Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs

SHIQIN YAN, HUAICHENG LI, MINGZHE HAO, and MICHAEL HAO TONG,
University of Chicago
SWAMINATHAN SUNDARARAMAN, Parallel Machines
ANDREW A. CHIEN and HARYADI S. GUNAWI, University of Chicago

Flash storage has become the mainstream destination for storage users. However, SSDs do not always deliver the performance that users expect. The core culprit of flash performance instability is the well-known garbage collection (GC) process, which causes long delays as the SSD cannot serve (blocks) incoming I/Os, which then induces the long tail latency problem. We present ttFlash as a solution to this problem. ttFlash is a "tiny-tail" flash drive (SSD) that eliminates GC-induced tail latencies by circumventing GC-blocked I/Os with four novel strategies: plane-blocking GC, rotating GC, GC-tolerant read, and GC-tolerant flush. These four strategies leverage the timely combination of modern SSD internal technologies such as powerful controllers, parity-based redundancies, and capacitor-backed RAM. Our strategies are dependent on the use of intra-plane copyback operations. Through an extensive evaluation, we show that ttFlash comes significantly close to a "no-GC" scenario. Specifically, between the 99 and 99.99th percentiles, ttFlash is only 1.0 to 2.6× slower than the no-GC case, while a base approach suffers from 5–138× GC-induced slowdowns.

CCS Concepts: • Hardware → External storage; • Software and its engineering → File systems management;

Additional Key Words and Phrases: ttFlash, flash-based SSD

ACM Reference format:
https://doi.org/10.1145/3121133

1 INTRODUCTION

Flash storage has become the mainstream destination for storage users. The solid-state drive (SSD) consumer market continues to grow at a significant rate [9], SSD-backed cloud virtual machine instances are becoming the norm [8, 14], and flash/SSD arrays are a popular solution for high-end storage servers [24, 31, 49]. From the users’ side, they demand fast and stable latencies [26, 29]. However, SSDs do not always deliver the performance that users expect [16]. Some even suggest
that flash storage “may not save the world” (due to the tail latency problem) [6]. Some recent works dissect why it is hard to meet service level agreement (SLA) with SSDs [40] and reveal high performance variability in 7 million hours of SSD deployments in customer sites [31].

The core problem of flash performance instability is the well-known and “notorious” garbage collection (GC) process. A GC operation causes long delays as the SSD cannot serve (blocks) incoming I/Os. Due to an ongoing GC, read latency variance can increase by 100× [6, 25]. Since the mid-2000s, there has been a large body of work that reduces the number of GC operations with a variety of novel techniques [30, 39–41, 43, 48, 55]. However, we find almost no work in the literature that attempts to eliminate the blocking nature of GC operations and deliver steady and stable SSD performance in the long runs.

We address this urgent issue with the “tiny-tail” flash drive (ttFlash), a GC-tolerant SSD that can deliver and guarantee stable performance. The goal of ttFlash is to eliminate GC-induced tail latencies by circumventing GC-blocked I/Os. That is, ideally there should be no I/O that will be blocked by a GC operation, thus creating a flash storage that behaves close to a “no-GC” scenario. The key enabler is that SSD internal technology has changed in many ways, which we exploit to build novel GC-tolerant approaches.

Specifically, there are three major SSD technological advancements that we leverage for building ttFlash. First, we leverage the increasing power and speed of today’s flash controllers that enable more complex logic (e.g., multi-threading, I/O concurrency, fine-grained I/O management) to be implemented at the controller. Second, we exploit the use of Redundant Array of Independent NAND (RAIN). Bit error rates of modern SSDs have increased to the point that error-correcting code (ECC) is no longer deemed sufficient [36, 41, 51]. Due to this increasing failure, modern commercial SSDs employ parity-based redundancies (RAIN) as a standard data protection mechanism [7, 13]. By using RAIN, we can circumvent GC-blocked read I/Os with parity regeneration. Finally, modern SSDs come with a large RAM buffer (hundreds of MBs) backed by “super capacitors” [11, 15], which we leverage to mask write tail latencies from GC operations.

The timely combination of the technology practices above enables four new strategies in ttFlash: (a) plane-blocking GC, which shifts GC blocking from coarse granularities (controller/channel) to a finer granularity (plane level), which depends on intra-plane copyback operations; (b) GC-tolerant read, which exploits RAIN parity-based redundancy to proactively generate the contents of read I/Os that are blocked by ongoing GCs; (c) rotating GC, which schedules GC in a rotating fashion to enforce at most one active GC in every plane group, hence ttFlash can be guaranteed to “cut” the tail latency of one GC operation per plane group; and, finally, (d) GC-tolerant flush, which evicts buffered writes from capacitor-backed RAM to flash pages, free from GC blocking.

One constraint of ttFlash is its dependency on intra-plane copybacks where GC-ed pages move within a plane without the data flowing through the SSD controller, hence skipping ECC checks for garbage collected pages, which may reduce data reliability. The full extent of this effect is not evaluated here and is left for future work. We recommend ECC checks to be performed in the background to overcome this limitation (Section 7).

We first implemented ttFlash in SSDSim [33] to simulate accurate latency analysis at the device level. Next, to run real file systems and applications, we also ported ttFlash to a newer QEMU/KVM-based platform based on VSSIM [57].

With a thorough evaluation (Section 6.1), we show that ttFlash successfully eliminates GC blocking for a significant number of I/Os, reducing GC-blocked I/Os from 2–7% (base case) to only 0.003–0.7%. As a result, ttFlash reduces tail latencies dramatically. Specifically, between the 99 and 99.99th percentiles, compared to the perfect no-GC scenario, a base approach suffers from 5.6 to 138.2× GC-induced slowdowns. ttFlash on the other hand is only 1.0–2.6× slower than
the no-GC case, which confirms that ttFlash almost delivers a near-complete elimination of GC blocking and the resulting “tiny” latency tail.

We also show that ttFlash is more stable than state-of-the-art approaches that reduce GC impacts such as preemptive GC [10, 44] (Section 6.2). Specifically, ttFlash continuously delivers stable latencies while preemptive GC exhibits latency spikes under intensive I/Os. Furthermore, we contrast the fundamental difference of GC-impact elimination from reduction (Section 6.3, Section 8).

In summary, by leveraging modern SSD internal technologies in a unique way, we have successfully built novel features that provide a robust solution to the critical problem of GC-induced tail latencies. In the following sections, we present extended motivation (Section 2), an SSD primer (Section 3), the ttFlash design (Section 4), implementation (Section 5), evaluation (Section 6), and limitations (Section 7), related work (Section 7), conclusion (9), and a proof sketch (Appendix).

2 EXTENDED MOTIVATION: GC-INDUCED TAIL LATENCY

We present two experiments that show GC’s cascading impacts and that motivate our work. Each experiment runs on a late-2014 128GB Samsung SM951, which can sustain 70 “KWPS” (70K of 4KB random writes/s).

In Figure 1(a), we ran a foreground thread that executes 16KB random reads concurrently with background threads that inject 4KB random-write noises at 1, 2.5, and 5KWPS (far below the max 70KWPS) across three experiments. We measure $L_i$, the latency of every 16KB foreground read. Figure 1(a) plots the CDF of $L_i$, clearly showing that more frequent GCs (from more-intense random writes) block incoming reads and create longer tail latencies. To show the tail is induced by GC, not queueing delays, we ran the same experiments but now with random-read noises (1, 2.5, and 5K RPS). The read-noise results are plotted as the three overlapping thin lines marked “ReadNoise,” which represent a perfect no-GC scenario. As shown, with 5 KWPS noise, read operations become $15 \times$, $19 \times$, and $96 \times$ slower compared to no-GC scenarios, at the 90th, 95th, and 99th percentiles, respectively.

In Figure 1(b), we keep the 5-KWPS noise and now vary the I/O size of the foreground random reads (8, 16, 32, 64, and 128KB across five experiments). Supposedly, a 2× larger read should only consume 2× longer latency. However, the figure shows that GC induces more tail latencies in larger reads. For example, at the 85th percentile, a 64KB read is 4× slower than a 32KB read. The core of the problem is this: If a single page of a large read is blocked by a GC, then the entire read cannot complete; as read size increases, the probability of one of the pages being blocked by GC also
Table 1. SSD Parameters.

<table>
<thead>
<tr>
<th>Sizes</th>
<th>Latencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSD Capacity</td>
<td>256GB</td>
</tr>
<tr>
<td>#Channels</td>
<td>8</td>
</tr>
<tr>
<td>#Planes/channel</td>
<td>8</td>
</tr>
<tr>
<td>Plane size</td>
<td>4GB</td>
</tr>
<tr>
<td>#Planes/chip</td>
<td><strong>1</strong></td>
</tr>
<tr>
<td>#Blocks/plane</td>
<td>4096</td>
</tr>
<tr>
<td>#Pages/block</td>
<td>256</td>
</tr>
<tr>
<td>Page size</td>
<td>4KB</td>
</tr>
</tbody>
</table>

Page Read 40μs (flash-to-register)
Page Write 800μs (register-to-flash)
Page data transfer 100μs (via channel)
Block erase 2ms

This article uses the above parameters. **One plane per chip is for simplicity of presentation and illustration. The latencies are based on average values; actual latencies can vary due to Read retry, different voltages, and so on. Flash reads and writes must use the plane register.**

increases, as we explain later (Section 3, Section 4.1). The pattern is more obvious when compared to the same experiments but with 5-KRPS noises (the five thin gray lines marked “ReadNoise”).

For a fairer experiment, because flash read latency is typically 20× faster than write latency, we also ran read noises that are 20× more intense and another where read noises are 20× larger in size. The results are similar.

3 SSD PRIMER: GC BLOCKING

Before presenting TtFlash, we first need to describe SSD internals essential for understanding GC blocking. This section describes how GC operates from the view of the physical hardware.

**SSD Layout:** Figure 2 shows a basic SSD internal layout. Data and command transfers are sent via parallel channels ($C_1$-$C_N$). A channel connects multiple flash planes; one to four planes can be packaged as a single chip (dashed box). A plane contains blocks of flash pages. In every plane, there is a 4KB register; all flash reads/writes must transfer through the plane register. The controller is connected to a capacitor-backed RAM used for multiple purposes (e.g., write buffering). For clarity, we use concrete parameter values shown in Table 1.

**GC operation (four main steps):** When the used-page count increases above a certain threshold (e.g., 70%), a garbage collection process (GC) will start. A possible GC operation reads valid pages from an old block, writes them to a free block, and erases the old block, within the same plane. Figure 2 shows two copybacks in a GC-ing plane (two valid pages being copied to a free block). Most importantly, with 4KB register support in every plane, page copybacks happen within the GC-ing plane without using the channel [12, 19].
Tiny-Tail Flash: Near-Perfect Elimination of Garbage

Fig. 3. Various levels of GC blocking. Shaded I/Os with bold edges I/Os in bright planes are servable while non-shaded I/Os in dark planes are blocked. (a) In controller-blocking (Section 3), a GC blocks the controller/entire SSD. (b) In channel-blocking (Section 3), a GC blocks the channel connected to the GC-ing plane. (c) In plane-blocking (Section 4.1), a GC only blocks the GC-ing plane.

The controller then performs the following for-loop of four steps for every page copyback: (1) send a flash-to-register read command through the channel (only 0.2μs) to the GC-ing plane, (2) wait until the plane executes the 1-page read command (40μs without using the channel), (3) send a register-to-flash write command, and (4) wait until the plane executes the 1-page write command (800μs without using the channel). Steps (1)–(4) are repeated until all valid pages are copied and then the old block is erased. The key point here is that copyback operations (steps (2) and (4); roughly 840μs) are done internally within the GC-ing plane without crossing the channel.

**GC Blocking:** GC blocking occurs when some resources (e.g., controller, channel, planes) are used by a GC activity, which will delay subsequent requests, similar to head-of-line blocking. Blocking designs are used as they are simple and cheap (small gate counts). But because GC latencies are long, blocking designs can produce significant tail latencies.

One simple approach to implementing GC is with a blocking controller. That is, even when only one plane is performing GC, the controller is busy communicating with the GC-ing plane and unable to serve outstanding I/Os that are designated to any other planes. We refer to this as *controller-blocking* GC, as illustrated in Figure 3(a). Here, a single GC (the striped plane) blocks the controller, thus all channels and planes are blocked (the bold lines and dark planes). All outstanding I/Os cannot be served (represented by the non-shaded I/Os). OpenSSD [4], VSSIM [57], and low-cost systems such as eMMC devices adopt this implementation.

Another approach is to support multi-threaded/multi-CPU with channel queuing. Here, while a thread/CPU is communicating to a GC-ing plane (in a for-loop) and blocking the plane’s channel (e.g., bold line in Figure 3), other threads/CPUs can serve other I/Os designated to other channels (the shaded I/Os with bold edges I/Os in bright planes). We refer this as *channel-blocking* GC (i.e., a GC blocks the channel of the GC-ing plane). SSDSim [33] and disksim+SSD [19] adopt this implementation. Commodity SSDs do not come with layout specifications, but from our experiments (Section 2), we suspect some form of channel-blocking (at least in client SSDs) exists.

Figure 1 also implicitly shows how blocked I/Os create cascading queuing delays. Imagine that the “Outstanding I/Os” represents a full device queue (e.g., typically 32 I/Os). When this happens, the host OS cannot submit more I/Os; hence user I/Os are blocked in the OS queues. We show this impact in our evaluation.

4 TTFLASH DESIGN

We now present the design of ttFlash, a new SSD architecture that achieves guaranteed performance close to a no-GC scenario. We are able to remove GC blocking at all levels with the following four key strategies:

Devise a non-blocking controller and channel protocol, pushing any resource blocking to only the affected planes. We call this fine-grained architecture plane-blocking GC (Section 4.1).

(2) Exploit RAIN-based redundancy (Section 4.2) and combine it with GC information to proactively regenerate reads blocked by GC at the plane level, which we name GC-tolerant read (Section 4.3).

(3) Schedule GC in a rotating fashion to enforce at most one GC in every plane group, such that no reads will see more than one GC; one parity can only "cut" one tail. We name this rotating GC (Section 4.4).

(4) Use a capacitor-backed write buffer to deliver fast durable completion of writes, allowing them to be evicted to flash pages at a later time in a GC-tolerant manner. We name this GC-tolerant flush (Section 4.5).

4.1 Plane-Blocking GC (PB)

Controller- and channel-blocking GC are often adopted due to their simplicity of hardware implementation; a GC is essentially a for-loop of copyback commands. This simplicity, however, leads to severe tail latencies as independent planes are unnecessarily blocked. Channel-blocking is no better than controller-blocking GC for large I/Os; as every large I/O is typically striped across multiple channels, one GC-busy channel still blocks the entire I/O, negating the benefit of SSD parallelism. Furthermore, as SSD capacity increases, there will be more planes blocked in the same channel. Worse, the GC period can be significantly long. A GC that copies 64 valid pages back (25% valid) will lead to 54 ms (64 × 840 μs) of blocked channel, which potentially leaves hundreds of other I/Os unservable. An outstanding read operation that supposedly only takes less than 100 μs is now delayed by order(s) of magnitude [6, 25].

To reduce this unnecessary blocking, we introduce plane-blocking GC, as illustrated in Figure 3(c). Here, the only outstanding I/Os blocked by a GC are the ones that correspond to the GC-ing plane (⃝ labels). All I/Os to non-GCing planes (non-⃝ labels) are servable, including the ones in the same channel of the GC-ing plane. As a side note, plane-blocking GC can be interchangeably defined as chip-blocking GC; in this article, we use 1 plane/chip for simplicity of presentation.

To implement this concept, the controller must perform a fine-grained I/O management. For illustration, let us consider the four GC steps (Section 3). In TtFlash, after a controller CPU/thread sends the flash-to-register read/write command (Steps 1 and 3), it will not be idle waiting (for 40 μs and 800 μs, respectively) until the next step is executable. (Note that in a common implementation, the controller is idling due to the simple for-loop and the need to access the channel to check the plane’s copyback status). With plane-blocking GC, after Steps 1 and 3 (send read/write commands), the controller creates a future event that marks the completion time. The controller can reliably predict how long the intra-plane read/write commands will finish (e.g., 40 and 800 μs on average, respectively). To summarize, with plane-blocking GC, TtFlash overlaps intra-plane copyback and channel usage for other outstanding I/Os. As shown in Figure 3(c), for the duration of an intra-plane copyback (the striped/GC-ing plane), the controller can continue serving I/Os to other non-GCing planes in the corresponding channel (▲ I/Os).

Plane-blocking GC potentially frees up hundreds of previously blocked I/Os. However, there is an unsolved GC blocking issue and a new ramification. The unsolved GC blocking issue is that the I/Os to the GC-ing plane (⃝ labels in Figure 3(c)) are still blocked until the GC completes; in other words, with only plane-blocking, we cannot entirely remove GC blocking. The new ramification of plane-blocking is a potentially prolonged GC operation; when the GC-ing plane is ready to take another command (end of Steps 2 and 4), the controller/channel might still be in the middle of serving other I/Os, due to overlaps. For example, the controller cannot start the GC write (Step 3)
Tiny-Tail Flash: Near-Perfect Elimination of Garbage

4.2 RAIN

To prevent blocking of I/Os to GC-ing planes, we leverage RAIN, a recently popular standard for data integrity [7, 13]. RAIN introduces the notion of parity pages inside the SSD. Just like the evolution of disk-based RAIDs, many RAIN layouts have been introduced [36, 41, 45, 46], but they mainly focus on data protection, write optimization, and wear leveling. In contrast, we design a RAIN layout that also targets tail tolerance. This section briefly describes our basic RAIN layout, enough for understanding how it enables GC-tolerant read (Section 4.3); our more advanced layout will be discussed later along with wear-leveling issues (Section 7).

Figure 4 shows our RAIN layout. For simplicity of illustration, we use 4 channels (C0−C3) and the RAIN stripe width matches the channel count (N = 4). The planes at the same position in each channel form a plane group (e.g., G1). A stripe of pages is based on logical page numbers (LPNs). For every stripe (N−1 consecutive LPNs), we allocate a parity page. For example, for LPNs 0-2, we allocate a parity page P012.

Regarding the flash translation layer (FTL) design (LPN-to-PPN mapping), there are two options: dynamic or static. Dynamic mapping, where an LPN can be mapped to any PPN, is often used to speed writes up (flexible destination). However, in modern SSDs, write latency issues are absorbed by capacitor-backed RAM (Section 4.5); thus, writes are spread across multiple channels. Second, dynamic mapping works well when individual pages are independent; however, RAIN pages are stripe dependent. With dynamic mapping, pages in a stripe can be placed behind one channel, which will underutilize channel parallelism. Another design decision is to choose page- or block-level mapping. We simply choose the page-level approach as block-level is known to suffer from high write amplification.

Given the reasons above, we create a page-level hybrid static-dynamic mapping. The static allocation policies are as follows: (a) an LPN is statically mapped to a plane (e.g., LPN 0 to plane G0C0. 

Fig. 4. Tiny Flash Architecture. The figure illustrates our RAIN layout (Section 4.2), GC-tolerant read (Section 4.3), rotating GC (Section 4.4), and GC-tolerant flush (Section 4.5). We use four channels (C0−C3) for simplicity of illustration. Planes at the same vertical position form a plane group (G0, G1, etc.). A RAIN stripe is based on N−1 LPNs and a parity page (e.g., 012P012).
in Figure 4), (b) \(N-1\) consecutive LPNs and their parity form a stripe (e.g., 012P_{012}), and (c) the stripe pages are mapped to planes across the channels within one plane group (e.g., 012P_{012} in \(G_0\)). Later, we will show how all of these are crucial for supporting GC-tolerant read (Section 4.3) and rotating GC (Section 4.4).

In terms of the dynamic allocation policy, inside each plane or chip, an LPN can be dynamically mapped to any PPN (hundreds of thousands of choices). An overwrite to the same LPN will be redirected to a free page in the same plane (e.g., overwrites to LPN 0 can be directed to any PPN inside the \(G_0C_0\) plane).

To prevent a parity-channel bottleneck (akin to the RAID-4 parity-disk bottleneck), we adopt RAID-5 with a slightly customized layout. First, we treat the set of channels as a RAID-5 group. For example, in Figure 4, P_{012} and P_{345} are in different channels, laid out in a diagonal fashion. Second, as SSD planes form a 2-dimensional layout \((G_iC_j)\) with wearout issues (unlike disk’s “flat” LPNs), we need to ensure that hot parity pages are spread out evenly. To handle this, a solution such as dynamic migration can be employed as we will discuss later (Section 7).

### 4.3 GC-Tolerant Read (GTR)

With RAIN, we can easily support GC-tolerant read (GTR). For a full-stripe read (which uses \(N-1\) channels), GTR is straightforward: if a page cannot be fetched due to an ongoing GC, the page content is quickly regenerated by reading the parity from another plane. In Figure 4, given a full-stripe read of LPNs 0–2, and if LPN 1 is unavailable temporarily, the content is rapidly regenerated by reading the parity (P_{012}). Thus, the full-stripe read is not affected by the ongoing GC. The resulting latency is order(s) of magnitude faster than waiting for GC completion; parity computation overhead takes less than 3\(\mu\)s for \(N \leq 8\) and the additional parity read takes a minimum of 40+100\(\mu\)s (read+transfer latencies; Table 1) and does not introduce much contention.

For a partial-stripe read (\(R\) pages where \(R<N-1\)), GC-tolerant read will generate in total \(N-R\) extra reads; the worst case is when \(R=1\). These \(N-R\) extra parallel reads will add contention to each of the \(N-R\) channels, which might need to serve other outstanding I/Os. Thus, we must introduce a cost calculation to decide whether the extra page reads should be performed or whether the incoming I/O should wait for the GC to finish.

More specifically, our cost calculation for the extra page reads are as follows: We only perform extra reads if \(T_{GCtoComplete}>B(40+100)\mu s\) where \(B\) is the number of busy channels in the \(N-R\) extra reads (for non-busy channels the extra reads are free). In our experience, this policy cuts GC tail latencies effectively and fairly without introducing heavy contention.

In contrast, a “greedy” approach that always performs extra reads causes high channel contention. Later, in Section 6.7, we will compare the performance of our cost calculation with the greedy approach. We also would like to note that our simple cost calculation is effective enough to cut tail latencies. Future extensions to our simple cost calculation can be investigated further.

We emphasize that unlike tail-tolerant speculative execution, often defined as an optimization task that may not be actually needed, GC-tolerant read is affirmative, not speculative; the controller knows exactly when and where GC is happening and how long it will complete. GTR is effective but has a limitation: it does not work when multiple planes in a plane group perform GCs simultaneously, which we address with rotating GC.

### 4.4 Rotating GC (RGC)

As RAIN distributes I/Os evenly over all planes, multiple planes can reach the GC threshold and thus perform GCs simultaneously. For example, in Figure 4, if planes of LPNs 0 and 1 (\(G_0C_0\) and \(G_0C_1\)) both perform GC, reading LPNs 0–2 will be delayed. The core issue is that one parity plane can only “cut” one tail. Double-parity RAIN is not used due to the larger space overhead.
To prevent this, we develop rotating GC (RGC), which enforces that at most one plane in each plane group can perform a GC at a time. Concurrent GCs in different plane groups are still allowed (e.g., one in each $G_i$ as depicted in Figure 4). Note that rotating GC depends on our RAIN layout that ensures every stripe to be statically mapped to a plane group.

We now emphasize our most important message: There will be zero GC-blocked I/Os if rotating GC holds true all the time. The issue here is that our rotating approach can delay a plane’s GC as long as $(N - 1) \times T_{dc}$ (the GC duration). During this period, if all the free pages are exhausted, multiple GCs in a plane group must execute concurrently. This could happen depending on the combination of $N$ and the write intensity. Later, in Section 6.6 and Appendix A, we provide a proof sketch showing that with stripe-width $N \leq 26$, rotating GC can always be enforced under realistic write-intensive scenarios.

Employing a large stripe width (e.g., $N = 32$) is possible but can violate rotating GC, implying that GC tail latencies cannot be eliminated all the time. Thus, in many-channel (e.g., 32) modern SSDs, we can keep $N = 8$ or 16 (e.g., create four 8-plane or two 16-plane groups across the planes within the same vertical position). Increasing $N$ is unfavorable not only because of rotating GC violations, but also due to reduced reliability and the extra I/Os generated for small reads by GTR (Section 4.3). In our evaluation, we use $N = 8$, considering 1/8 parity overhead is bearable.

4.5 GC-Tolerant Flush (GTF)

So far, we only address read tails. Writes are more complex (e.g., due to write randomness, read-and-modify parity update, and the need for durability). To handle write complexities, we leverage the fact that the flash industry heavily employs capacitor-backed RAM as a durable write buffer (or "cap-backed RAM" for short) [15]. To prevent data loss, the RAM size is adjusted based on the capacitor discharge period after power failure; the size can range from tens to hundreds of MB, backed by “super capacitors” [11].

We adopt cap-backed RAM to “durably” absorb all writes quickly. When the buffer occupancy is above 80%, a background flush will run to evict some pages. When the buffer is full (e.g., due to intensive large writes), a foreground flush will run, which will block incoming writes until some space is freed. The challenge to address here is that foreground flush can induce write tails when the evicted pages must be sent to GC-ing planes.

To address this, we introduce GC-tolerant flush (GTF), which ensures that page eviction is free from GC blocking, which is possible given rotating GC. For example, in Figure 4, pages belonging to 3′, 4′ and $P_{3'4'5'}$ can be evicted from RAM to flash while page 5′ eviction is delayed until the destination plane finishes the GC. With rotating GC, GTF can evict $N - 1$ pages in every $N$ pages per stripe without being blocked. Thus, the minimum RAM space needed for the pages yet to be flushed is small. Appendix A suggests that modern SSD RAM sizes are sufficient to support GTF.

For partial-stripe writes, we perform the usual RAID read-modify-write eviction but still without being blocked by GC. Let us imagine a worst-case scenario of updates to pages 7′ and 8′ in Figure 4. The new parity should be $P_{6'7'8'}$, which requires a read of page 6 first. Despite page 6 being unreachable, it can be regenerated by reading the old pages $P_{678}$, 7, and 8, after which pages 7′, 8′, and $P_{6'7'8'}$ can be evicted.

We note that such an expensive parity update is rare as we prioritize the eviction of full-stripe dirty pages to non-GCing planes first and then full-stripe pages to mostly non-GCing planes with GTF. Next, we evict partial-stripe dirty pages to non-GCing planes and finally partial-stripe pages to mostly non-GCing planes with GTF. Compared to other eviction algorithms that focus on reducing write amplification [39], our method adds GC tail tolerance.
5 IMPLEMENTATION

This section describes our implementations of ttFlash, which is available on our website [1]. The number of lines of code (LOC) we report below is obtained with cloc tool with blank lines excluded.

- **ttFlash-Sim (SSDSim):** To facilitate accurate latency analysis at the device level, we first implemented ttFlash in SSDSim [33], a recently popular simulator whose accuracy has been validated against a real hardware platform. It is a newer simulator (released in 2011) compared to the 2008 disksim+SSD [19]. We use SSDSim due to its clean-slate design. We implemented all the ttFlash features by adding 2482 LOC to SSDSim. This involves a substantial modification (+36%) to the vanilla version (6844 LOC). The breakdown of our modification is as follow: plane-blocking (523 LOC), RAIN (582), rotating GC (254), GC-tolerant read (493), and write (630 lines).

- **ttFlash-Emu (“VSSIM++”):** To run Linux kernel and file system benchmarks, we also ported ttFlash to VSSIM, a QEMU/KVM-based platform that “facilitates the implementation of the SSD firmware algorithms” [57]. VSSIM emulates NAND flash latencies on a RAM disk. Unfortunately, VSSIM’s implementation is based on the 5-year-old QEMU-v0.11 IDE interface, which only delivers 10K I/O operations per second (IOPS). Furthermore, as VSSIM is a single-threaded design, it essentially mimics a controller-blocking SSD (1K IOPS under GC).

  These limitations led us to make major changes. First, we migrated VSSIM’s single-threaded logic to a multi-threaded design within the QEMU AIO module, which enables us to implement channel-blocking. Second, we migrated this new design to a recent QEMU release (v2.6) and connected it to the PCIe/NVMe interface. Our modification, which we refer as “VSSIM++”, can sustain 50K IOPS. Finally, we ported ttFlash’s features to VSSIM++, which we refer as ttFlash-Emu, for a total of 869 LOC of changes.

- **Other attempt #1 (OpenSSD):** We attempted implementing ttFlash on real hardware platforms (2011 Jasmine and 2015 Cosmos OpenSSD boards [4]). After a few months of trying, we hit many limitations of the OpenSSD API and programming model. OpenSSD exhibits a controller-blocking behavior with a single-threaded implementation on a single CPU. Its GC operation is designed as a foreground for-loop; thus, when a GC happens, no other incoming IOs can be served (regardless of the fact that the GC does not use all channels or planes). Developers cannot create multiple threads for background GC operations, and channel queues are not exposed through the OpenSSD programming API. All of these make plane-blocking impossible. Furthermore, there is no accessible command for data transfer from the SSD’s DRAM to the host’s DRAM; only a flash-to-host read command is available. This prevents us from transferring parity-regenerated late data (initially stored in SSD’s DRAM) to the host’s DRAM. Finally, there is no access to wall-clock time, which limits our ability to predict the GC remaining time. We reviewed some articles that used OpenSSD and found that they mainly modified the FTL [34, 47, 53] buffering logic [35], and interface [37, 50], which are all possible within the elegant simplicity of the OpenSSD programming model (which is its main goal). We would like to reiterate that these are not hardware limitations, but rather, the ramifications of the elegant simplicity of OpenSSD programming model (which is its main goal). While our efforts in implementing ttFlash on OpenSSD did not lead to a fruitful result, our conversations with hardware architects suggest that ttFlash is implementable on a real firmware (e.g., roughly a 1-year development and testing project on a FPGA-based platform).

- **Other attempt #2 (LightNVM QEMU):** Finally, we also investigated the LightNVM (OpenChannel SSD) QEMU test platform [17]. LightNVM [22] is an in-kernel framework that manages OpenChannel SSD (which exposes individual flash channels to the host, akin to Software-Defined Flash [49]). Currently, neither OpenChannel SSD nor LightNVM’s QEMU test platform support intra-SSD copy-page command. Without such support and since GC is managed by the
host OS, GC-ed pages must cross back and forth between the device and the host. This creates heavy background-vs.-foreground I/O transfer contention between GC and user I/Os. For example, the user’s maximum 50K IOPS can downgrade to 3K IOPS when GC is happening. We leave this integration for future work after the intra-SSD copy-page command is supported.

## 6 Evaluation

We now present extensive evaluations showing that TTFLASH significantly eliminates GC blocking (Section 6.1), delivers more stable latencies than the state-of-the-art preemptive GC (Section 6.2) and other GC optimization techniques (Section 6.3), and does not significantly increase program and erase cycles (P/E cycles) beyond the RAIN overhead (Section 6.4).

**Workloads:** We evaluated two implementations: ttFlash-Sim (on SSDSim) and ttFlash-Emu (on VSSIM++), as described in Section 5. For ttFlash-Sim evaluation, we used six real-world block-level traces from Microsoft Windows Servers as listed in the figure titles of Figure 5. Their detailed characteristics are publicly reported [3, 38]. By default, for each trace, we chose the busiest hour (except the 6min TPCC trace). For ttFlash-Emu evaluation, we used filebench [2] with six personalities as listed in the x-axis of Figure 8.

**Hardware parameters:** For ttFlash-Sim, we used the same 256GB parameter values provided in Table 1 with 64MB cap-backed RAM and a typical device queue size of 32. ttFlash-Emu used the same parameters but its SSD capacity was only 48GB (limited by the machine’s DRAM). We used a machine with 2.4GHz eight-core Intel Xeon Processor E5-2630-v3 and 64GB DRAM. The simulated and emulated SSD drives are pre-warmed up with the same workload.

### 6.1 Main Results

- **Tiny tail latencies:** Figure 5 shows the CDF of read latencies from the six trace-driven experiments run on ttFlash-Sim. Note that we only show read latencies; write latencies were fast and stable as all writes were absorbed by cap-backed RAM (Section 4.5). As shown in Figure 5, the base approach (“Base” = the default SSDSim with channel-blocking and its most-optimum FTL [33] and without RAIN) exhibits long tail latencies. In contrast, as we add each TTFLASH feature one at a time on top of the other: +PB (plane-blocking GC), +GTR (GC-tolerant read), and RGC (rotating GC), significant improvements are observed. When all features are added (RGC+GTR+PB), the tiny tail latencies are close to those of the no-GC scenario, as we explain later.

  Figure 6 plots the average latencies of the same experiments. This graph highlights that although the latencies of TTFLASH and Base are similar at 90th percentile (Figure 5), the Base’s long tail latencies severely impact its average latencies. Compared to Base, TTFLASH’s average latencies are 2.5–7.8 × faster.

- **ttFlash vs. NoGC:** To characterize the benefits of TTFLASH’s tail latencies, we compared TTFLASH to a perfect “no-GC” scenario (“NoGC” = TTFLASH without GC and with RAIN). In NoGC, the same workload ran without any GC work (with a high GC threshold), thus all I/Os observed raw flash performance.

  Table 2 shows the slowdown from NoGC to TTFLASH at various high percentiles. As shown, TTFLASH significantly reduces GC blocking. Specifically, at the 99–99.9th percentiles, TTFLASH’s slowdowns are only 1.00 to 1.02×. Even at 99.99th percentile, TTFLASH’s slowdowns are only 1.0 to 2.6×. In comparison, Base suffers from 5.6–138.2× slowdowns between 99–99.99th percentiles (as obvious in Figure 5); for readability, NoGC lines are not plotted in that figure. In terms of average latencies, Figure 6 shows that TTFLASH performs the same with or without GC.

- **GC-blocked I/Os:** To show what is happening inside the SSD behind our speed-ups, we count the percentage of read I/Os that are blocked by GC (“%GC-blocked I/Os”), as plotted in Figure 7.
Fig. 5. **Tail Latencies.** The figures show the CDF of read latencies ($x=0–80$ ms) in different workloads as we add each ttFlash strategy: +PB (plane-blocking GC), +GTR (GC-tolerant read), and +RGC (rotating GC). The y-axis shows 95–100th percentiles.

Equally important, we emphasize that GC-blocked I/Os fill up the device queue, creating queueing delays that prevent new host I/Os from entering the device, which we count as “%queue-blocked I/Os.” Thus, each bar in the figure has two parts: %GC-blocked (bottom, bold edge) and %queue-blocked I/Os (top), divided with a small horizontal borderline.

Figure 7 shows that with Base, without GC tolerance, 2–5% of reads are blocked by GC. As they further cause queueing delays, in total, 2–7% of blocked I/Os cannot be served. As each ttFlash feature is added, more I/Os are unblocked. With all the features in place (“All” bars), there are only 0.03–0.05% blocked I/Os, with the exception of MSNFS (0.7%). The only reason why it is not 0% is that for non-full-stripe reads, ttFlash will wait for GC completion only if the remaining time is shorter than the overhead of the extra reads (as explained in Section 4.3). We still count these I/Os as blocked, albeit only momentarily.

We next evaluate ttFlash-Emu with filebench [2]. Figure 8 shows the average latencies of filebench-level read operations (including kernel, file-system, and QEMU overheads in addition
Fig. 6. **Average Latencies.** The figure compares the average read latencies of **Base**, **ttFlash**, and **NoGC** scenarios from the same experiments in Figure 5.

Table 2. **ttFlash vs. NoGC (Almost No Tail)**

<table>
<thead>
<tr>
<th>Percentile:</th>
<th>DAP</th>
<th>DTRS</th>
<th>Exch</th>
<th>LMBE</th>
<th>MSN</th>
<th>TPCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.99th</td>
<td>1.00x</td>
<td>1.24</td>
<td>1.18</td>
<td>1.96</td>
<td>1.00</td>
<td>2.56</td>
</tr>
<tr>
<td>99.9th</td>
<td>1.00x</td>
<td>1.01</td>
<td>1.01</td>
<td>1.02</td>
<td>1.01</td>
<td>1.01</td>
</tr>
<tr>
<td>99th</td>
<td>1.00x</td>
<td>1.02</td>
<td>1.10</td>
<td>1.01</td>
<td>1.03</td>
<td>1.02</td>
</tr>
</tbody>
</table>

The numbers above represent the **Slowdown Ratio** of **ttFlash** read latencies compared to **NoGC** at high percentiles. For example, in DTRS, at 99.99th percentile, **ttFlash**’s read latency is only 1.24× slower than **NoGC**’s read latency.

Fig. 7. **%GC-blocked read I/Os.** The figure above corresponds to the results in Figure 5. The bars represent the ratio (in percentage) of read I/Os that are GC-blocked (bottom bar) and queue-blocked (top bar) as explained in Section 6.1. "All" implies PB+GTR+RGC (plane-blocking GC + GC-tolerant read + rotating GC).

Fig. 8. **Filebench on ttFlash-Emu.** The top and bottom figures show the average latencies of read operations and the percentage of GC-blocked reads, respectively, across six filebench personalities. **Base** represents our **VSSIM++** with channel-blocking (Section 5).
to device-level latencies) and the percentage of GC-blocked reads measured inside ttFlash-Emu. We do not plot the latency CDF as filebench only reports average latencies. Overall, ttFlash-Emu shows the same behavior as ttFlash-Sim.

### 6.2 ttFlash vs. Preemptive GC

As mentioned before, many existing works optimize GC, but does not eliminate its impact. One industry standard in eliminating (“postponing”) GC impact is preemptive GC \[10\]. We implemented preemptive GC in SSDSim based on existing literature \[44\]. The basic idea is to interleave user I/Os with GC operations. That is, if a user I/O arrives while a GC is happening, future copybacks should be postponed.

Figure 9(a) compares ttFlash-Sim, preemptive, and NoGC scenarios for the DTRS workload (other workloads lead to the same conclusion). As shown, ttFLASH is closer to NoGC than to preemptive GC. The reason is that preemptive GC incurs a delay from waiting for the block erase (up to 2ms) or the current page copyback to finish (up to 800μs delay), mainly because the finest-grained preemption unit is a page copyback (Section 3). ttFLASH on the other hand can rapidly regenerate the delayed data.

Most importantly, ttFLASH does not postpone GC indefinitely. In contrast, preemptive GC delays GC impact to the future, with the hope that there will be idle time. However, with a continuous I/O stream, at some point the SSD will hit a GC high water mark (not enough free pages), which is when preemptive GC becomes non-preemptive \[44\]. To create this scenario, we ran the same workload but made SSDSim’s GC threshold hit the high water mark. Figure 9(b) shows that as preemptive GC becomes non-preemptive, it becomes GC-intolerant and creates long tail latencies.

To be more realistic with the setup, we performed an experiment similar to the Semi-Preemptive GC article \[44, Section IV\]. We re-rated DTRS I/Os by 10× and re-sized them by 30×, to reach the high GC water mark (which we set to 75% to speed up the experiment). Figure 9(c) shows the
timeline of observed latencies with ttFlash and preemptive GC. We also ran a synthetic workload with continuous I/Os to prevent idle time (Figure 9(d)); the workload generates 28-KB I/Os (full-stripe) every 130μs with 70% read and 30% write). Overall, Figures 9(c) and (d) highlight that preemptive GC created backlogs of GC activities, which eventually caused SSD “lock-down” when page occupancy reached the high watermark. On the other hand, ttFlash provides stable latencies without postponing GC activities indefinitely.

The last two experiments above created a high intensity of writes, and within the same experiments, our GC-tolerant flush (GTF; Section 4.5) provided stable latencies, as implicitly shown in Figures 9(c) and (d).

6.3 ttFlash vs. GC Optimizations

GC can be optimized and reduced with better FTL management, special coding, a novel write buffer scheme or an SSD-based log-structured file system. For example, in comparison to base approaches, value locality reduces the erase count by 65% [30, Section 5], flash-aware RAID by 40% [36, Figure 20], BPLRU by 41% [39, Section 4 and Figure 7], eSAP by 10–45% [41, Figures 11–12], F2FS by 10% [43, Section 3], LARS by 50% [45, Figure 4], and FRA by 10% [46, Figure 12], SFS by 7.5× [48, Section 4], and WOM codes by 33% [55, Section 6].

In contrast to these efforts, our approach is fundamentally different. We do not focus on reducing the number of GCs, but instead, eliminate the blocking nature of GC operations. With reduction, even if the GC count is reduced multiple times, reduction only makes GC-induced tail latencies shorter but not disappear (e.g., as in Figure 5). Nevertheless, the techniques above are crucial to extending SSD lifetime, hence orthogonal to ttFlash.

6.4 Write (P/E Cycle) Overhead

Figure 10 compares the number of GCs (P/E cycles) completed by the Base approach and ttFlash within the experiments in Figure 5. We make two observations. First, ttFlash does not delay GCs; it actively performs GCs at a similar rate as in the base approach, but yet still delivers predictable performance. Second, ttFlash introduces 15–18% additional P/E cycles (in 4 of 6 workloads), which mainly comes from RAIN; as we use $N = 8$, there are roughly 15% (1/7) more writes minimum, from one parity write for every seven ($N-1$) consecutive writes. The exceptions are 53% additional P/E cycles in MSNFS and TPCC, which happen because the workloads generate many small random writes, causing almost one parity write for every write. For this kind of workload, large buffers do not help. Overall, higher P/E cycles are a limitation of ttFlash, but also a limitation of any scheme that employs RAIN.

6.5 ttFlash vs. No RAIN

Earlier, in Figure 6, we showed that ttFlash has about the same average latencies as NoGC (ttFlash without GC and with RAIN). In further experiments, we also compared ttFlash to
“NoGC\textsubscript{IR}” (i.e., Base without GC and without RAIN). As shown in Figure 11, ttFlash’s average latencies are 1.09–1.33× of NoGC\textsubscript{IR}’s. The RAIN-less NoGC\textsubscript{IR} is faster because it can utilize all channels. Thus, if the workload is pure read only, ttFlash performs less optimally than NoGC\textsubscript{IR}. This is a limitation of ttFlash (or any SSD that employs RAIN across channels); that is, as ttFlash employs RAIN, the channels experience a slight contention. For example, in Figure 4, reading LPNs 0–3 will incur contention on channel 0 (as LPNs 0 and 3 must be read via channel 0). In a RAIN-less setup, the same read will utilize all the four channels (LPNs 0 and 3 do not contend on channel 0). In our experiments, as we use $N = 8$, we lose 1 of 8 channels.

### 6.6 ttFlash Under Write Bursts

ttFlash can circumvent GC blocking when rotating GC (RGC) is enforced (i.e., at most one GC runs in every plane group as explained in Section 4.4). A limitation of ttFlash is that under heavy write bursts, multiple GCs per plane group must be allowed to keep the number of free pages stable. Figure 12(a) shows the limit of our 256GB drive setup (Table 1) with $N = 8$. In the figure, we use “Drive Writes Per Day (DWPD)” as a metric to represent the workload. This is important to reflect whether the workload exhibits a typical write intensity in practice.

As shown in Figure 12, at 6 DWPD (55MB/s in our setup), there are almost no GC-blocked reads, hence tiny tail latencies. In our setup, 1 DWPD (“Drive Writes Per Day”) implies 256GB/8h (9.1MB/s) of writes; as described in Appendix A, we generously use 8h to represent a “day” (as opposed to 24h). However, at 7 DWPD (64MB/s), ttFlash exhibits some tail latencies, observable at the 90th percentile. We emphasize that this is still much better than the Base approach, where the tail latencies are observed starting at the 20th percentile (not shown in the figure).

Figure 12(b) shows that if we force (“F”) only one GC per plane group all the time (“F-RGC”), at 7 DWPD, the percentage of free pages (the y-axis) continuously drops over time (the x-axis). That is, RGC cannot keep up with the write bursts (i.e., there are more pages being written compared to...
pages that are being cleaned and freed). Thus, to keep the number of free pages stable (e.g., around 30%) under intensive write bursts, we must allow multiple GCs to happen per plane group. The result is reflected by the “RGC, 7DWPD” line in Figure 12(b) where the number of free pages is kept stable at 30% of the SSD space. Because during a write burst, we now allow multiple GCs to happen in a plane group, ttFlash (with single parity) cannot cut the GC tail latency. This is the reason we observed latency tail in the “7 DWPD” line in Figure 12(a).

Although ttFlash does not completely eliminate GC tail latency under write bursts, we would like to emphasize that such intensive writes are hopefully rare. For example, for 3- to 5-year lifespans, modern multi-level cell (MLC) / triple-level cell (TLC) drives must conform to 1-5 DWPD [18]. Thus, we believe ttFlash is a fitting solution for typical workloads.

6.7 ttFlash Cost Calculation vs. Greedy Algorithm

In Section 4.3, we discussed that ttFlash employs a simple cost calculation in deciding whether to wait for an ongoing GC to finish or to spawn extra page reads to regenerate the GC-blocked page. At the other extreme, ttFlash can also use a “greedy” approach that always performs extra reads, which will actually cause high channel contention.

Figure 13 shows the performance difference between the ttFlash cost calculation and the greedy approach. Figure 14 shows the percent of extra page reads they generated. As shown, and as discussed in Section 4.3, because the greedy approach always generates extra page reads for every GC-blocked IO, including partial-stripe read IOs, the extra page reads cause more contentions to other channels than are needed to serve other IOs. As a result, the overall performance is reduced. As shown, the latencies observed with the greedy approach are longer than the latencies in ttFlash. The most obvious impact is felt within the highly-intensive MSNFS server. The latency gap between ttFlash cost calculation and the greedy approach can be above 1ms.

7 LIMITATIONS AND DISCUSSIONS

We now summarize the limitations of ttFlash. First, ttFlash depends on RAIN, hence the loss of one channel per N channels (as evaluated in Section 6.5). Increasing N will reduce channel loss but cut fewer tail latencies under write bursts (Appendix A). Under heavy write bursts, ttFlash cannot cut all tails (as evaluated in Section 6.6 and discussed in Appendix A). Finally, ttFlash requires intra-plane copybacks, skipping ECC checks, which requires future work as we address below. Below we discuss ECC checking and wear-leveling issues.

7.1 ECC Checking (with Scrubbing)

ECC-check is performed when data pass through the ECC engine (part of the controller). On foreground reads, before data is returned to the host, ECC is always checked (ttFlash does not modify this property). Due to increasing bit errors, we suggest that ECC checking runs more frequently, for example, by forcing all background GC pages copied back to be read out from the plane and through the controller, albeit with reduced performance.

ttFlash, however, depends on intra-plane copybacks, which implies no ECC checking on pages copied back, potentially compromising data integrity. A simple possible solution to compensate for this problem is periodic idle-time scrubbing within the SSD, which will force flash pages (user and parity) to flow through the ECC engine. This is a reasonable solution for several reasons. First, SSD scrubbing (unlike disk) is fast given the massive read bandwidth. For example, a 2GB/s 512GB client SSD can be scrubbed in under 5min. Second, scrubbing can be easily optimized, for example, by only selecting blocks that have recently been GC-ed or have higher P/E counts and a larger history of bit flips, which by implication can also reduce read disturbances. Third, periodic background operations can be scheduled without affecting foreground performance (there is a rich
literature in this space [20]). Finally, such scrubbing can be done within the SSD itself. Supporting scrubbing in the drive firmware (not the host) has long been a standard practice by disk vendors. We believe that adding similar functionality to SSD firmware is feasible. This way, background ECC-checked data does not need to flow between the host DRAM and the flash drive.

We would like to emphasize that we only propose when background ECC checking will be run, but not where and how ECC is placed and managed. The standard is to store ECC along with the flash sectors (e.g., 13–117 ECC bits within every 528-byte sector [5]). The intra-plane copyback will automatically move the entire sector including the ECC. The ttFlash design does not change this standard. Nevertheless, more future work is required to evaluate the ramifications of background ECC checks.

7.1.1 Cost Estimation of Background ECC Checks. Table 3 shows the cost estimation of background scrubbing (for ECC checking). In this experiment, we ran some of the workloads with
Fig. 14. %Extra page reads in TTFlash cost calculation vs. greedy approach. The graph is explained in Section 6.7. The bar graph (with log-scaled y-axis) shows the % of extra page reads needed to circumvent GC blocking.

Table 3. Cost Estimation of Background ECC Checks

<table>
<thead>
<tr>
<th>Traces</th>
<th>1 Hour</th>
<th></th>
<th>24 Hours</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#ECC Checks</td>
<td>#GC Copybacks</td>
<td>%</td>
<td>#ECC Checks</td>
<td>#GC Copybacks</td>
</tr>
<tr>
<td>DAPPS</td>
<td>240k</td>
<td>289k</td>
<td>83%</td>
<td>1,524k</td>
<td>3,318k</td>
</tr>
<tr>
<td>DTRS</td>
<td>444k</td>
<td>471k</td>
<td>94%</td>
<td>3,362k</td>
<td>8,557k</td>
</tr>
<tr>
<td>EST</td>
<td>134k</td>
<td>334k</td>
<td>40%</td>
<td>149k</td>
<td>8,258k</td>
</tr>
<tr>
<td>MSNFS</td>
<td>702k</td>
<td>890k</td>
<td>79%</td>
<td>2,872k</td>
<td>17,766k</td>
</tr>
</tbody>
</table>

"#ECC Checks" represents the number of to-be-ECC-checked pages in the background (every 1 hour or 24 hours) and "#GC Copybacks" the number of pages GC copied back. "%" represents the ratio of "#ECC Checks" over "#GC Copybacks" as discussed in Section 7.1.1. "k" denotes "x1,000." The table shows results for 1-hour and 24-hour traces. The latter implies 24 continuous runs of the 1-hour trace. LMBE and TPCC are not shown; TPCC is less than one hour long and SSDSim crashes (potentially due to memory leaks) when running the 24-hour LMBE trace.

1- to 24-hour windows and measured two metrics: (a) the number of GC pages copied back and (b) the number of to-be-ECC-checked pages.

The first metric, the number of pages copied back ("#GC Copybacks" in Table 3), represents the valid pages that have been moved by the GC process during the experiment. Because we leverage intra-plane copybacks, these pages copied-back only flow within the plane, through the plane registers (see Section 3). However, in another SSD design standard where GC-ed valid pages must be checked through the controller, this number represents the total GC-related page movements through the controller. The second metric, the number of to-be-ECC-checked pages ("#ECC Checks" in Table 3), represents the number of pages that must be checked in the background later by the background scrubbing process.

As shown in the "%" column in Table 3, the number of to-be-ECC-checked pages is smaller than the number of pages copied back. The reason is that a valid page with LPN \( P \) might be moved many times by the GC process, but at the end, the scrubbing only needs to ECC-check the latest physical location of LPN \( P \). For example, if a page with LPN \( P \) is copied back 5 times to five different physical pages throughout the experiment, the background scrubbing only needs to ECC-check the data in the latest location.

The "%" column in the 1h section of Table 3 shows that for 1h traces, the number of to-be-ECC-checked pages is relatively the same as the number of pages copied back (between 40 and 94%). However, when we ran them for 24 hours, the number of to-be-ECC-checked pages was stable over time while the number of GC copybacks continued to increase (as GC process continuously runs). As shown in the table, the percentage for the 24-hour window reduces to 2–46%.

The results in Table 3 suggest that a nightly scrubbing method is feasible and non-intrusive. Even within the 1h traces, the number of pages to be ECC-checked is only between 134 and 702
Table 4. Read Disturbances Potential from Background ECC Checks

<table>
<thead>
<tr>
<th>Traces</th>
<th>1 Hour</th>
<th>24 Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#ECC Checks</td>
<td>#Fg. Read</td>
</tr>
<tr>
<td>DAPPS</td>
<td>240k</td>
<td>567k</td>
</tr>
<tr>
<td>DTRS</td>
<td>444k</td>
<td>8,349k</td>
</tr>
<tr>
<td>EST</td>
<td>134k</td>
<td>132k</td>
</tr>
<tr>
<td>MSNFS</td>
<td>702k</td>
<td>5,453k</td>
</tr>
</tbody>
</table>

"#ECC Checks" represents the number of to-be-ECC-checked pages in the background (every 1 hour or 24 hours) and "#Fg. Read" the number of pages read in the foreground (e.g., by users). "%" represents the ratio of #ECC over #FgRead as discussed in Section 7.1.1. "k" denotes "x1,000." The table shows results for 1h and 24-hour traces.

Another concern with background ECC checks is the potential increase in read disturbances (as the scrubbing process introduces more read operations). For example as shown in 1h section of Table 4, the foreground operations ("Fg. Read" column) read between 132 and 8,349 thousand pages, thus the to-be-ECC-checked pages are 5–102% of the foreground reads. However, with daily scrubbing, the 24h section of Table 4 shows that the amount of data to be scrubbed is far less (only 2–11%) than the total number of bytes read by foreground operations. Thus, foreground reads are still the dominant cause of read disturbances.

7.2 Wear Leveling (via Horizontal Shifting and Vertical Migration)

Our static RAIN layout (Section 4.2) in general does not lead to wear-out imbalance in common cases. However, rare cases such as random-write transactions (e.g., MSNFS) cause imbalanced wear-outs (at chip/plane level).

Imbalanced wear-outs can happen due to the two following cases: (1) There is write imbalance within a stripe (MSNFS exhibits this pattern). In Figure 4, for example, if in stripe S₀ {012P₀12₀}, LPN 1 is more frequently updated than the rest, the planes of LPN 1 and P₀12₀ will wear out faster than the other planes in the same group. (2) There is also write imbalance across the stripes. For example, if stripes in group G₀ (e.g., stripe {012P₀12₀}) are more frequently updated than stripes in other groups, then the planes in G₀ will wear out faster.

The two wear-out problems above can be fixed by dynamic horizontal shifting and vertical migration. With horizontal shifting, we can shift the parity locations of stripes with imbalanced hot pages. For example, S₀ can be mapped as {12P₀12₀} across the 4 planes in the same group; LPN 1 and P will now be directed to colder planes. With vertical migration, hot stripes can be migrated from one plane group to another, balancing the wear-out across plane groups.

As a combined result, an LPN is still and always statically mapped to a stripe number. A stripe, by default, is statically mapped to a plane group and has a static parity location (e.g., S₀ is in group G₀ with P₀12 behind channel C₃). However, to mark dynamic modification, we can add a "mapping-modified" bit in the standard FTL table (LPN-PPN mapping). If the bit is zero, then the LPN-PPN translation performs as usual, as the stripe mapping stays static (the common case). If the bit is set (e.g., in rare workload cases), then the LPN-PPN translation must consult a new
stripe-information table that stores the mapping between a stripe \( S_k \) to a group number \( G_i \) and parity channel position \( C_j \).

Overall, this article focuses on the elimination of GC tail latencies. The discussion above is to show that ECC and wear-leveling concerns can be addressed.

8 RELATED WORK

We now discuss other work related to Tiny-Tail Flash.

**GC-impact reduction:** Our work is about eliminating GC impacts, while many other existing works are about reducing GC impacts. There are two main reduction approaches: isolation and optimization, both with drawbacks. Isolation (e.g., OPS isolation [40]) only isolates one tenant (e.g., sequential) from another (e.g., random-write). It does not help a tenant with both random-write and sequential workloads on the same dataset. OPS isolation must differentiate users while Tiny-Tail Flash is user-agnostic. GC optimization, which can be achieved by better page layout management (e.g., value locality [30], log-structured [24, 43, 48]) only helps in reducing GC period but does not eliminate blocked I/Os.

**GC-impact elimination:** We are only aware of a handful of works that attempt to eliminate GC impact, which fall into two categories: without or with redundancy. Without redundancy, one can eliminate GC impact by preemption [23, 44, 54]. We already discussed the limitations of preemptive GC (Section 6.2; Figure 9). With redundancy, one must depend on RAIN. To the best of our knowledge, our work is the first one that leverages an SSD’s internal redundancy to eliminate GC tail latencies. There are other works that leverage redundancy in flash arrays (described later below).

**RAIN:** SSD’s internal parity-based redundancy (RAIN) has become a reliability standard. Some companies reveal such usage but unfortunately without topology details [7, 13]. In the literature, we are aware of only four major ones: eSAP [41], PPC [36], FRA [46], and LARS [45]. These efforts, however, mainly are concerned with write optimization and wear leveling in RAIN but do not leverage RAIN to eliminate GC tail latencies.

**Flash array:** Tiny-Tail Flash works within a single SSD. In the context of SSD arrays, we are aware of two published techniques on GC tolerance: Flash on Rails [52] and Harmonia [42]. Flash on Rails [52] eliminates read blocking (read-write contention) with a ring of multiple drives where one to two drives are used for write logging and the other drives are used for reads. The major drawback is that read/write I/Os cannot utilize the aggregate bandwidth of the array. In Harmonia [42], the host OS controls all the SSDs to perform GC at the same time (i.e., it is better that all SSDs are “unavailable” at the same time, but then provide stable performance afterwards), which requires more complex host-SSD communication.

**Storage tail latencies:** A growing number of works recently investigated sources of storage-level tail latencies, including background jobs [20], file system allocation policies [32], block-level I/O schedulers [56], and disk/SSD hardware-level defects [27, 28, 31]. An earlier work addresses load-induced tail latencies with RAID parity [21]. Our work specifically addresses GC-induced tail latencies.

9 CONCLUSION

SSD technologies have changed rapidly in the last few years; faster and more powerful flash controllers are capable of executing complex logic; parity-based RAIN has become a standard means of data protection; and capacitor-backed RAM is a de-facto solution to address write inefficiencies. In our work, we leverage a combination of these technologies in a way that has not been done before. This in turn enables us to build novel techniques such as plane-blocking GC, rotating GC,
GC-tolerant read and flush, which collectively deliver a robust solution to the critical problem of GC-induced tail latencies.

A PROOF SKETCH

Limitation of maximum stripe width (N): We derive the maximum stripe width allowable (N) such that rotating GC (Section 4.4) is always enforced. That is, as we can only cut one tail, there should be at most one GC per plane group at all time. Thus, a plane might need to postpone its GC until other planes in the same group complete their GCs (i.e., delayed by (N−1) × Tgc). We argue that N should be at least 8 for a reasonable parity space overhead (12.5%); a smaller stripe width will increase space overhead. Below we show that N = 8 is safe even under intensive writes. Table 5 summarizes our proof, which is based on a per-plane, per-second analysis. We first use concrete values and later generalize the proof.

- Table 5a: We use typical parameters: 4KB page (Spage), 4KB register size (Sreg), 25% valid pages (%validPg), 840μs of GC copyback time per page (Tcopyback), and 900μs of user write latency per page (TusrWrt). Due to intensive copybacks (tens of ms), the 2ms erase time is set to 0 to simplify the proof.
- Table 5b: Each plane’s bandwidth (BWpl) defines the maximum write bandwidth, which is 4.5MB/s, from the register size (Sreg) divided by the user-write latency (TusrWrt); all writes must go through the register.
- Table 5c: With the 4.5MB/s maximum plane bandwidth, there are 1,152 pages written per second (#Wpg/s), which will eventually be GC-ed.
- Table 5d: Intensive writes imply frequent overwrites; we assume 25% valid pages (%validPg) to be GC-ed, resulting in 288 pages copied back per second (#CBpg/s). The %validPg can vary depending on user workload.
- Table 5e: With 288 page copybacks, the total GC time per second per plane (Tgc/s) is 242 ms.

Table 5. Proof Sketch (Appendix A)

<table>
<thead>
<tr>
<th>Step</th>
<th>Equation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.</td>
<td>Spage=4KB; Sreg=4KB; %validPg=25%; Tprog=800μs; Tread=40μs; Tchannel=100μs; Tcopyback=Tprog+Tread=840μs; (Terase=0); TusrWrt=Tprog+Tchannel=900μs;</td>
<td></td>
</tr>
<tr>
<td>b.</td>
<td>BWpl = Sreg/TusrWrt</td>
<td>4.5 MB/s</td>
</tr>
<tr>
<td>c.</td>
<td>#Wpg/s = BWpl/Spage</td>
<td>1152 pg/s</td>
</tr>
<tr>
<td>d.</td>
<td>#CBpg/s = %validPg × #Wpg/s</td>
<td>288 pg/s</td>
</tr>
<tr>
<td>e.</td>
<td>Tgc = #CBpg/s × Tcopyback</td>
<td>242 ms</td>
</tr>
<tr>
<td>f.</td>
<td>N &lt; 1/Tgc</td>
<td>&lt; 4</td>
</tr>
<tr>
<td>g.</td>
<td>N &lt; Spage/BWplane×%validPg×Tcopyback</td>
<td></td>
</tr>
<tr>
<td>h.</td>
<td>DWPD=5; PWPD=5; Spl=4GB; day=8hrs</td>
<td></td>
</tr>
<tr>
<td>i.</td>
<td>BWpl = Spl×DWPD/day (in practice) = 4GB × 5/8hrs = 0.7 MB/s</td>
<td></td>
</tr>
<tr>
<td>j.</td>
<td>Tgc/s = plug (i) to (c,d,e) = 38 ms</td>
<td></td>
</tr>
</tbody>
</table>

With our concrete values above, for rotating...
GC to hold true all the time, \( N \) must be less than 4 (\( T_{gc} \) of 242 ms). Fortunately, \( N \) can be larger in practice (Table 5g-j). To show this, below we first generalize the proof.

- **Table 5g:** We combine all the equations above to the equation in Table 5g, which clearly shows that \( N \) goes down if \( BW_{pl} \) or \( \%_{validPg} \) is high. Fortunately, we find that the constant 4.5 MB/s throughput (\( BW_{pl} \)) in Table 5b is unrealistic in practice, primarily due to limited SSD lifetime. An MLC block is limited to about 5000–10,000 erase cycles and a TLC block to 3000 erase cycles. To ensure 3- to 5-year lifespan, users typically conform to the Drive Writes Per Day (DWPD) constraint (1–5 DWPD for MLC/TLC drives) [18].

- **Table 5h:** Let us assume a worst-case scenario of 5 DWPD, which translates to 5 \( P_{WPD} \) (plane writes per day) per plane. To make it worse, let us assume a “day” is 8h. We set the plane size (\( S_{pl} \)) to 4 GB (Section 3).

- **Table 5i:** The more realistic parameters above suggest that a plane only receives 0.7MB/s (4GB \times 5/8h), which is 6.5× less intense than the raw bandwidth (5b).

- **Table 5j:** If we plug in 0.7MB/s to the equations in Table 5c-e, then the GC time per plane (\( T_{gc} \)) is only 38ms, which implies that \( N \) can be as large as 26.

In conclusion, \( N \approx 8 \) is likely to always satisfy rotating GC in practice. In a 32-channel SSD, \( N = 32 \) can violate rotating GC; GC-tolerant read (Section 4.3) cannot always cut the tails. Overall, Table 4g defines the general constraint for \( N \). We believe the most important value is \( BW_{pl} \). The other parameters stay relatively constant; \( S_{page} \) is usually 4KB, \( \%_{validPg} \) is low with high overwrites, and \( T_{copyback} \) can increase by 25% in TLC chips (vs. MLC).

**Minimum size of cap-backed RAM:** With rotating GC, the RAM needs to only hold at most \( 1/N \) of the pages whose target planes are GC-ing (Section 4.5). In general, the minimum RAM size is \( 1/N \) of the SSD maximum write bandwidth. Even with an extreme write bandwidth of the latest datacenter SSD (e.g., 2GB/s) the minimum RAM size needed is only 256MB.

**ACKNOWLEDGMENTS**

We thank Sam H. Noh (our shepherd), Nisha Talagala, and the anonymous reviewers for their tremendous feedback.

**REFERENCES**


Received June 2017; accepted June 2017