Partitioned Real-Time NAND Flash Storage

Katherine Missimer
Boston University
kzhao@cs.bu.edu

Richard West
Boston University
richwest@cs.bu.edu

***Abstract*—This paper addresses the problem of guaranteeing performance and predictability of NAND flash memory in a real-time storage system. Our approach implements a new flash translation layer scheme that exploits internal parallelism within solid state storage devices. We describe the Partitioned Real-Time Flash Translation Layer (PaRT-FTL), which splits a set of flash chips into separate read and write sets. This ensures reads and writes to separate chips proceed in parallel. However, PaRT-FTL is also able to rebuild the data for a read request from a flash chip that is busy servicing a write request or performing garbage collection. Consequently, reads are never blocked by writes or storage space reclamation. PaRT-FTL is compared to previous real-time approaches including scheduling, over-provisioning and partial garbage collection. We show that by isolating read and write requests using encoding techniques, PaRT-FTL provides better latency guarantees for real-time applications.**

I. Introduction

With the advent of autonomous vehicles, from driverless cars to unmanned aerial vehicles (UAVs), there is a growing need to store, retrieve and process large volumes of sensor data [11]. Autonomous vehicles rely on the real-time processing of sensor data to perform collision avoidance, path planning, object detection, 3D scene reconstruction, simultaneous localization and mapping (SLAM), and other mission tasks. 3D cameras, laser range finders, radar, sonar and inertial sensors provide numerous data streams, from high definition images to point clouds. Google's self-driving car is already reported to generate on the order of 1GB/s of data from its various onboard sensors [1]. As autonomous vehicle technology advances, we can expect the quantity of sensor data produced per unit time to be even greater, requiring local as well as cloud storage. The sheer volume of sensor data dictates the need for real-time information storage and retrieval, to accomplish machine learning and mission objectives.

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. However, NAND flash memory management is complicated by the fact that in-place updates are not possible. Once a memory location is written to, it must be erased before being written to again. In addition, reads and writes operate at the granularity of a flash *page*, while erasures operate on flash *blocks* spanning multiple pages.

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, garbage collection 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 garbage collection, it could be blocked waiting for one block erasure and $P-1$ read and write operations to copy the valid pages. Techniques that have been used for real-time flash storage to reduce the time needed to perform garbage collection include over-provisioning and partial garbage collection [16] [23] [7]. With over-provisioning, the logical address space is a fraction of the physical address space, so in the worst case, only a fraction of the number of pages in a block will need to be copied during garbage collection. Another technique is performing partial garbage collection, which divides garbage collection into several small steps. This guarantees that a request will only be blocked by a partial step.

This paper proposes a solution that significantly reduces the latency of a flash page write and read. In an age of data gathering and mining, there is an increasing need for real-time systems to be able to fetch data that cannot all be stored in memory in order to make time-critical decisions. This paper presents a partitioned real-time flash translation layer (PaRT-FTL) that guarantees bounded, low latency read and write requests by taking advantage of the internal parallelism in a solid-state storage device (SSD). Input and output (I/O) requests are partitioned so that read requests are not blocked by write requests or garbage collection.

The rest of the paper is as follows: Section II summarizes the internal features of NAND flash storage systems. Section III describes the FTL partitioning, followed by a description of the real-time task model, and admission control for a set of tasks. Related work is discussed in Section IV, to introduce comparable approaches to PaRT-FTL, before they are subsequently compared in the evaluation section. Section V describes the implementation of PaRT-FTL. This is followed by Section VI, which presents our results for both simulations and PaRT-FTL implemented on the Cosmos OpenSSD board. Finally, conclusions and future work are described in Section VII.

II. NAND Flash Internals

This section explains the physical layout of NAND flash memory, and the parallelism that exists in modern SSDs, as shown in Figure 1.

The internal structure of flash storage is significantly different to that of traditional mechanical hard drives. In flash devices, the smallest read and write unit is a page. A page used to be standardized at 512 and 2048 bytes [13]. However, recently much larger page sizes have been seen ranging from

4 to 16 KB [14]. 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) 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. In the earlier days, a flash block contains 32 or 64 pages [13]. Nowadays, it can range from 128 to 512 pages [14]. Multiple blocks form a plane, and typically two to four planes form a die. The flash die is the smallest unit that can independently execute commands or report status. Typically, 1, 2, 4, 8 or 16 flash dies form a flash chip. A flash chip exists on a specific way on a specific channel. There are usually 4 to 8 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. There are usually 4 to 8 channels in the SSD as flash controllers with 16 or 32 channels are too complex, too big and consume too much power [10] [15]. Each channel contains its own NAND interface block and error correction code block, so it can operate independently. This is called channel striping. Way interleaving and channel striping are the two main methods of parallelization that modern flash controllers support [6].

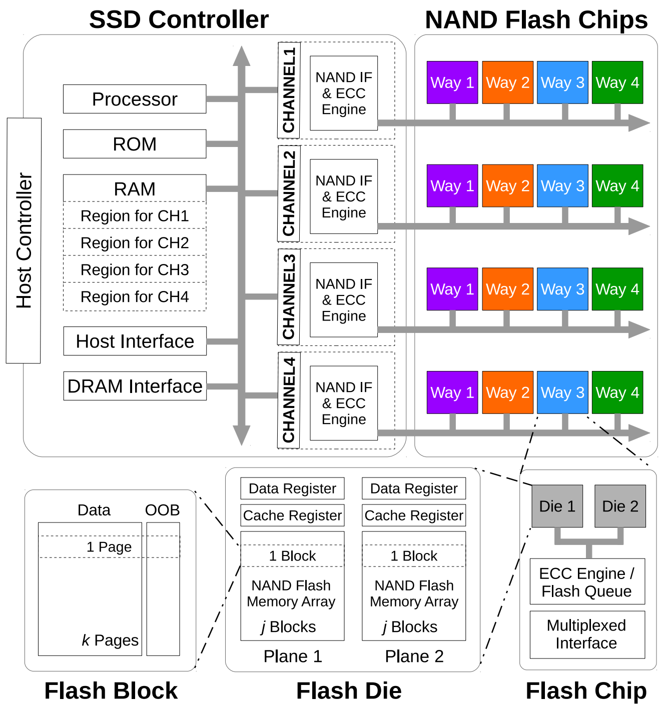


Fig. 1: SSD internal architecture.

A. Flash Parallelism Observations

We measured the effects of way interleaving and channel striping for page-based reads and writes, and block-based erasures, using the OpenSSD Cosmos Board [20]. The way arbiter in the Cosmos Board grants access to the shared buses in a round-robin manner. Detailed hardware specifications are in Section VI. While our focus is on Micron Technology's MLC NAND flash, way interleaving and channel striping are common characteristics found in other modern NAND flash technologies. Parallelism within a flash chip (i.e. die and plane) is not explored due to hardware limitations. Each Micron Technology NAND flash chip contains one flash die, and plane parallelism could not be exploited due to limitations of the OpenSSD FPGA implementation. For the rest of the paper, we therefore use flash chip and flash die interchangeably.

	Min	Max	Avg	Stddev
4 same die writes	0.844	9.70	5.18	2.50
4 way writes	0.826	2.90	1.87	0.645
4 channel writes	0.598	2.37	1.33	0.631
4 same die reads	0.856	1.44	1.04	0.040
4 way reads	0.833	1.25	1.23	0.081
4 channel reads	0.369	0.382	0.375	0.004
4 same die erasures	4.51	16.2	12.6	3.05
4 way erasures	2.58	4.06	3.84	0.111
4 channel erasures	2.77	4.06	3.84	0.114

TABLE I: Latency in milliseconds for flash operations on an SSD with 4 channels and 4 ways per channel, showing the effects of way interleaving and channel striping. Read, write, and erase operations are parallelizable by channel striping. However, read operations do not show any performance benefits under way interleaving while write and erasure operations do.

For each flash operation (read, write and erasure), the latency of the operation is measured when performed four times on the same flash die, on different flash chips that exist in the same channel (way interleaving), or on different flash chips that exist in different channels (channel striping). Table I shows the results.

The maximum and average latency for page writes are reduced by both way interleaving and channel striping. The slight slowdown in the way interleaving compared to channel striping is most likely due to accessing the page buffer, which is shared among all the flash chips in a channel.

For page read operations, while the maximum and average latency are reduced by channel striping, no significant improvements are seen for way interleaving. Read operations do not show any performance benefits under way interleaving because the majority of the time is spent accessing the page buffer, which cannot be performed in parallel during way interleaving. On average, reads with way interleaving actually perform worse than reads on the same flash die. We hypothesize that the extra time comes from performing status checks on the different flash chips.

For block erasures, the maximum and average latency are reduced by both channel striping and way interleaving.

In summary, we observe that read, write, and erase operations are parallelizable by channel striping. However, read operations do not show any performance benefits under way interleaving while write and erasure operations do.

B. Flash Translation Layer

The flash translation layer (FTL) is the firmware on the SSD that addresses challenges in flash memory such as the lack of in-place updates and block endurance. By taking care of garbage collection and wear-leveling, in addition to providing a logical to physical address mapping, the FTL allows flash memory to appear as a block device. This permits file systems to interact with an SSD transparently, similar to how a file system would interact with a mechanical hard drive. Although wear-leveling is important for increased block endurance, it is not the focus of this work and will not be discussed in detail.

One of the address mapping algorithms commonly used is page-level mapping, where there is a one-to-one translation of a logical address to a physical page. This scheme efficiently utilizes blocks in flash, but it requires a large SRAM to be able to store the mapping table. When garbage collection is invoked, a block is selected to be erased. All valid pages in that block are copied to a clean block, and the mappings are updated. Figure 2 shows an example of page-level mapping and block reclamation.

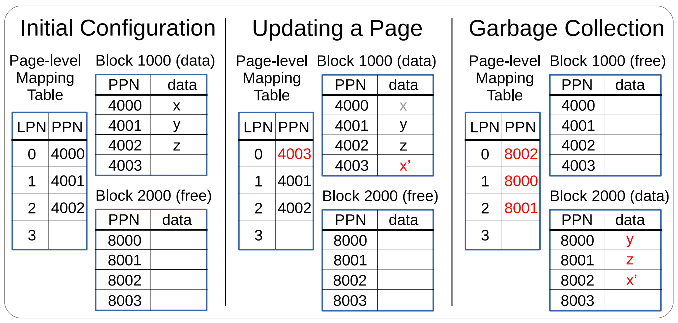


Fig. 2: Page-level mapping and block reclamation. Every logical page number (LPN) is mapped to a physical page number (PPN). When overwriting LPN=0, the updated data x' will be written to a different physical page since in-place updates cannot be performed; the mapping is updated accordingly. When the SSD fills up, garbage collection is triggered and valid pages in a victim block are copied to a free block and the victim block is erased.

III. Design

PaRT-FTL is designed for real-time systems with hard deadlines associated with the storage and retrieval of persistent data. For example, an autonomous vehicle management system might require the processing, storage and retrieval of multiple data-intensive sensor streams, including video images and point clouds to render a 3D map of its surroundings as it performs simultaneous localization and mapping. We envision scenarios for next-generation real-time applications where main memory has insufficient capacity to store all the data needed for information processing, machine learning, path planning, decision making and other mission-critical tasks. In particular, having low-latency access to stored data that can be processed and augmented with updated sensor information is particularly relevant to our intended usage of PaRT-FTL.

The design of PaRT-FTL is motivated by our observations of the behavior of NAND flash memory, as described in Table I. To achieve predictable read performance for real-time workloads, read and write requests are partitioned onto different flash chips and parity pages are calculated using XORs. Read requests of pages on flash chips that are servicing write requests are rebuilt using the parity page. In this way, read requests are never blocked by write requests or garbage collection. PaRT-FTL was designed with the following goals:

- an FTL design that takes advantage of internal parallelism in SSDs;
- a real-time task model for read and write requests on multiple flash chips;
- bounded and low-latency read requests that are not blocked by write requests or garbage collection.

A. FTL Data Layout

PaRT-FTL partitions the set of flash chips into write flash chips F_w and read flash chips F_r. Table II contains symbol definitions. F_w is the set of flash chips servicing only write requests and performing garbage collection, and F_r is the set of flash chips servicing only read requests. These two sets are mutually exclusive. When servicing a read request, if the physical page exists on a write flash chip, the page is rebuilt by reading the associated encoding page and data pages in F_r. Given the observations from Table I, F_w should contain flash chips from different channels in order to maximize read performance since read operations do not show any performance benefits under way interleaving. For example, in the SSD layout depicted by Figure 3, 4 flash chips are servicing write requests and 12 flash chips are servicing read requests. F_w contains flash chips on Way 1 of each channel. Read requests for pages in Ways 2 and 3 are handled normally while read requests for pages in Way 1 will be rebuilt by reading and decoding the corresponding pages in Ways 2, 3 and 4. Data is encoded and decoded using XORs. In the example in Figure 3, each parity page is the XOR of its corresponding data pages in the same channel.

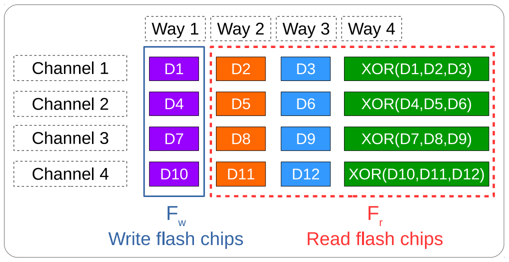


Fig. 3: Flash chip layout. In this example, there are 12 flash chips storing data and 4 flash chips storing encoding pages. Flash chips being written to are on Way 1 of each channel, while other flash chips are servicing read requests.

After a block of data is written to each write chip, F_w rotates to a different set of flash chips. For example, in Figure 4, F_w rotates to be the flash chips in Way 2, and flash chips in Ways 1, 3 and 4 will be used to service read requests.

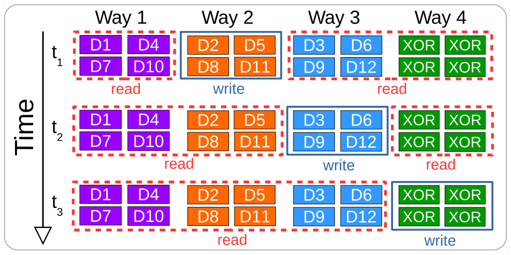


Fig. 4: The set of write flash chips rotates to a different way after a block of data is written to each write flash chip.

B. Real-Time Task Model

Let $\{\tau_1, \tau_2, ..., \tau_n\}$ be a set of n periodic tasks. Each task τ_i guarantees that an application can perform r_i page reads every T_i^r time units and w_i page writes every T_i^w time units. Note that τ_i, which has parameters $[(r_i, T_i^r), (w_i, T_i^w)]$, does not account for the CPU computation time. These tasks exist on the FTL and utilize the NAND bus. A task is assumed to be scheduled on the CPU in a way that is able to guarantee the above read and write request rates on the SSD.

For a read request, the worst-case scenario is that the page exists on a flash chip being written to and the page needs to be rebuilt. To rebuild the page, all the associated data and encoding pages have to be read. For scheduling purposes, we set the read capacity C_i^r for task τ_i as follows:

$$C_i^r = r_i \cdot (t_r + t_c^d) \tag{1}$$

where t_r is the time it takes to read a flash page on every read flash chip, and t_c^d is the time it takes to decode a page.

A write request is first written to a buffer and later written to flash chips in F_w. Since there are no in-place updates in flash memory, a write request consisting of multiple pages can be distributed to different flash chips. Recall that page writes are parallelizable through both channel striping and way interleaving, and thus, parallelizable across every flash chip in $|F|$. Since only $|F_w|$ flash chips are servicing write requests, the write capacity C_i^w for task τ_i is the following:

$$C_i^w = \lceil \frac{w_i}{|F_w|} \rceil \cdot t_w \tag{2}$$

where t_w is the time it takes to write a flash page on every write flash chip. Note that t_r and t_w depends on the configuration of F_r and F_w, which determines how much way and channel parallelism exists.

1) Updating Encoding Pages: When a block of new data has been written to each of the data flash chips, a block of encoding has to be written to each of the encoding flash chips. Let k be the number of flash chips storing data, m be the number of flash chips storing encoding information, and P be the number of pages in a block. The write granularity is k/m pages so that each write operation to the SSD results in one page written to each way and the corresponding parity page is updated. After $k \cdot P$ pages of new data has been written, $m \cdot P$ encoding pages are updated. For a task τ_i, if $w_i = k \cdot P$, then $m \cdot P$ encoding pages are updated every write period T_i^w, so the period for the encoding task is the same as the write period. If $w_i \neq k \cdot P$, then the period for the encoding task is the write period multiplied by $\frac{k \cdot P}{w_i}$ to ensure that all the encoding pages can be written. For each task τ_i, if $w_i > 0$, then an encoding task with the following encoding capacity C_i^e and period T_i^e exists:

$$\begin{aligned} C_i^e &= m \cdot P \cdot t_c^e + \lceil \frac{m \cdot P}{|F_w|} \rceil \cdot t_w \\ T_i^e &= T_i^w \cdot \frac{k \cdot P}{w_i} \end{aligned} \tag{3}$$

where t_c^e is the time it takes to compute an encoding page.

Symbol	Definition
F_w	Set of flash chips servicing write requests
F_r	Set of flash chips servicing read requests
k	Number of flash chips storing data
m	Number of flash chips storing encoding info
P	Number of flash pages in a flash block
τ_i	A periodic task
r_i	Number of page read requests from τ_i
C_i^r	Read capacity for τ_i
T_i^r	Period for read requests from τ_i
w_i	Number of page write requests from τ_i
C_i^w	Write capacity for τ_i
T_i^w	Period for write requests from τ_i
C_i^e	Encoding capacity for τ_i
T_i^e	Encoding period for τ_i
C_i^g	Garbage collection capacity for τ_i
T_i^g	Garbage collection period for τ_i
t_r	Time to read a page on every flash chip in F_r
t_w	Time to write a page on every flash chip in F_w
t_e	Time to erase a block on every flash chip in F_w
t_c^e	Time to encode a parity page
t_c^d	Time to decode a page using parity
λ	Ratio of logical to physical address space
α	Lower bound of reclaimed pages in a block

TABLE II: Symbol Definitions.

2) Garbage Collection: When a flash chip in F_w runs out of free pages, garbage collection is triggered on that chip. So, every task with $w_i > 0$ will have a corresponding garbage collection (GC) task to ensure that enough free pages are reclaimed for task τ_i to write w_i pages every period T_i^w.

When garbage collection starts, a victim block is selected and valid pages are copied from the victim block to a free block. Then, the victim block is erased and the number of invalid pages that were previously in the victim block are reclaimed. Over-provisioning provides a lower bound on the number of invalid pages that exist. With over-provisioning, the logical address space becomes a fraction λ of the physical address space in the SSD.

$$\lambda = \frac{\text{logical address space}}{\text{physical address space}} \tag{4}$$

When garbage collection selects a block with the smallest number of valid pages to reclaim, at most $\lceil \lambda \cdot P \rceil$ pages need to be copied. This is because in the worst-case, the number of valid pages are spread out evenly among all blocks. The number of valid pages that need to be copied in all the data flash chips can be upper bounded by $k \cdot \lceil \lambda \cdot P \rceil$. When k blocks of data become invalid, their corresponding encoding also becomes invalid. Therefore, $m \cdot \lceil \lambda \cdot P \rceil$ is the number of valid encoding pages that need to be copied. Thus, to reclaim $k \cdot \alpha$ pages, where $\alpha = P - \lceil \lambda \cdot P \rceil$ is the lower bound of reclaimed pages in a block, $(k + m) \cdot \lceil \lambda \cdot P \rceil$ pages need to be copied and $(k + m)$ blocks erased. For a task τ_i, if $w_i = k \cdot \alpha$, then garbage collection needs to reclaim a block every write period T_i^w, so the period for the GC task is the same as the write period. If the number of pages written w_i, is more than $k \cdot \alpha$ pages, then the GC task needs to guarantee that at least w_i pages are reclaimed every write period T_i^w. Since erasures happen at the block level and not page level, the GC task needs to upper bound the number of pages reclaimed to a multiple of blocks. Thus, even if w_i is just one more than $k \cdot \alpha$ pages, two blocks will need to be reclaimed every T_i^w. Similarly, if the w_i is only half of $k \cdot \alpha$ pages, then the GC task only needs to reclaim a block every $2 {\cdot} T_i^w$. However, if more than half of $k \cdot \alpha$ pages is requested, the GC task will need to guarantee a block every T_i^w, thus, the floor is used. If $w_i > 0$, then a garbage collection task with the following capacity C_i^g and period T_i^g is established for task τ_i:

$$\begin{aligned} C_i^g &= \lceil \frac{k+m}{|F_w|} \rceil \cdot (\lceil \lambda {\cdot} P \rceil (t_r' + t_w) + t_e) \\ T_i^g &= \begin{cases} T_i^w / \lceil \frac{w_i}{k \cdot \alpha} \rceil, & \text{if } w_i > k \cdot \alpha \\ T_i^w \cdot \lfloor \frac{k \cdot \alpha}{w_i} \rfloor, & \text{otherwise} \end{cases} \end{aligned} \tag{5}$$

where t_e and t_r' are the latency for block erasure and page read, respectively, on every write flash chip in F_w.

C. Admission Control

A schedulability test is invoked for each chip set to ensure that all the read and write requests are schedulable. Baker's Stack Resource Policy (SRP) [2] with Earliest Deadline First (EDF) is used. Under SRP with EDF, each job of a task is assigned a priority according to its absolute deadline and a static preemption level that is inversely proportional to its relative deadline. Each shared resource is assigned a ceiling which is the maximum preemption level of all the tasks that will lock this resource. A system ceiling is defined as the highest ceiling of all resources currently locked. A job is not allowed to start executing until its priority is the highest among the active tasks and its preemption level is greater than the system ceiling. In this way, SRP guarantees that once a job is started, it can only be preempted by higher priority tasks and it will not be blocked for the duration of more than one critical section of a lower priority task [17].

Since read requests are isolated from write requests and occur on separate flash chips, a separate schedulability test is provided for read and write requests. For the read requests that are serviced by flash chips in F_r, the longest non-preemptive period is the time it takes to read a flash page. All flash operations, i.e. read, write and erasure, are non-preemptive, but since read flash chips are not performing writes or erasures, the longest flash operation is t_r. The schedulability of the read requests using SRP with EDF is the following [2]:

$$\frac{t_r}{min(T^r)} + \sum_{i=1}^{n} \frac{C_i^r}{T_i^r} \leq 1 \tag{6}$$

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

For write requests that are serviced by flash chips in F_w, the largest non-preemptive period is the longest flash operation that takes place on write flash chips, which is a block erasure. A block reclamation, for example, can be preempted many times between reading and writing valid pages. However, once a flash operation takes place, it cannot be preempted. The feasibility of the real-time write requests can be verified by the following [2]:

$$\frac{t_e}{min(T)} + \sum_{i=1}^{n} (\frac{C_i^w}{T_i^w} + \frac{C_i^e}{T_i^e} + \frac{C_i^g}{T_i^g}) \leq 1 \tag{7}$$

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

At the interface level, the user specifies the number of read and write pages per read and write period upon the open() syscall. If the admission control fails, the open() would fail.

D. Latency Calculation

The admission control in PaRT-FTL guarantees that write requests can always be buffered, so the worst-case write latency is minimal. The worst-case latency for a page read happens when the page needs to be rebuilt. The time it takes to read all the pages needed for the rebuild and the time to decode and rebuild the page is:

$$t_r + t_c^d \tag{8}$$

E. Bandwidth Calculation

We define the write bandwidth as the number of page writes that can be performed in reclaimed blocks across all k data flash chips divided by the time it takes to perform garbage collection and those page writes. After garbage collection is initialized, the block with the largest number of invalid pages is selected as the victim block to be reclaimed. Let B_v be the number of valid pages in the victim block and B_r be the number of invalid or reclaimed pages. To garbage collect the victim block, B_v pages need to be copied to a free block and the victim block is erased. At this point, B_r pages are reclaimed and garbage collection starts again after B_r page writes. The write bandwidth is defined as follows:

$$\frac{k {\cdot} B_r}{(|F|/|F_w|) \cdot [B_v(t_r + t_w) + t_e + B_r {\cdot} t_w]} \tag{9}$$

where F is the set of all flash chips and F_w is the set of write flash chips. The worst-case theoretical bandwidth occurs under the worst case scenario for garbage collection. Since the victim block chosen for garbage collection is the block with the most invalid pages, the worst case is when the invalid pages are evenly spread out among all the blocks. Thus, $B_v = \lceil \lambda {\cdot} P \rceil$ and $B_r = \alpha$.

The maximum read bandwidth occurs when none of the pages need to be rebuilt. Thus, the read bandwidth equals the number of page reads that can be performed in parallel, f_r, divided by the time it takes to perform a page read on every read flash chip, t_r:

$$\frac{f_r}{t_r} \tag{10}$$

The worst-case theoretical read bandwidth occurs when every page needs to be rebuilt:

$$\frac{1}{t_r + t_c^d} \tag{11}$$

where $t_r + t_c^d$ is the time to read a page on every flash chip and the time to decode and rebuild the page.

IV. Related Work

Many previous real-time FTL designs use partial garbage collection. Guarantee Flash Translation Layer (GFTL) [7] first introduced partial garbage collection with block-level mapping. Partial garbage collection reduces the latency experienced by traditional garbage collection by dividing the operation to reclaim one flash block into multiple steps. Because block-level mapping incurs extra OOB operations to get the real mapping information, GFTL is shown to have high worst-case latency [23].

Real-time Flash Translation Layer (RFTL) [16] showed that good performance is possible using a "distributed" partial garbage collection. RFTL uses partial garbage collection, and assumes that the request arrival rate is bounded by the block erasure time. A logical block is mapped to three physical blocks, and partial garbage collection is triggered when the primary physical block is full. The two other physical blocks serve as a buffer for write requests to the corresponding logical block during garbage collection and for copying valid pages from the primary block to allow block erasure. In this way, garbage collection is managed by each logical block in a distributed manner. Similar to GFTL, RFTL also suffers from extra OOB operations [23] as well as low space utilization since each logical block is mapped to three physical blocks.

WAO-GC FTL [23], which stands for worst-case and average-case joint optimization for garbage collection, builds upon the partial garbage collection technique. In addition to providing ideal worst-case bounds for page read and write, WAO-GC is able to achieve better average-case performance than GFTL and RFTL by using over-provisioning to delay garbage collection. When a victim block is selected, it has at most v valid pages. This is guaranteed by the over-provisioning. Let n be the number of partial garbage collection steps needed to copy v pages and erase the victim block. Since a partial garbage collection step is executed after a page write request, n is the number of new pages that need to be stored. Therefore, the following constraint exists: $n + v \leq P$. This constraint guarantees that one free block can hold both v valid pages from the victim block and the n pages from page write requests during the reclamation of the victim block. Let c be the number of page copies that can be done in a partial step. Thus, $n = \lceil \frac{v}{c} \rceil + 1$. Given that $v \leq \lceil \lambda \cdot P \rceil$ due to the over-provisioning, substituting n into the constraint gives the relationship between partial garbage collection and space configuration [23]:

$$\lambda < \frac{(P-1)c}{(c+1)P} \tag{12}$$

The following equations presented will be used in Section VI showing results of WAO-GC and PaRT-FTL implemented on the Cosmos OpenSSD board.

1) Latency Calculation: The worst-case latency for a page write occurs after a partial garbage collection step, which is bounded by t_e. Thus, the worst-case latency for a page write is the following:

$$t_w + max(t_e, c(t_r + t_w)) \tag{13}$$

The worst-case latency for a page read occurs when trying to read a page on a flash die that is busy performing a partial garbage collection step for a write request. Therefore, the worst-case latency for a page read is:

$$t_r + t_w + max(t_e, c(t_r + t_w)) \tag{14}$$

2) Bandwidth Calculation: The write bandwidth is the number of page writes that can be performed in parallel, $|F|$, divided by the time it takes to perform a page write. WAO-GC guarantees that a write request will not be blocked by more than the time it takes to perform a partial garbage collection if the inter-arrival time of write requests does not exceed the partial garbage collection step time. Under this restriction, the maximum theoretical write bandwidth is the following:

$$\frac{|F|}{t_w + max(t_e, c(t_r + t_w))} \tag{15}$$

The read bandwidth is the number of page reads that can be performed in parallel divided by the time it takes to perform a page read given way interleaving. Let f_r be the number of page reads that can be performed in parallel. The read bandwidth is then defined as follows:

$$\frac{f_r}{t_r + t_w + max(t_e, c(t_r + t_w))} \tag{16}$$

Chang et al. [5] proposed a real-time garbage collection mechanism (RTGC) with a real-time task model that schedules read and write periodic tasks with a corresponding real-time garbage collection task. Although it is not a real-time FTL as the scheduler exists on top of the FTL, a standard commercial SSD with a built-in FTL cannot be used because the real-time garbage collectors require that the FTL provides services for block erasure and atomic page copy that can be triggered by the garbage collection tasks. Since read and write requests are not partitioned on separate flash dies, the admission control for RTGC will reject more task sets that have higher read utilization compared to the admission control for PaRT-FTL.

Huang et al. [9] exploited the internal parallelism in SSDs by reserving banks for servicing read and write requests and performing garbage collection to guarantee stable read and write throughput. Partial garbage collection is used, where each step is either a page copy or a block erasure, so each page read is potentially blocked by a partial step.

There have also been many designs using redundancy on flash storage by implementing RAID on multiple commercial SSDs. However, depending on the FTL design in the SSDs, latency can vary greatly when garbage collection is triggered. Purity [8] measures the latency of each request and uses Reed-Solomon to reconstruct the requested data whenever a request takes longer than the 95th percentile latency. Flash on Rails [19] and Shin et al. [18] both partition read and write requests to different SSDs to provide predictable read performance. Our work differs from these previous works in that we implement our technique in the flash translation layer. The advantage of our approach is that we are able to precisely control each flash die. When the solution is built on top of an existing FTL, there is no way of knowing if a flash die is busy doing garbage collection when switching an SSD from writing to reading. Therefore, read latency cannot be guaranteed.

Tiny-Tail Flash [22] partitions flash planes into two sets, one for garbage collection and one for servicing read and write requests. The garbage collection planes rotate periodically and parity-based redundancy is used to rebuild reads. The configuration of the partitioning differs in our design as we separate read and write requests onto different flash dies. Whereas simulation results show that Tiny-Tail Flash reduces GC-blocked I/Os to 0.003-0.7%, our real-time model guarantees that deadlines will not be missed.

V. Implementation

The following section outlines implementation details for PaRT-FTL and the over-provisioning used. Each read or write request is split up into flash page-size requests by the device driver and then inserted into a request circular buffer. The FTL retrieves the requests and orders them according to Earliest Deadline First and starts handling the request if the flash die is not busy.

Write requests are buffered and admission control guarantees that the buffer will not overflow. Write dies flush pages in the buffer to the SSD. If garbage collection is initialized, one flash operation for the garbage collector will be done on that die, either reading a valid page, writing a valid page, or erasing a block. If there is a read request for a page on a write flash die, a rebuild operation will be initialized by adding the pages that need to be read for the rebuild to the front of the queue. In the beginning of the request loop, page rebuilds are checked. If all the pages for a rebuild are read, the requested page is decoded.

A. Over-provisioning

For PaRT-FTL, 25% of storage is reserved for parity checking, which is typical for a RAID design. Of the 75% used for data, the ratio of logical to physical address space is set to 64.3%. Thus, 48.2% of the SSD is used for data, with 26.8% over-provisioning and 25% for parity information.

When comparing PaRT-FTL against other approaches such as RTGC and WAO-GC, we use a matching data storage capacity. Thus, in RTGC and WAO-GC, 48.2% of the SSD is used for data with 51.8% over-provisioning. To ensure that the over-provisioning for WAO-GC is enough, the upper bound of λ is calculated. We define t_r as the time it takes to read a flash page assuming other flash dies on the same way are busy and not idle. In our hardware, a page is 8 KB. The way arbiter in the Cosmos Board grants access in a round-robin manner to the common flash bus to access the NAND flash or to use the internal data bus to access the page buffer, which stores 2 KB of data. Since a flash page is 8 KB, data transfer between the page buffer occurs four times for each flash page. This means that if we are measuring the latency of a page read on a die, we have to assume that if the other dies on the same way are also reading, $t_r = 1.23$ ms on average for that page read based on our observations in Table I. Similarly, a page write takes 1.87 ms and a block erasure takes 3.84 ms, on average. Let one partial garbage step consist of two page copies and $P = 256$ pages per flash block, the upper bound on the ratio of logical to physical address space for WAO-GC can be calculated using equation 12 as 66.4%.

VI. Evaluation

The experimental evaluation consists of two sections: 1) simulation-based schedulability tests, and 2) experiments conducted using two different FTL implementations on the Cosmos OpenSSD board. The simulations show that PaRT-FTL has a higher feasible utilization, while the OpenSSD experiments show that PaRT-FTL has lower read and write latency.

A. Simulation Experiments

Random task sets were generated with varying total utilization using the UUnifast algorithm [4]. 500 task sets were generated for each utilization value ranging from 0.05 to 0.95 with 0.05 increments. Each task set contains 10 tasks. Each task makes read requests of one flash page, which equals 8 KB, and write requests of 3 flash pages. The periods for each write request and read request are calculated as follows:

$$T_i^w = \frac{\lceil w_i/|F|\rceil \cdot t_w}{U_i^w} \tag{17}$$

$$T_i^r = \frac{r_i \cdot t_r}{U_i^r} \tag{18}$$

where U_i^w and U_i^r are the write and read utilizations generated, respectively.

Each task set was tested to see if it was schedulable under PaRT-FTL, RTGC and WAO-GC FTL. Figure 5 shows the simulation results for admission control with PaRT-FTL, which is calculated with Equations 6 and 7, and the admission control with RTGC [5]. The WAO-GC FTL has no admission control. Each flash operation could potentially be blocked by a partial garbage collection step from a different task. In the worst case, a task will be blocked by all the other tasks. In our experiments, we used a uniform probability to determine the interference from other tasks. We also show a percentage of the likelihood that a write operation would trigger a partial garbage collection step. When λ is set to the maximum value in Equation 12, partial garbage collection will occur after every write operation. However, when λ is less, some write operations will not trigger a partial garbage collection step. We use a percentage to show the effects of higher over-provisioning (WAO-GC 75 and WAO-GC 50). For example, in our implementation, $\lambda = 48.2\%$, which is 72.5% of the maximum value for WAO-GC, so schedulable tasksets would be close to WAO-GC 75.

We also varied the size of the read and write requests to identify their effects on schedulability, shown in Figures 6 and 7. We measured the weighted schedulability [3], which

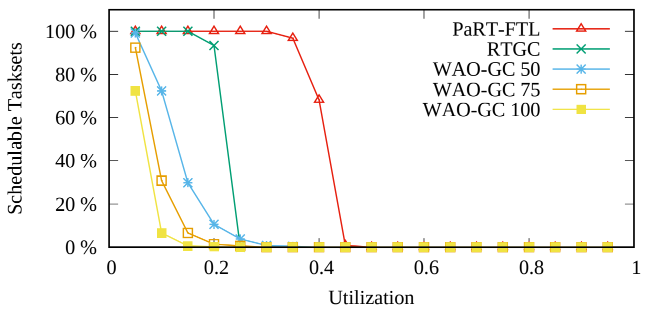


Fig. 5: Admission Control Simulation.

is the sum of all the total utilizations of task sets that were schedulable divided by the sum of all the total utilizations. The weighted schedulability compresses a three-dimensional plot to two dimensions and places higher value on task sets with higher utilization.

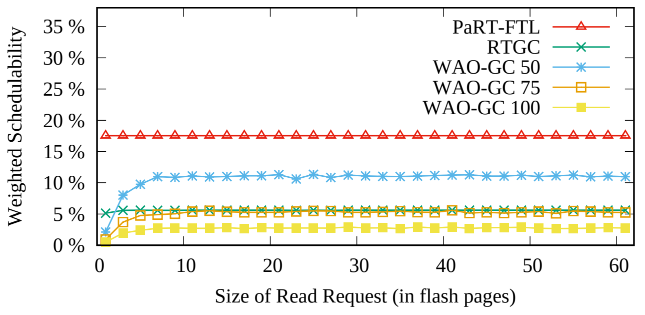


Fig. 6: Weighted Schedulability vs Read Request Size.

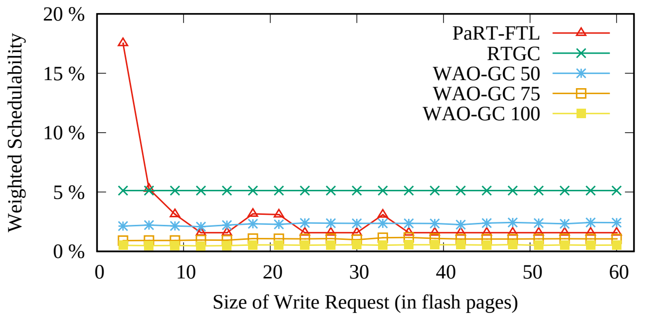


Fig. 7: Weighted Schedulability vs Write Request Size.

For various read request sizes, PaRT-FTL consistently shows higher schedulability than RTGC and WAO-GC. When varying the write request sizes, PaRT-FTL has lower schedulability than RTGC in general and lower schedulability than WAO-GC 50 when $w_i > 10$ as seen in Figure 7. This is due to PaRT-FTL's lower write bandwidth as writes are only occurring on 4 out of the 16 flash dies. Also, note that WAO-GC 50 has much lower space utilization compared to our configuration for PaRT-FTL. The peaks at write size equaling 18, 21 and 33 in Figure 7 are due to how the task sets are generated. For example, since there are 16 flash chips in our NAND flash memory, when $w_i = 18$, the write period generated is twice as large as the write period generated when $w_i = 15$, thus increasing schedulability.

B. Hardware Experiments

The OpenSSD Cosmos board [20] (as described in Figures 8 and 9) is used to implement the FTL. It 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 [21].

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 PS 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 the ARM Cortex-A9.

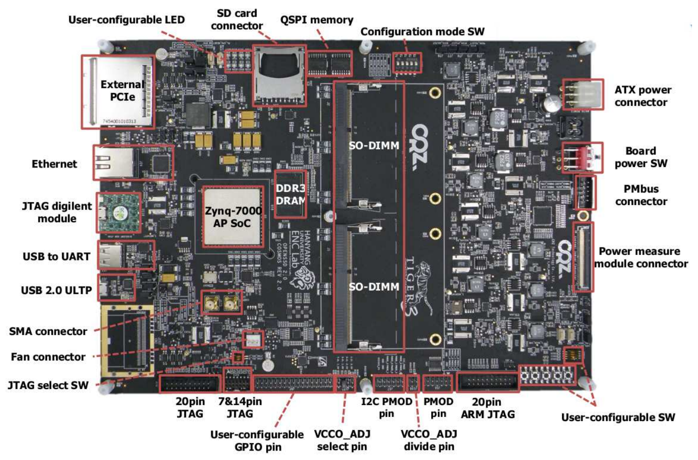
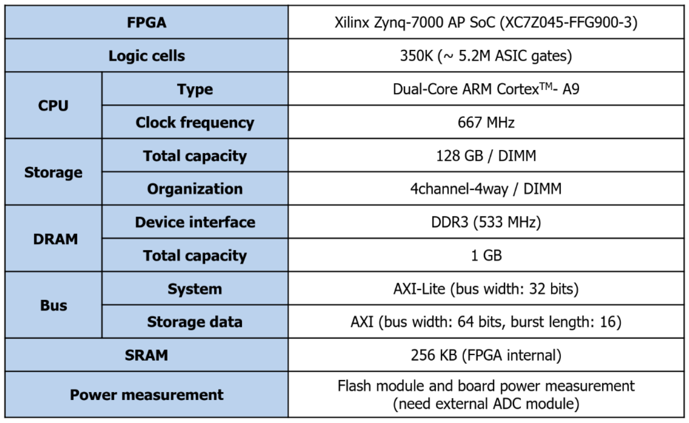


Fig. 8: OpenSSD Cosmos board [20].

The OpenSSD Cosmos board has two small outline dual in-line memory modules (SO-DIMMs), each containing Micron Technology's MLC NAND [1] flash. 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 error correction code (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 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 are transferred to the ECC decoder. If there are errors in the

[1]Part number MT29F256G08CMCABH2.



FPGA		Xilinx Zynq-7000 AP SoC (XC7Z045-FFG900-3)
Logic cells		350K (~ 5.2M ASIC gates)
CPU	Type	Dual-Core ARM Cortex™- A9
	Clock frequency	667 MHz
Storage	Total capacity	128 GB / DIMM
	Organization	4channel-4way / DIMM
DRAM	Device interface	DDR3 (533 MHz)
	Total capacity	1 GB
Bus	System	AXI-Lite (bus width: 32 bits)
	Storage data	AXI (bus width: 64 bits, burst length: 16)
SRAM		256 KB (FPGA internal)
Power measurement		Flash module and board power measurement (need external ADC module)

Fig. 9: OpenSSD Cosmos board specifications [20].

data, the ECC decoder corrects the data and transfers the data to the page buffer. Data is then transferred to DRAM.

1) PaRT-FTL: The experimental setup is as follows in Table III, with both random accesses and writes. Note that in the following experiment, a task either reads or writes. The SSD is initially only written to so that the effects of garbage collection on write latency can be observed.

	Tasks	Request Size	Period
Write	4	12 pages	60 ms
Read	4	3 pages	15 ms

TABLE III: Experimental setup.

The XOR implementation using NEON instructions takes 1.2 ms. We verified that the data is correct with the encoding and decoding functions to rebuild read pages. The following experiments, however, do not include the overhead to compute the XORs. We assume that in a production-ready system, computing the XORs will be built into the hardware. While we assume this cost to be negligible in our experiments, it is accounted for in the task model in Equations 1 and 3.

The PaRT-FTL maximum write bandwidth is 7.3 MB/s (Eq. 9) with $k = 12$, $|F_w| = 4$ and parameters in Table IV. The maximum read bandwidth is 76 MB/s (Eq. 10).

The write bandwidth for 4 tasks each making 12-page write requests is plotted in Figure 10a. The read bandwidth for 4 tasks each making 12-page write requests and 4 tasks each making 3-page read requests every 15 milliseconds is plotted in Figure 12a. Note that the first few points are below average because the tasks are initialized in the middle of the time slice.

The response times of write and read requests are measured and plotted in Figure 10b and Figure 12b, respectively. This is the time it took to complete a 12-page write request or a 3-page read request. The latency of a single page write and a single page read is also measured and plotted in Figure 10c and Figure 12c, respectively. The device driver inserts single-page requests into the request buffer. Latency is measured in the FTL as the time from when a page read or write is put into the request buffer to when the FTL marks that request as completed. The task set passes PaRT-FTL's admission control, so no deadlines are missed.

2) WAO-GC FTL: The same experimental setup shown in Table III is run with WAO-GC FTL. The WAO-GC maximum write bandwidth is 15.5 MB/s (Eq. 15) and read bandwidth is 13.4 MB/s (Eq. 16) with parameters in Table IV.

	PaRT-FTL F_w	PaRT-FTL F_r	WAO-GC
$\|F\|$	16	16	16
f_r	-	12	16
t_r	0.375 msec	1.23 msec	1.23 msec
t_w	1.33 msec	-	1.87 msec
t_e	3.84 msec	-	3.84 msec

TABLE IV: Bandwidth and Latency parameters.

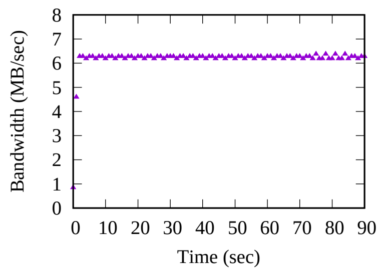
As in PaRT-FTL, the response times of write and read requests are measured and plotted in Figure 11b and Figure 13b, respectively. The latency of a single page write and a single page read under WAO-GC FTL is also measured and plotted in Figure 11c and Figure 13c, respectively. As expected from the high read latency, some read requests using WAO-GC miss deadlines, as shown in Figure 13b by the data above the 15 millisecond horizontal line.

C. Discussion

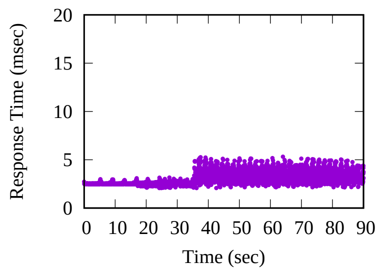
PaRT-FTL significantly reduces read and write latencies compared to previous real-time FTL approaches that use partial garbage collection (Figures 14 and 15). This is because read requests are never blocked by a busy flash die that is servicing a write request and potentially performing garbage collection. As shown in Figure 15, the maximum write latency with PaRT-FTL is 20% of the maximum write latency with WAO-GC FTL. The maximum read latency with PaRT-FTL is 35% of the maximum read latency with WAO-GC FTL. We did not implement RTGC, however, as the worst-case latency of RTGC has been measured and compared in previous work [23].

PaRT-FTL does sacrifice being able to write at a higher bandwidth since it partitions flash dies into read and write dies. Whereas WAO-GC FTL could write in parallel to all 16 flash dies, PaRT-FTL could only write to 4 dies in parallel since the other dies are servicing read requests. WAO-GC guarantees that a request will not be blocked by more than a partial garbage collection step given that the inter-arrival time of write requests does not exceed the time it takes to do a partial garbage collection step. However, there is no admission control, so the effects of garbage collection can be seen in Figure 16. The workload consists of two streams, each sending requests of 24 pages as fast as possible. Initially, writes occur at 27 MB/s. After 7 seconds, garbage collection starts and bandwidth fluctuates down to 14 MB/s.

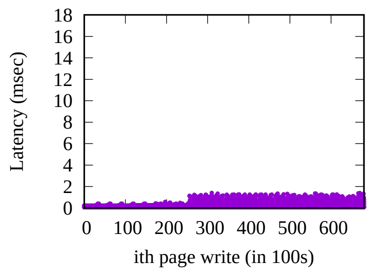
PaRT-FTL is suited to time-critical systems that require low latency guarantees, whereas other approaches may be more suitable for tasks that need high bandwidth and can tolerate some missed deadlines. Our experiments are also limited by our hardware. Modern SSDs have much higher bandwidth and more parallelism such as more channels, ways per channel, and flash dies per flash chip. All these features would improve the bandwidth in PaRT-FTL.



(a) Write bandwidth

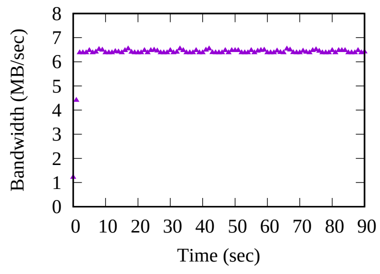


(b) Time for each 12-page write request

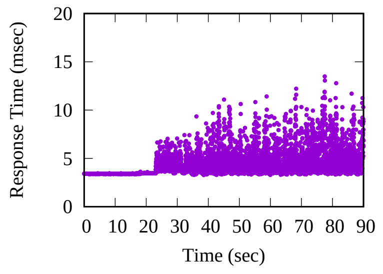


(c) Latency for each page write

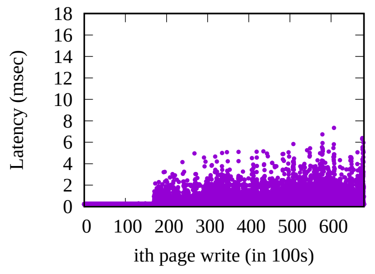
Fig. 10: Write bandwidth, request response time, and page latency with PaRT-FTL.



(a) Write bandwidth

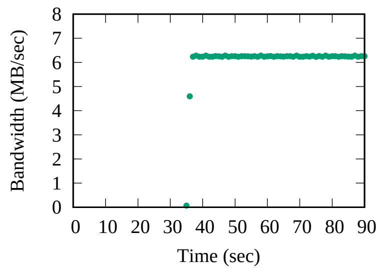


(b) time for each 12-page write request

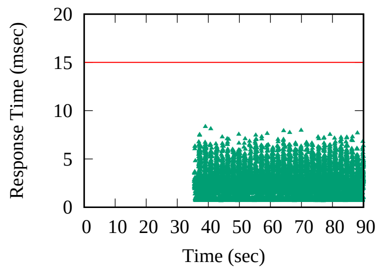


(c) Latency for each page write

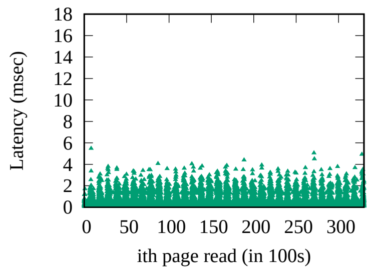
Fig. 11: Write bandwidth, request response time, and page latency with WAO-GC FTL.



(a) Read bandwidth

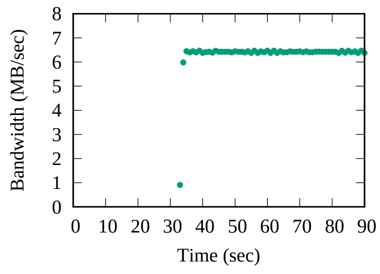


(b) Time for each 3-page read request

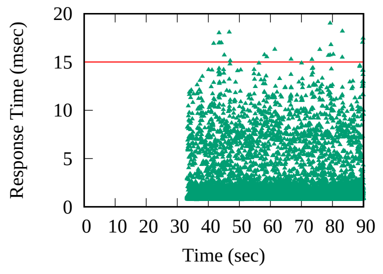


(c) Latency for each page read

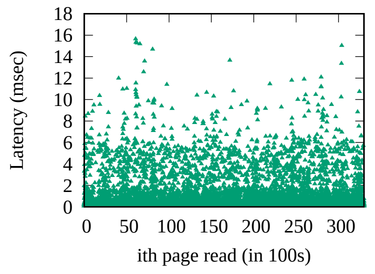
Fig. 12: Read bandwidth, request response time, and page latency with PaRT-FTL.



(a) Read bandwidth



(b) Time for each 3-page read request



(c) Latency for each page read

Fig. 13: Read bandwidth, request response time, and page latency with WAO-GC FTL.

VII. Conclusions and Future Work

PaRT-FTL is motivated by the emerging need for bounded and low latency access to solid state storage in time-critical systems. We present a flash translation layer design that partitions read and write requests onto different flash chips, based on the parallelism in the SSD. We demonstrate the performance of PaRT-FTL by comparison to previous work in real-time FTL design. Empirical results show that we are able to significantly reduce read and write latency.

Future work includes improving the bandwidth of PaRT-FTL by exploring different hardware configurations. We are

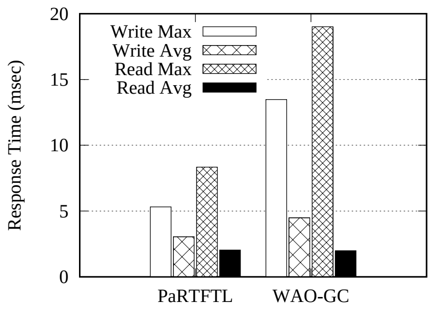


Fig. 14: Response times of PaRT-FTL and WAO-GC FTL.

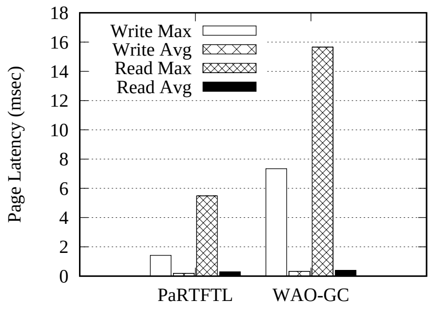


Fig. 15: Flash page latencies of PaRT-FTL and WAO-GC FTL.

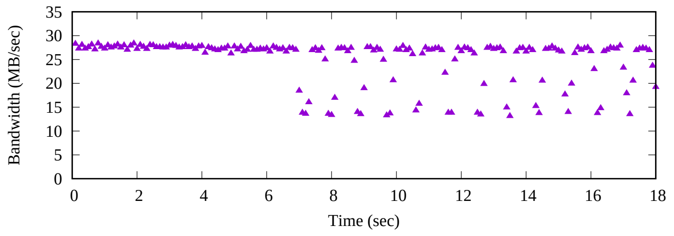


Fig. 16: The effects of garbage collection on write bandwidth with WAO-GC. Garbage collection is initialized after 7 seconds.

also interested in adding fault tolerance to PaRT-FTL. Each flash page contains error correction codes, and when blocks of flash cells are deemed unreliable for further use, the SSD controller discards them to avoid the risk of encountering an uncorrectable error [12]. With PaRT-FTL, a page can be reconstructed, providing further fault tolerance to errors that may occur as flash cells degrade over time.

VIII. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1527050. 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.

References

[1] A. D. Angelica, "Google's Self-driving Car Gathers Nearly 1 GB/sec," http://www.kurzweilai.net/googles-self-driving-car-gathers-nearly-1-gbsec, May 2013.

[2] T. P. Baker, "Stack-based Scheduling of Real-time Processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–100, 1991.

[3] A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability," in *Proceedings of the Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '10)*, 2010.

[4] E. Bini and G. C. Buttazzo, "Measuring the Performance of Schedulability Tests," *Journal of Real-Time Systems*, vol. 30, no. 1-2, 2005.

[5] L.-P. Chang, T.-W. Kuo, and S.-W. Lo, "Real-time Garbage Collection for Flash-memory Storage Systems of Real-time Embedded Systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 4, pp. 837–863, 2004.

[6] F. Chen, B. Hou, and R. Lee, "Internal Parallelism of Flash Memory-based Solid-state Drives," *ACM Transactions on Storage*, vol. 12, no. 3, 2016.

[7] S. Choudhuri and T. Givargis, "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*, 2008.

[8] J. Colgrove, J. D. Davis, J. Hayes, E. L. Miller, C. Sandvig, R. Sears, A. Tamches, N. Vachharajani, and F. Wang, "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)*, 2015.

[9] S.-M. Huang and L.-P. Chang, "Providing SLO Compliance on NVMe SSDs Through Parallelism Reservation," *ACM Transactions on Design Automation of Eletronic Systems*, vol. 23, no. 3, 2018.

[10] S. Kim, H. Oh, C. Park, S. Cho, and S.-W. Lee, "Fast, Energy Efficient Scan Inside Flash Memory SSDs," in *Proceedings of the International Workshop on Accelerating Data Management Systems*, 2011.

[11] LITE-ON, "Best-in Class SSD Solutions with High-Reliability, Manageability, and Durability," http://www.liteonssd.com/en/datasheet/J8/Lite-On\%20Automotive.pdf, 2018.

[12] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "A Large-scale Study of Flash Memory Failures in the Field," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '15)*, 2015.

[13] Micron, "Micron Technical Report: Small-block vs. Large-block NAND Flash Devices," Tech. Rep. TN-29-07, 2005.

[14] ——, "NAND Flash Die – 128Gb Die: x8 300mm MLC MT29F128G08CBECB," February 2018.

[15] H. Oh, "Single Controller 4/8TB SSD," 2013, Flash Memory Summit Conference.

[16] Z. Qin, Y. Wang, D. Liu, and Z. Shao, "Real-time Flash Translation Layer for NAND Flash Memory Storage Systems," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2012.

[17] L. Sha, T. Abdelzaher, K.-E. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok, "Real Time Scheduling Theory: A Historical Perspective," *Real-Time Systems*, vol. 28, no. 2/3, pp. 101–155, 2004.

[18] W. Shin, M. Kim, K. Kim, and H. Y. Yeom, "Providing QoS through Host Controlled Flash SSD Garbage Collection and Multiple SSDs," in *Proceedings of the International Conference on Big Data and Smart Computing (BIGCOMP '15)*, 2015.

[19] D. Skourtis, D. Achlioptas, N. Watkins, C. Maltzahn, and S. Brandt, "Flash on Rails: Consistent Flash Performance through Redundancy," in *Proceedings of the USENIX Annual Technical Conference*, 2014.

[20] Y. H. Song, S. Jung, S.-W. Lee, and J.-S. Kim, "Cosmos OpenSSD: A PCIe-based Open Source SSD Platform," 2014, Flash Memory Summit Conference.

[21] R. West, Y. Li, E. Missimer, and M. Danish, "A Virtualized Separation Kernel for Mixed Criticality Systems," *ACM Transactions on Computer Systems*, vol. 34, 2016.

[22] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '17)*, 2017.

[23] Q. Zhang, X. Li, L. Wang, T. Zhang, Y. Wang, and Z. Shao, "Optimizing Deterministic Garbage Collection in NAND Flash Storage Systems," in *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS '15)*. IEEE, 2015.