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

Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminatahan Sundararaman†, Andrew A. Chien, and Haryadi S. Gunawi

Abstract

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. It is built on three SSD internal advancements: powerful controllers, parity-based RAIN, and capacitor-backed RAM, but is dependent on the use of intra-plane copyback operations. We show that TTFLASH comes significantly close to a “no-GC” scenario. Specifically, between 99–99.9th 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.

1 Introduction

Flash storage has become the mainstream destination for storage users. The SSD consumer market continues to grow at a significant rate [8]. SSD-backed cloud-VM instances are becoming the norm [7, 13], and flash/SSD arrays are a popular solution for high-end storage servers [23, 30, 44]. From the users side, they demand fast and stable latencies [25, 28]. However, SSDs do not always deliver the performance that users expect [15]. Some even suggest that flash storage “may not save the world” (due to the tail latency problem) [5]. Some recent works dissect why it is hard to meet SLAs with SSDs [36] and reveal high performance variability in 7 million hours of SSDs deployments [30].

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× [5, 24]. In the last decade, there is a large body of work that reduces the number of GC operations with a variety of novel techniques [29, 35, 36, 37, 39, 43, 48]. However, we find almost no work in literature that attempts to eliminate the blocking nature of GC operations and deliver steady SSD performance in long runs.

We address this urgent issue with “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 ECC is no longer deemed sufficient [33, 37, 45]. Due to this increasing failure, modern commercial SSDs employ parity-based redundancies (RAIN) as a standard data protection mechanism [6, 12]. 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” [10, 14], 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 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 the guarantee to always cut “one tail” with one parity, 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 and left for future work. We recommend background ECC checks to be performed in the background to overcome this limitation (§7).

We first implemented TTFLASH in SSDSim [32] in order to simulate accurate latency analysis at the device level. Next, to run real file systems and applications, we also port TTFLASH to a newer QEMU/KVM-based platform based on VSSIM [50].

With a thorough evaluation (§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 99–99.99th percentiles, compared to the perfect no-GC scenario, a base approach suffers from 5.6–138.2× GC-induced slowdowns. TTFLASH on the other hand is only 1.0 to 2.6× slower than the no-GC case, which confirms our near-complete elimination of GC blocking and the resulting tail latencies.

We also show that TTFLASH is more stable than state-of-the-art approaches that reduce GC impacts such as preemptive GC [9, 40] (§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 (§6.3, §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 (§2), SSD primer (§3), TTFLASH design (§4), implementation (§5), evaluation (§6), and limitations (§7), and related and conclusion (§7-9).

2 GC-Induced Tail Latency

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

In Figure 1a, we ran a foreground thread that executes 16-KB random reads, concurrently with background threads that inject 4-KB random-write noises at 1, 2.5, and 5 KWPS (far below the max 70 KWPS) across three experiments. We measure \( L_i \), the latency of every 16-KB foreground read. Figure 1a 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 queuing delays, we ran the same experiments but now with random-read noises (1, 2.5, and 5 KRPS). The read-noise results are plotted as the three overlapping thin lines marked “ReadNoise,” which represents a perfect no-GC scenario. As shown, with 5 KWPS noise, read operations become 15×, 19×, and 96× slower compared to no-GC scenarios, at 90th, 95th and 99th percentiles, respectively.

In Figure 1b, we keep the 5-KWPS noise and now vary the I/O size of the foreground random reads (8, 16, 32, and 128 KB 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 85th percentile, a 64-KB read is 4× slower than a 32-KB read. The core of the problem is this: if a single page of a large read is blocked by a GC, the entire read cannot complete; as read size increases, the probability of one of the pages being blocked by GC also increases, as we explain later (§3, §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 that are 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 (\( C_1 \ldots C_N \)). A channel connects multiple flash planes; 1–4 planes can be packaged as a single chip (dashed box). A plane contains blocks of flash pages. In every plane, there is a 4-KB register support; 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.
Figure 2: SSD Internals (Section 3).

GC operation (4 main steps): When used-page count increases above a certain threshold (e.g., 70%), a 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 4-KB register support in every plane, page copybacks happen within the GC-ing plane without using the channel [11, 18].

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 implement 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 3a. Here, a single GC (the striped plane) blocks the controller, thus technically all channels and planes are blocked (the bold lines and dark planes). All outstanding I/Os cannot be served (represented by the non-colored I/Os). OpenSSD [4], VSSIM [50], 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 3b), other threads/CPUs can serve other I/Os designated to other channels (the colored 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 [32] and diskSim+SSD [18] adopt this implementation. Commodity SSDs do not come with layout specifications, but from our experiments (§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 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:

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

2. Exploit RAIN parity-based redundancy (§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 (§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 (§4.4).

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

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies

Latencies
Figure 4: TITFlASH Architecture. The figure illustrates our RAIN layout (§4.2), GC-tolerant read (§4.3), rotating GC (§4.4), and GC-tolerant flush (§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).

ping, where an LPN can be mapped to any PPN, is often used to speed-up writes (flexible destination). However, in modern SSDs, write latency issues are absorbed by capacitor-backed RAM (§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.

Given the reasons above, we create a page-level hybrid static-dynamic mapping. The static allocation policies are: (a) an LPN is statically mapped to a plane (e.g., LPN 0 to plane G0C0 in Figure 4), (b) N−1 consecutive LPNs and their parity form a stripe (e.g., 012P012), and (c) the stripe pages are mapped to planes across the channels within one plane group (e.g., 012P012 in G0). Later, we will show how all of these are crucial for supporting GC-tolerant read (§4.3) and rotating GC (§4.4).

The dynamic allocation policy is: inside each plane/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 G0C0 plane).

To prevent parity-channel bottleneck (akin to 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, P012 and P345 are in different channels, in a diagonal fashion. Second, as SSD planes form a 2-dimensional layout (G0C0) with wearout issues (unlike disk’s “flat” LPNs), we need to ensure hot parity pages are spread out evenly. To handle this, we will present dynamic migrations later (§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 (P012). 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 only takes less than 3µs for N≤8 and the additional parity read only takes a minimum of 40+100µs (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 only perform extra reads if \(T_{GCtoComplete}>B×(40+100)\)µs 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 simple policy cuts GC tail latencies effectively and fairly without introducing heavy contention. In the opposing end, a “greedy” approach that always performs extra reads causes high channel contention.

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 (G0C0 and G0C1) both perform GC, reading LPNs 0–2 will be delayed. The core issue is: one parity 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 ensures that at most one plane in each plane group can perform one GC at a time. Concurrent GCs in different plane groups are still allowed (e.g., one in each G1 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 \((N-1) \times T_p\) (the GC duration). During this period, when 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 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 violation, but also due to reduced reliability and the more extra I/Os generated for small reads by GTR (§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 flash industry heavily employs capacitor-backed RAM as a durable write buffer (or “cap-backed RAM” for short) [14]. 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” [10].

We adopt cap-backed RAM to 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\(_{345}\) can be evicted from RAM to flash while page 5’ eviction is delayed until the destination plane finishes the GC. With proven 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 size is 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 \(P_{678’}\), which requires 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 \(P_{678’}\) 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 [35], our method adds GC tail tolerance.

### 5 Implementation

This section describes our implementations of ttFLASH, which is available on our website [1].

- **ttFlash-Sim (SSDSim):** To facilitate accurate latency analysis at the device level, we first implement ttFLASH in SSDSim [32], a recently-popular simulator whose accuracy has been validated against a real hardware platform. 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 port ttFLASH to VSSIM, a QEMU/KVM-based platform that “facilitates the implementation of the SSD firmware algorithms” [50]. VSSIM emulates NAND flash latencies on RAM disk. Unfortunately, VSSIM’s implementation is based on 5-year old QEMU-v0.11 IDE interface, which only delivers 10K 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 port ttFLASH features to VSSIM++, which we refer as ttFlash-Emu, for a total of 869 LOC of changes.
• Other attempts (OpenSSD and LightNVM): We attempted implementing ttFlash on real hardware platforms (2011 Jasmine and 2015 Cosmos OpenSSD boards [4]). After a few months trying, we hit many limitations of OpenSSD: single threaded (no pthread support), single logic for all channels (cannot control channel queues), no accessible commands for data transfer from flash RAM to host DRAM (preventing parity regeneration), no support for wall-clock time (preventing GC time prediction), inaccessible request queues and absence of GC queues (OpenSSD is whole-blocking). 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). Nevertheless, 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).

Finally, we also investigated the LightNVM (Open-Channel SSD) QEMU test platform [16]. LightNVM [21] is an in-kernel framework that manages OpenChannel SSD (which exposes individual flash channels to the host, akin to Software-Defined Flash [44]). 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 (§6.1), delivers more stable latencies than the state-of-the-art preemptive GC (§6.2) and other GC optimization techniques (§6.3), and does not significantly increase P/E cycles beyond the RAIN overhead (§6.4).

Workloads: We evaluate two implementations: ttFlash-Sim (on SSDSim) and ttFlash-Emu (on VS-SIM++), as described in Section 5. For ttFlash-Sim evaluation, we use 6 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, 34]. By default, for each trace, we chose the busiest hour (except the 6-minute TPCC trace). For ttFlash-Emu evaluation, we use filebench [2] with six personalities as listed in the x-axis of Figure 8.

Hardware parameters: For ttFlash-Sim, we use the same 256-GB parameter values provided in Table 1 with 64 MB cap-backed RAM and a typical device queue size of 32. ttFlash-Emu uses the same parameters but its SSD capacity is only 48 GB (limited by the machine’s DRAM). We use a machine with 2.4GHz 8-core Intel Xeon Processor E5-2630-v3 and 64-GB 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 are fast and stable as all writes are absorbed by cap-backed RAM (§4.5). As shown in Figure 5, the base approach (“Base” = the default SSDSim with channel-blocking and its most-optimum FTL [32] 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 (§4.1), +GTR (§4.3), and +RGC (§4.4), sig-
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 90\textsuperscript{th} 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 times faster.

- **TTFlash vs. NoGC:** To characterize the benefits of TTFlash’s tail latencies, we compare TTFlash to a perfect “no-GC” scenario (“NoGC” = TTFlash without GC and with RAIN). In NoGC, the same workload runs without any GC work (with a high GC threshold), thus all I/Os observe 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 99–99.9\textsuperscript{th} percentiles, TTFlash’s slowdowns are only 1.00 to 1.02 times. Even at 99.9\textsuperscript{th} percentile, TTFlash’s slowdowns are only 1.0 to 2.6 times. In comparison, Base suffers from 5.6–138.2 times slowdowns between 99–99.9\textsuperscript{th} percentiles (as obvious in Figure 5); for readability, NoGC lines are not plotted in the 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. As 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, there are 2–7% of blocked I/Os that 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 0.003–0.05% of 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 §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 to device-level latencies) and the percentage of GC-blocked reads measured inside ttFlash-Emu. We do not plot 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 work optimize GC, but does not eliminate its impact. One industry standard in eliminating (“postponing”) GC impact is preemptive GC [9]. We implement preemptive GC in SSDSim based on existing literature [40]. 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 9a 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 preemptive GC. The reason is that preemptive GC must incur a delay from waiting for

---

**Figure 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).** The numbers above represent the slowdown ratio of TTFlash read latencies compared to NoGC at high percentiles. For example, in DTRS, at 99.9\textsuperscript{th} percentile, TTFlash’s read latency is only 1.24 times slower than NoGC’s read latency.

<table>
<thead>
<tr>
<th>Percentile</th>
<th>Base</th>
<th>DAP</th>
<th>DTRS</th>
<th>Exch</th>
<th>LMBE</th>
<th>MSN</th>
<th>TPCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.9\textsuperscript{th}</td>
<td>1.00x</td>
<td>1.01x</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
</tr>
<tr>
<td>99\textsuperscript{th}</td>
<td>1.00x</td>
<td>1.02</td>
<td>1.02</td>
<td>1.02</td>
<td>1.02</td>
<td>1.02</td>
<td>1.02</td>
</tr>
</tbody>
</table>

**Figure 7: %GC-blocked read I/Os.** The figure above corresponds to the results in Figure 5. The bars represent the ratio (in percent) of read I/Os that are GC-blocked (bottom bar) and queue-blocked (top bar) as explained in §6.1. “All” implies PB+GTR+RGC (please see Figure 5’s caption).
the block erase (up to 2 ms) or the current page copyback to finish (up to 800 μs delay), mainly because the finest-grained preemption unit is a page copyback (§3). 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 (§5).

Most importantly, ttFlash does not postpone GC indefinitely. In contrast, preemptive GC piles up GC impact to the future, with the hope that there will be idle time. However, with a continuous I/O stream, at one point, the SSD will hit a GC high water mark (not enough free pages), which is when preemptive GC becomes non-preemptive [40]. To create this scenario, we run the same workload but make SSDSim GC threshold hit the high water mark. Figure 9b 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 perform a similar experiment as in the Semi-Preemptive GC paper [40, §IV]. We re-rate DTRS I/Os by 10× and re-size them by 30×, in order to reach the high GC water mark (which we set to 75% to speed up the experiment). Figure 9c shows the timeline of observed latencies with ttFlash and preemptive GC. We also run a synthetic workload with continuous I/Os to prevent idle time (Figure 9d); the workload generates 28-KB I/Os (full-stripe) every 130μs with 70% read and 30% write. Overall, Figures 9c–d highlight that preemptive GC creates backlogs of GC activities, which will eventually cause SSD “lock-down” when page occupancy reaches the high watermark. On the other hand, ttFlash can provide stable latencies without postponing GC activities indefinitely.

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

Figure 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 (§5).

Figure 9: ttFlash vs. Preemptive GC. The figures are explained in Section 6.2.

6.3 ttFlash vs. GC Optimizations
GC can be optimized and reduced with better FTL management, special coding, novel write buffer scheme or SSD-based log-structured file system. For example, in comparison to base approaches, Value locality reduces erase count by 65% [29, Section 5], flash-aware RAID by 40% [33, Figure 20], BPLRU by 41% [35, Section 4 and Figure 7], eSAP by 10-45% [37, Figures 11–12], F2FS by 10% [39, Section 3], LARS by 50% [41, Figure 4], and FRA by 10% [42, Figure 12], SFS by 7.5× [43, Section 4], WOM codes by 33% [48, Section 6].

Contrary to these efforts, our approach is fundamentally different. We do not focus in reducing the number of GCs, but instead, we eliminate the blocking nature of GC operations. With reduction, even if GC count is reduced by multiple times, it only makes GC-induced tail latencies shorter, but not disappear (e.g., as in Figure 5). Nevertheless, the techniques above are crucial in 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% of additional P/E cycles (in 4 out of 6 workloads), which mainly comes from RAIN; as we use N=8, there are roughly 15% (1/7) more writes in
minimum, from one parity write for every seven \((N-1)\) consecutive writes. The exceptions are 53% of additional P/E cycles in MSNFS and TPCC, which happen because the workloads generate many small random writes, causing one parity write for almost every write. For this kind of workload, large buffering does not help. Overall, higher P/E cycles is a limitation of \text{TtFLASH}, but also a limitation of any scheme that employs RAIN.

### 6.5 \text{TtFLASH} vs. No RAIN

Earlier, in Figure 6, we show that \text{TtFLASH} has about the same average latencies as \text{NoGC} (\text{TtFLASH} without GC and with RAIN). In further experiments (not shown due to space), we also compare \text{TtFLASH} to \text{“NoGC”} (i.e., \text{Base} without GC and without RAIN). We observed \text{TtFLASH}'s average latencies are 1.09–1.33 \times of \text{NoGC}'s. The RAIN-less \text{NoGC} is faster because it can utilize all channels. This is a limitation of \text{TtFLASH}; that is, as \text{TtFLASH} (or any SSD that) employs RAIN, the channels experience a slight contention. In Figure 4 for example, reading LPNs 0–3 will incur contention on channel-0 (from LPNs 0 and 3). In a RAIN-less setup, the same read will utilize all four channels.

### 6.6 \text{TtFLASH} under Write Bursts

\text{TtFLASH} can circumvent GC blocking when rotating GC is enforced (§4.4). A limitation of \text{TtFLASH} is that under heavy write bursts, multiple GCs per plane group must be allowed to keep the number of free pages stable. Figure 11a shows the limit of our 256GB drive setup (Table 1) with \text{N}=8. As shown, at 6 DWPD (55 MB/s), there is almost no GC-blocked reads, hence tiny tail latencies. 1 DWPD (“Drive Writes Per Day”) implies 256GB/8hours (9.1 MB/s) of writes; we generously use 8 hours to represent a “Day” (Appendix A). However, at 7 DWPD (64 MB/s), \text{TtFLASH} exhibits some tail latencies, observable at the 90th percentile. We emphasize that this is still much better than the \text{Base} approach, where the tail latencies are observed starting at 20th percentile (not shown). We also believe that such intensive writes are hopefully rare; for 3-5yr lifespans, modern MLC/TLC drives must conform to 1-5 DWPD [17]. Figure 11b shows that if we force 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. Thus, to keep the number of free pages stable, under write bursts, we must allow multiple GCs to happen per plane group (the “RGC, 7DWPD” line).

### 7 Limitations and Discussions

We now summarize the limitations of \text{TtFLASH}. First, \text{TtFLASH} depends on RAIN, hence the loss of one channel per \text{N} channels (as evaluated in §6.5). Increasing \text{N} will reduce channel loss but cut less tail latencies under write bursts (Appendix A). Under heavy write bursts, \text{TtFLASH} cannot cut all tails (as evaluated in §6.6 and discussed in Appendix A). Finally, \text{TtFLASH} requires intra-plane copybacks, skipping ECC checks, which requires future work as we address below.

- **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 (\text{TtFLASH} does not modify this property). Due to increasing bit errors, it is suggested that ECC checking runs more frequently, for example, by forcing all background GC copyback-ed pages read out from the plane and through the controller, albeit reduced performance.

\text{TtFLASH}, however, depends on \text{intra-plane} copybacks, which implies no ECC checking on copyback-ed pages, potentially compromising data integrity. A simple possible solution to compensate this problem is periodic idle-time scrubbing within the SSD, which will force flash pages (user and parity) 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 2 GB/s 512-GB client SSD can be scrubbed under 5 minutes. Second, scrubbing can be easily optimized, for example, by only selecting blocks that are recently GC-ed or have higher P/E counts and history of bit flips, which by implication can also reduce read disturbs. Third, periodic background operations can be scheduled without affecting...
foreground performance (a rich literature in this space [19]). However, more future work is required to evaluate the ramifications of background ECC checks.

- **Wear leveling (via horizontal shifting and vertical migration):** Our static RAIN layout (§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 \( S_0 \) \{012P_{012}\}, LPN 1 is more frequently updated than the rest, the planes of LPN 1 and \( P_{012} \) will wear out faster than the other planes in the same group. (2) There is write imbalance across the stripes. For example, if stripes in group \( G_0 \) (e.g., stripe \{012P_{012}\}) are more frequently updated than stripes in other groups, then the planes in \( G_0 \) will wear out faster.

  The two wear-out problems above can be fixed by *dynamic* horizontal shifting and vertical migration, respectively. With horizontal shifting, we can shift the parity locations of stripes with imbalanced hot pages. For example, \( S_0 \) can be mapped as \{12P_{012}\} across the 4 planes in the same group; LPN 1 and \( P_{012} \) will now be directed to colder planes. With vertical migration, hot stripes can be migrated from one plane group to another (“vertically”), 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., \( S_0 \) is in group \( G_0 \) with \( P_{012} \) behind channel \( C_3 \)). 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, the LPN-PPN translation performs as usual, as the stripe mapping stays static (the common case). If the bit is set (the rare case in rare workloads), 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 \( C_j \).

8 Related Work

We now discuss other works related to TTFLASH.

- **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. First, isolation (e.g., OPS isolation [36]) only isolates a tenant (e.g., sequential) from another one (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 TTFLASH is user-agnostic. Second, GC optimization, which can be achieved by better page layout management (e.g., value locality [29], log-structured [23, 39, 43]) 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* [22, 40, 47]. We already discussed the limitations of preemptive GC (§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 SSD internal redundancy to eliminate GC tail latencies. There are other works that leverage redundancy in flash array (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 [6, 12]. In literature, we are aware of only four major ones: eSAP [37], PPC[33], FRA [42] and LARS [41]. These efforts, however, mainly concern about write optimization and wear leveling in RAIN but do not leverage RAIN to eliminate GC tail latencies.

  **Flash array:** TTFLASH works within a single SSD. In the context of SSD array, we are aware of two published techniques on GC tolerance: Flash on Rails [46] and Harmonia [38]. Flash on Rails [46] eliminates read blocking (read-write contention) with a ring of multiple drives where 1–2 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 [38], 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 [19], file system allocation policies [31], block-level I/O schedulers [49], and disk/SSD hardware-level defects [26, 27, 30]. An earlier work addresses load-induced tail latencies with RAID parity [20]. 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 cable 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 combina-
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.

10 Acknowledgments

We thank Sam H. Noh (our shepherd), Nisha Talagala, and the anonymous reviewers for their tremendous feedback. This material is based upon work supported by the NSF (grant Nos. CCF-1336580, CNS-1350499, CNS-1526304, CNS-1405959, and CNS-1563956) as well as generous donations from EMC, Google, Huawei, NetApp, and CERES Research Center.

A Proof Sketch

Limitation of maximum stripe width ($N$): We derive the maximum stripe width allowable ($N$) such that rotating GC (§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 $(N-1)\times T_{gc}$). We argue that $N$ should be at least 8 for a reasonable parity space overhead (12.5%); a lower stripe width will increase space overhead. Below we show that $N=8$ is safe even under intensive write. Table 3 summarizes our proof, which is based on a per-plane, per-second analysis. We first use concrete values and later generalize the proof.

• Table 3a: We use typical parameters: 4-KB page ($S_{page}$), 4-KB register size ($S_{reg}$), 25% valid pages ($\%_{validP g}$), 840\,$\mu$s of GC copyback time per page ($T_{copyback}$), and 900\,$\mu$s of user write latency per page ($T_{usrWrt}$). Due to intensive copybacks (tens of ms), the 2ms erase time is set to “0” for proving simplicity.

• Table 3b: Each plane’s bandwidth ($BW_{pl}$) defines the maximum write bandwidth, which is 4.5 MB/s, from the register size ($S_{reg}$) divided by the user-write latency ($T_{usrWrt}$); all writes must go through the register.

• Table 3c: With the 4.5 MB/s maximum plane bandwidth, there are 1152 pages written per second (\#$W_{pg/s}$), which will eventually be GC-ed.

• Table 3d: Intensive writes imply frequent overwrites; we assume 25% valid pages ($\%_{validP g}$) to be GC-ed, resulting in 288 pages copybacked per second (\#$C_{Bpg/s}$). The $\%_{validP g}$ can vary depending on user workload.

• Table 3e: With 288 page copybacks, the total GC time per second per plane ($T_{gc/s}$) is 242 ms.

• Table 3f: $N$ planes in each group must finish their GCs in rotating manner. As each plane needs $T_{gc}$ time every second, the constraint is: $N<1/T_{gc}$. 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 3g-j). To show this, below we first generalize the proof.

• Table 3g: We combine all the equations above to the equation in Table 3h, which clearly shows that $N$ goes down if $BW_{pl}$ or $\%_{validP g}$ is high. Fortunately, we find that the constant 4.5 MB/s throughout ($BW_{pl}$) in Table 3b is unrealistic in practice, primarily due to limited SSD lifetime. MLC block is only limited to about 5000–10,000 erase cycles and TLC block 3000 erase cycles. To ensure multi-year (3–5) lifespan, users typically conform to the Drive Writes Per Day (DWPD) constraint (1–5 DWPD for MLC/TLC drives) [17].

• Table 3h: Let us assume a worst-case scenario of 5 DWPD, which translates to 5 $PW_{PD}$ (planes write per day) per plane. To make it worse, let us assume a “day” is 8 hours. We set plane size ($S_{pl}$) to 4 GB (§3).

• Table 3i: The more realistic parameters above suggest that a plane only receives 0.7 MB/s (4GB/5/8hrs), which is 6.5× less intense than the raw bandwidth (3b).

• Table 3j: If we plug in 0.7 MB/s to the equations in Table 3c-e, the GC time per plane ($T_{gc}$) is only 38 ms, which implies that $N$ can be as large as 26.

In conclusion, $N=8$ is likely to always satisfy rotating GC in practice. In 32-channel SSD, $N=32$ can violate rotating GC; GC-tolerant read (§4.3) cannot always cut the tails. Overall, Table 3g defines the general constraint for $N$. We believe the most important value is $BW_{pl}$. The other parameters relatively stay the same; $S_{page}$ is usually 4 KB, $\%_{validP g}$ 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 (§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., 2 GB/s) the minimum RAM size needed is only 256 MB.
References


[33] Sookjun Im and Dongkun Shin. Flash-Aware RAID Techniques for Dependable and High-Performance Flash Memory SSD. IEEE Transactions on Computers (TOC), 60(1), October 2010.


