symposium (RTSS'18)*.

[29] Vidyabhushan Mohan, Taniya Siddiqua, Sudhanva Gurumurthi, and Mircea R. Stan. 2010. How I learned to stop worrying and love flash endurance. In *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems (FAST'10)*.

[30] HakJune Oh. 2013. Single Controller 4/8TB SSD. Retrieved November 20, 2021 from https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2013/20130815_301A_Oh.pdf.

[31] Dongchul Park and David H. C. Du. 2011. Hot data identification for flash-based storage systems using multiple Bloom filters. In *Proceedings of the Conference on Massive Storage Systems and Technology (MSST'11)*.

[32] Roman Pletka, Ioannis Koltsidas, Nikolas Ioannou, Sasa Tomic, Nikolaos Papandreou, Thomas Parnell, Haralampos Pozidis, Aaron Fry, and Tim Fisher. 2018. Management of next-generation NAND flash to achieve enterprise-level endurance and latency targets. *ACM Transactions on Storage* 14, 4 (2018), Article 33, 25 pages.

[33] Roman A. Pletka and Sasa Tomic. 2016. Health-binning: Maximizing the performance and the endurance of consumer-level NAND flash. In *Proceedings of the 9th ACM International Conference on Systems and Storage (SYSTOR'16)*.

[34] Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao. 2012. Real-time flash translation layer for NAND flash memory storage systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'12)*.

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

[36] Gustavo Henrique Ruffo. 2019. Tesla Cars Have a Memory Problem That May Cost You a Lot to Repair. https://insideevs.com/news/376037/tesla-mcu-emmc-memory-issue/.

[37] SanDisk. 2018. iNAND Automotive Embedded Flash Drives. Retrieved November 20, 2021 from https://www.sandisk.com/oem-design/automotive/inand.

[38] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. 2014. Flash on rails: Consistent flash performance through redundancy. In *Proceedings of the USENIX Annual Technical Conference (ATC'14)*.

[39] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. 2014. Cosmos OpenSSD: A PCIe-Based Open Source SSD Platform. Retrieved November 20, 2021 from http://www.flashmemorysummit.com/English/Collaterals/Proceedings/\2014/20140807_301B_Song.pdf.

[40] Josh Spires. 2020. How Drones Have Helped Fight COVID-19—And Become More Mainstream. Retrieved November 20, 2021 from https://dronedj.com/2020/06/04/.

[41] Radu Stoica and Anastasia Ailamaki. 2013. Improving flash write performance by using update frequency. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'13)*.

[42] Western Digital. 2019. Zoned Storage. Retrieved November 20, 2021 from http://zonedstorage.io.

[43] Michael Wu and Willy Zwaenepoel. 1994. eNVy: A non-volatile, main memory storage system. *ACM SIGPLAN Notices* 29, 11 (1994), 86–97.

[44] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'17)*.

[45] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. 2017. AutoStream: Automatic stream management for multi-streamed SSDs. In *Proceedings of the ACM International Conference on Systems and Storage (SYSTOR'17)*.

[46] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. 2014. Don't stack your log on my log. In *Proceedings of the Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*.

[47] Qi Zhang, Xuandong Li, Linzhang Wang, Tian Zhang, Yi Wang, and Zili Shao. 2015. Optimizing deterministic garbage collection in NAND flash storage systems. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS'15)*. IEEE, Los Alamitos, CA.

[48] K. Zhao, W. Zhao, H. Sun, T. Zhang, X. Zhang, and N. Zheng. 2013. LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'13)*.

[49] R. West, Y. Li, E. Missimer, and M. Danish. 2016. A virtualized separation Kernel for mixed criticality systems. *ACM Transactions on Computer Systems* 34 (2016).

Received January 2021; revised June 2021; accepted July 2021