GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices

Niv Dayan
Harvard University
dayan@seas.harvard.edu

Philippe Bonnet
IT University of Copenhagen
phbo@itu.dk

Stratos Idreos
Harvard University
stratos@seas.harvard.edu

ABSTRACT

The volume of metadata needed by a flash translation layer (FTL) is proportional to the storage capacity of a flash device. Ideally, this metadata should reside in the device's integrated RAM to enable fast access. However, as flash devices scale to terabytes, the necessary volume of metadata is exceeding the available integrated RAM. Moreover, recovery time after power failure, which is proportional to the size of the metadata, is becoming impractical. The simplest solution is to persist more metadata in flash. The problem is that updating metadata in flash increases the amount of internal IOs thereby harming performance and device lifetime.

In this paper, we identify a key component of the metadata called the Page Validity Bitmap (PVB) as the bottleneck. PVB is used by the garbage-collectors of state-of-the-art FTLs to keep track of which physical pages in the device are invalid. PVB constitutes 95% of the FTL's RAM-resident metadata, and recovering PVB after power fails takes a significant proportion of the overall recovery time. To solve this problem, we propose a page-associative FTL called GeckoFTL, whose central innovation is replacing PVB with a new data structure called Logarithmic Gecko. Logarithmic Gecko is similar to an LSM-tree in that it first logs updates and later reorganizes them to ensure fast and scalable access time. Relative to the baseline of storing PVB in flash, Logarithmic Gecko enables cheaper updates at the cost of slightly more expensive garbage-collection queries. We show that this is a good trade-off because (1) updates are intrinsically more frequent than garbage-collection queries to page validity metadata, and (2) flash writes are more expensive than flash reads. We demonstrate analytically and empirically through simulation that GeckoFTL achieves a 95% reduction in space requirements and at least a 51% reduction in recovery time by storing page validity metadata in flash while keeping the contribution to internal IO overheads 98% lower than the baseline.

1. INTRODUCTION

In recent years, storage devices based on NAND flash memory such as eMMCs (embedded multimedia cards) and SSDs (solid state drives) have become widely used for various applications. Relative to hard disk drives, the benefits of NAND flash include superior read/write performance, shock-resistance, noiselessness, and lower power consumption.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2915219

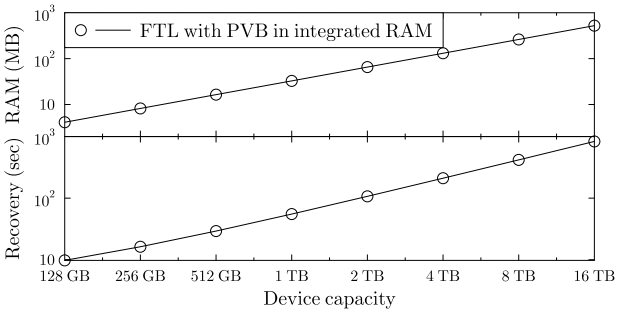


Figure 1: RAM-resident FTL metadata and recovery time are increasing unsustainably as device capacity grows.

The Need for a Flash Translation Layer. Flash memory is organized into thousands of flash blocks, each of which contains hundreds of atomic storage units called pages. A critical property of flash is that a page cannot be updated until the block that contains it is erased. This characteristic creates the requirement for a Flash Translation Layer (FTL). When a logical page is updated, its new version is written on a free flash page, and the original flash page is marked by the FTL as invalid. A lot of metadata is needed to support this task. At the very least, this metadata includes a translation table from logical to physical addresses (to keep track of the whereabouts of live logical pages), as well as a bookkeeping of which flash pages are invalid (to later reclaim them through garbage-collection).

Costly Integrated RAM. A small RAM module is integrated into flash devices to store FTL metadata. However, the amount of this integrated RAM is limited due to its costs. In lower-end devices like eMMCs, manufacturers use Static RAM (SRAM) due to its power-efficiency [27]. Today, the cost of SRAM is approximately $5 per megabyte. This limits the amount of metadata that can be stored in integrated RAM to a few megabytes at most. Higher-end flash devices like SSDs typically embed an additional Dynamic RAM (DRAM) module comprising tens to hundreds of megabytes [13]. DRAM is 2 orders of magnitude cheaper than SRAM, but it also contributes to a device's cost by requiring a more sophisticated controller and circuitry. DRAM also increases runtime costs because it is more power-hungry than SRAM.

Scalability Problem. The amount of metadata needed by state-of-the-art FTLs is increasing in proportion to device capacity. This increases their minimum integrated RAM requirement. It also increases the time that it takes such FTLs to recover from power failure, because recovery time is proportional to the amount of metadata that has to be recovered.

A natural solution is to persist metadata in flash. This frees up integrated RAM and also reduces recovery time because flash is non-volatile and so metadata is not lost when power fails. Indeed, state-of-the-art FTLs already store the largest metadata structure, the logical to physical translation table, in flash [22, 26], while caching frequently accessed mapping entries in integrated RAM. This approach has kept the integrated RAM requirement and recovery time acceptable until now.

However, Figure 1 shows that the integrated RAM requirement and recovery time are increasing unsustainably as flash devices continue to grow in capacity. The results are based on an implementation of LazyFTL [22], which is a state-of-the-art FTL (we discuss the figure's derivation in detail in Section 5). In particular, the figure shows that integrated RAM reemerges as a dominant cost for low-end devices at capacities of $\approx$ 128 GB, at which point 4 MB of SRAM are needed. Similarly, recovery time becomes impractical at capacities of $\approx$ 2 TB, at which point recovery takes tens of seconds. Since there already exist 128 GB low-end[1] and 2 TB high-end[2] devices on the market, we are at the point in which these scalability challenges need to be addressed.

In this paper, we identify a key components of the metadata called the Page Validity Bitmap (PVB) as the bottleneck. PVB is used by state-of-the-art FTLs to keep track of which physical pages in the device are invalid. It is updated whenever a flash page becomes invalid, and it is queried by garbage-collection operations to determine which flash pages on a victim block are still valid. PVB accounts for 95% of all RAM-resident metadata and comprises a significant proportion of recovery time.

The simplest solution is storing PVB in flash. The problem is that PVB is frequently updated, so storing it in flash increases write-amplification, the phenomena whereby several internal flash reads and writes take place for every logical write issued by the application. Write-amplification harms throughput. It also wears out the device at a faster rate, since flash blocks have a limited lifetime with respect to the number of times they have each been overwritten.

Flash Devices and Databases. With more and more database systems and installations utilizing flash devices, it is increasingly important to scale flash devices to sustain the growth of very large database applications. At the same time, ensuring quick recovery in the presence of failures as devices grow is an essential property that very large databases require.

The Solution: GeckoFTL. To address these scalability challenges, we propose GeckoFTL. Its central innovation is storing page validity metadata in flash using a new write-optimized data structure called Logarithmic Gecko, where Gecko stands for **G**arbage-**C**ollector. Logarithmic Gecko is similar to an LSM-tree [32] in that it logs updates and later reorganizes them in flash to ensure fast and scalable access time. Relative to the baseline solution of storing PVB in flash, Logarithmic Gecko enables cheaper updates at the cost of slightly more expensive garbage-collection queries. We show that this is a good trade-off because (1) updates are much more frequent than garbage-collection queries to page validity metadata, and (2) flash writes are an order of magnitude more expensive than flash reads. Thus, Logarithmic Gecko reduces the write-amplification generated by a flash-resident PVB by 98% while still enabling a 95% reduction in the integrated RAM requirement. Even for devices that have enough integrated RAM to store PVB, we show that using Logarithmic Gecko reduces recovery time by at least 51% and improves performance as the freed integrated RAM is used to cache a larger proportion of the translation table.

[1]E.g. Samsung KLMDGAWEBD-B031

[2]E.g. Samsung MZ-7KE2T0BW

Since GeckoFTL involves storing more metatadata in flash, a natural question is how to garbage-collect flash-resident metadata. The GeckoFTL garbage-collector answers this by accounting for the different update frequencies of user data and metadata thereby further significantly decreasing the overall write-amplification.

Finally, GeckoFTL removes a significant contention between recovery time and write-amplification. In state-of-the-art FTLs, the flash-resident translation table is updated lazily and in bulk to amortize the cost of updates. Meanwhile, so-called dirty mapping entries for recently updated logical pages are cached in integrated RAM. When power fails, these dirty entries are lost, and the time to recover them is proportional to the number of dirty entries that were in the cache when power failed. To bound recovery time, existing FTLs limit the number of dirty entries in the cache. However, doing so also limits the amount by which updates to the flash-resident translation table can be amortized, so write-amplification increases. GeckoFTL presents a lazy recovery algorithm based on checkpoints that removes this contention between recovery time and write-amplification.

Contributions. Our contributions are as follows.

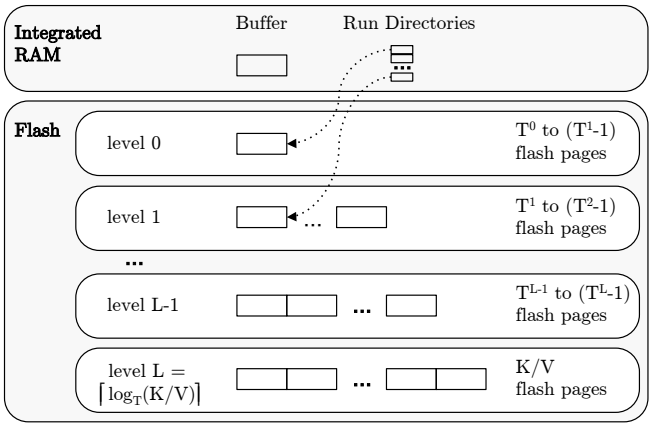
- We show that as flash devices scale, more metadata will need to be stored in flash to keep the integrated RAM requirement and recovery time practical. However, storing metadata naively in flash can significantly increase write-amplification thereby compromising throughput and device lifetime.
- We present GeckoFTL, a page-associative FTL that enables flash devices to scale while keeping the integrated RAM requirement, recovery time, and write-amplification low.
- GeckoFTL uses a novel data structure called Logarithmic Gecko to store page validity metadata in flash. Relative to the baseline solution of storing PVB in flash, Logarithmic Gecko generates 98% less write-amplification while still enabling a 95% reduction in the RAM requirement and at least a 51% reduction in recovery time.
- GeckoFTL keeps garbage-collection overheads low by differentiating between user data and flash-resident metadata.
- GeckoFTL includes a fast recovery algorithm for dirty cached mapping entries that neither requires a battery nor involves a contention between recovery time and write-amplification.

2. BACKGROUND

This section provides the necessary background on flash devices and flash translation layers.

Flash Devices. Flash devices consist of multiple NAND chips, each of which is organized into independent arrays of memory cells. An array is a flash block, and a row within the array is a flash page. A flash page typically stores 4-16 KB and a block typically contains 64-256 pages.

Idiosyncrasies. Flash memory is subject to several idiosyncrasies. (1) The minimum granularity of reads and writes is a flash page. (2) Before a flash page can be updated, the block that contains it must be erased. (3) Blocks become increasingly prone to random bit-shifts as a function of the number of erases and rewrites they have endured. Thus, they have a limited lifetime. (4) Writes must take place sequentially within a block to minimize bit-shifts due to electronic side-effects [1]. (5) Page reads and writes have asymmetric costs; they take tens and hundreds of microseconds to execute respectively [30], and this discrepancy is increasing as an industry trend [15]. Moreover, flash writes have an additional cost in that they wear out the device.



Term	Description	E.g.
K	Number of blocks in the flash device	2^{22}
B	Number of pages per block	2^7
P	Size of a flash page in bytes	2^{12}
R	Ratio of logical capacity to physical capacity	0.7
T	Size ratio between runs in Logarithmic Gecko	
L	Number of levels in Logarithmic Gecko	
V	Number of Gecko entries that fit into the buffer	
S	Entry partitioning factor in Logarithmic Gecko	
C	Capacity of LRU cache in GeckoFTL	
δ	Time to write flash page / time to read flash page	

Figure 2: Overview of Logarithmic Gecko, and terms used throughout the paper.

FTL Tasks. A layer of firmware called the Flash Translation Layer (FTL) manages these idiosyncrasies while exposing a simple block device interface to applications. Its principle tasks are performing out-of-place updates to avoid having to erase and rewrite an entire flash block for every page update, and performing wear-leveling to ensure that all blocks age at a similar rate.

Out-of-place updates have three important implications. (1) The physical capacity of the device must be larger than the logical capacity exposed to the application in order to accommodate free and invalid flash pages. This extra physical capacity is called over-provisioned space. (2) A translation table from logical to physical addresses is needed to keep track of the whereabouts of live logical pages. (3) A garbage-collector is needed to free space that is occupied by invalid pages.

FTL Metadata. The FTL relies on metadata to achieve these tasks. There is an integrated RAM module in flash devices whereon some of this metadata can be stored to enable fast access. However the amount of integrated RAM is limited by costs. Lower-end devices rely on power-efficient yet expensive SRAM, which limits capacity to few megabytes at most. Higher-end devices embed additional tens to hundreds of megabytes of DRAM, but this increases costs by requiring a more sophisticated controller and circuity as well as higher power-consumption during runtime.

In addition to integrated RAM, every flash page has an adjacent spare area whereon metadata about the page can be stored. The spare area is physically adjacent to its flash page and typically smaller by a factor of 32 [1]. Since a spare area cannot be updated before the underlying block is erased, it is used to store metadata that is relevant for a flash page during one of the page's life-cycles (e.g. current logical address, timestamp of when it was last written, error-correction code, etc).

Translation Table. The largest metadata structure in the FTL is the translation table from logical to physical addresses. The translation table is an associative array, and the value at offset i is the physical address of where logical page i currently resides in flash [16]. Using the terms in Figure 2, a device with physical flash capacity of $K \cdot B \cdot P$ bytes has a translation table of size $4 \cdot K \cdot B \cdot R$ bytes[3], which we denote as TT. For example, a 2 TB flash device described by the example values in Figure 2 has a 1.4 GB translation table. how to implement the translation table under integrated RAM constraints has been at the heart of FTL design for two decades [27].

[3]Each physical address is 4 bytes.

Early FTL designs tackled this problem by increasing the mapping granularity from a page to a block [27, 14, 9, 10, 21, 23, 33]. This allows using one mapping entry per block rather than per page thereby reducing the size of the translation table by a factor of $\approx B$. The problem with these block-associative FTLs is that continuously reorganizing adjacent logical pages into the same flash block dramatically increases write-amplification, especially for update patterns that are random in the logical address space.

On the other hand, state-of-the-art FTLs [22, 26, 24, 18] store a page-associative translation table in flash while storing frequently accessed mapping entries in an LRU cache in integrated RAM. The flash pages that store the translation table are called translation pages, and each of them contains a contiguous range of mapping entries. Translation pages are also updated out-of-place, and so a Global Mapping Directory (GMD) is needed in integrated RAM to keep track of the most recent version of each translation page [22, 26]. The size of GMD is $(4 \cdot TT)/P$. Thus, for the 2 TB device described in Figure 2, GMD is 1.4 MB.

Recovering Dirty Entries. The flash-resident translation table is updated lazily and in bulk to amortize the cost of updates [16]. Meanwhile, dirty mapping entries for recently updated logical pages are stored in the LRU cache in integrated RAM. When power fails, these dirty mapping entries are lost and must be recovered to keep track of live user data. The brute-force recovery approach is to scan the spare areas of all flash pages in the device and create a dirty mapping entry for any page with user data that was updated after the last time its corresponding translation page was updated [23]. However, this approach is impractical for very large flash devices as it involves $K \cdot B$ spare area reads, which amount to $\approx$ 26 minutes[4] for the example values in Figure 2. An alternative is using a battery to synchronize all dirty mapping entries with the flash-resident translation table before power runs out [22]. However, a battery increases manufacturing costs.

The problem of how to efficiently recover dirty mapping entries without a battery is open, but the best known approach is to break it into two subproblems [26]: (1) identifying and recreating mapping entries for all recently updated logical pages (this should be done quickly while ensuring that we do not miss any unsynchronized logical pages), and (2) efficiently synchronizing these mapping entries with the flash-resident translation table. If step (2) is done before normal operation resumes, then it elongates recovery time by $\min(C, \frac{TT}{P})$ page reads and writes. When $C > \frac{TT}{P}$, this amounts

[4]A spare area read takes $100/32 \approx 3$ μs since reading a page takes ≈ 100 μs [15] and a spare area is 32 times smaller than a page [1].

to ≈ 7 minutes[5] for the values in Figure 2. To bound recovery time, existing approaches restrict the number of dirty entries in the cache [26, 18]. The problem is that this also limits the amount by which updates to the flash-resident translation table can be amortized, and so write-amplification increases.

PVB. A page-associative FTL must maintain a Page Validity Bitmap (PVB) to keep track of which flash pages are invalid. PVB is structured such that bits that correspond to pages on the same block are adjacent. Whenever a flash page is invalidated, the FTL shifts the corresponding bit in PVB from 0 to 1. During a garbage-collection operation, the FTL queries PVB to determine which flash pages are still valid and need to be migrated before erasing the victim block. When the victim block is erased, all bits in PVB corresponding to pages on the erased block are set to 0.

Scalability of PVB. PVB contains one bit for each flash page in the device, and so its size is $\frac{B \cdot K}{8}$ bytes. For the example values in Figure 2, PVB comprises 64 MB. Thus, the RAM requirement for PVB is roughly 45 times larger than the RAM requirements for GMD. This makes PVB the primary bottleneck in terms of integrated RAM. Moreover, recovering PVB after failure takes a long time as it requires scanning all translation pages (to identify all valid flash pages). This takes $\frac{T \cdot T}{P}$ page reads, which amount to ≈ 36 seconds for the example values in Figure 2. To bound the integrated RAM requirement and recovery time for PVB, the simplest solution is to store it in flash [24]. The problem is that this requires updating one flash page of PVB for every application update. This significantly increases write-amplification.

3. LOGARITHMIC GECKO

Overview. Logarithmic Gecko is a novel write-optimized data structure that replaces PVB by indexing page validity metadata in flash. Figure 2 gives a high-level overview of Logarithmic Gecko as well as terms that we use throughout the paper.

The operations that Logarithmic Gecko supports at its interface are updates and garbage-collection queries (henceforth called GC queries). An update occurs when a flash page is invalidated. A GC query occurs during a garbage-collection operation to determine which flash pages are invalid in the victim block. The design goal of Logarithmic Gecko is to support both operations while minimizing the internal IO overhead.

Buffered Updates. Logarithmic Gecko handles an update by inserting the address of the invalidated flash page into a RAM-resident buffer. The buffer's size is one flash page, and we denote V as the number of entries that fit into it. When the buffer fills up, it is flushed to flash. Thus, V updates lead to one flash write. This is cheaper than PVB, for which V updates lead to V flash reads and writes.

Merge Operations. Logarithmic Gecko stores page validity metadata in substructures in flash called runs, which are organized into L levels. When the buffer is flushed, it is inserted as a run into level 0. Runs are merged in the background as an LSM-tree [32] to keep query time scalable. Whenever there are two runs in the same level, they are merged. The two original runs are then discarded, and the new run may be promoted to the next level based on its size. Thus, a merge operation may continue recursively. As shown in Figure 2, a run is placed in level i if it consists of between T^i and $T^{i+1} - 1$ flash pages, where T is a tuning parameter that controls a trade-off between the costs of updates and GC queries. The logical and physical details of merge operations are given in Section 3.1 and Appendix A respectively.

[5] A page read and write take ≈ 100 μs and ≈ 1 ms respectively [15].

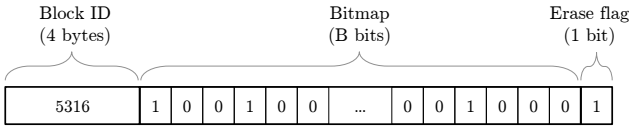


Figure 3: A Gecko entry

Gecko Entries. Logarithmic Gecko stores addresses of invalid flash pages in the buffer and in runs as key-value pairs called Gecko entries (Figure 3). The key of a Gecko entry is a block ID, and the Gecko entries in each run are sorted by this key. The value of a Gecko entry is a bitmap of size B, where the bit at offset i indicates if the physical page at offset i in the block is invalid.

A GC query for key X finds all Gecko entries with key X across all runs and merges their bitmaps using the bitwise OR operator. Similarly, when runs that contain two Gecko entries with the same key are merged, the two Gecko entries are merged in the resulting run by taking the OR product of their bitmaps. These operations are described in detail in Section 3.1.

Run Directories. The RAM-resident run directories in Figure 2 serve as indexes for the flash-resident runs. They contain the physical location of each flash page in each run and the key range of the Gecko entries that it contains. They speed up GC queries by enabling them to only read the flash page in each run that contains the relevant key range. A GC query always traverses the runs from most recently created to least recently created.

Erase Flag. When a block is erased, all of its pages become free. Thus, all Gecko entries for this block that were created before the erase become obsolete and must be ignored during subsequent GC queries. To accomplish this, a simple approach is to find every Gecko entry with the block's key and either erase it or mark it as invalid. However, this would be a costly operation involving $O(L)$ flash reads and writes. To avoid this cost, we add one bit to each Gecko entry called the erase flag (see Figure 3). When a block is erased, we insert a Gecko entry for the block to the buffer and set its erase flag to true. Moreover, we terminate a GC query when it encounters a Gecko entry for the target block with an erase flag set to true, because all Gecko entries on larger runs were created before the last time the block was erased and are therefore obsolete.

Logarithmic Gecko removes obsolete entries during merge operations so that they do not consume space. When two runs that contain Gecko entries with the same key are merged, the entry from the older run is discarded if the entry from the newer run has its erase flag set to true.

All in all, the erase flag allows handling a flash erase through one insertion to the buffer rather than through $O(L)$ flash reads and writes. This makes the performance of Logarithmic Gecko largely independent of the frequency of garbage-collection operations.

In Section 3.1, we describe the basic operations of Logarithmic Gecko in more detail and in Section 3.2 we analyze their costs. In Section 3.3, we introduce a technique called entry-partitioning that makes the performance of Logarithmic Gecko independent of the block size B. In Section 4, we describe how GeckoFTL and Logarithmic Gecko interact. We show how to recover Logarithmic Gecko's run directories and buffer after failure in Appendices C.1 and C.2 respectively.

3.1 Operations

Updates. Logarithmic Gecko's role is to keep track of invalid flash pages. Whenever the FTL identifies an invalid page (as explained later in Section 4.1) it reports it to Logarithmic Gecko via Algorithm 1. The algorithm checks if there is already a Gecko entry in the buffer corresponding to the invalid page's block and creates one if not. It then sets to 1 the bit in the Gecko entry's bitmap at

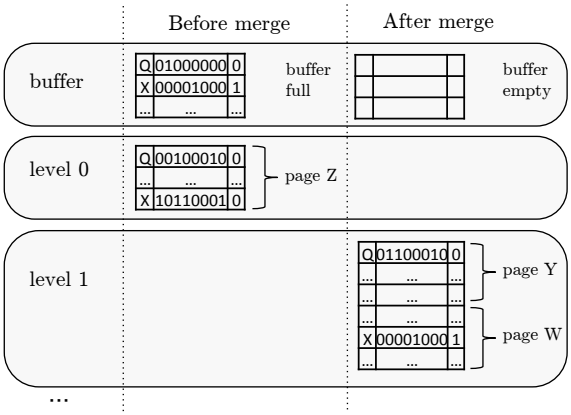


Figure 4: Example of a merge operation.

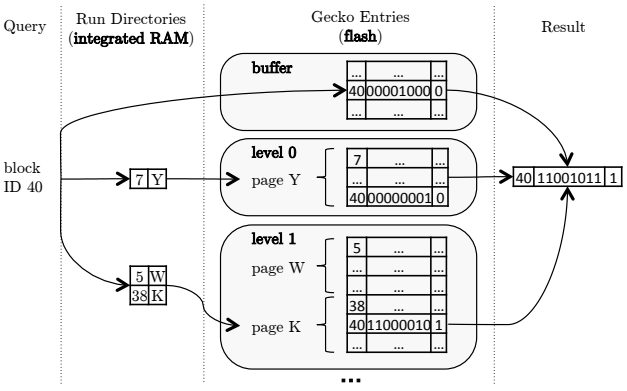


Figure 5: Example of a GC query for key 40.

the offset corresponding to the invalidated page in the block. If the buffer fills up, it is flushed to level 0 in flash.

Erases. When a flash block X is erased, all of its pages become free, and so we must ensure that all Gecko entries created before the block was erased are ignored in subsequent GC queries. This is done by invoking Algorithm 2, which takes as an input a block ID and inserts a Gecko entry with a corresponding key into the buffer with a blank bitmap and the erase flag set to 1.

Merging Runs. When we merge two runs, they may contain Gecko entries corresponding to the same flash block. We denote this as a collision. Collisions are easy to detect during a merge because the runs are sorted by key and we merge them through linear scans. Algorithm 3 is called to handle a collision. If the erase flag of the entry from the more recently created run is set to 1, it means that the other entry was created before the last time the corresponding block was erased, and so it is discarded. Otherwise, the bitmaps for the two entries are merged using the bitwise OR operator to save space.

Figure 4 shows an example of a merge operation. In the example, the buffer has filled up and is merged with the run at level 0. Two collisions occur during this merge for the entries with keys X and Q. For key X, the older entry is discarded since the newer entry's erase flag is set to 1. For key Q, the entries are merged using the bitwise OR operator. The resulting run consists of two flash pages. In this example, T is 2 so the new run is promoted to level 1.

```
Input: physical_address pa, logical_address la
block_id = pa.block_id;
page_offset = pa.page_offset;
if !buffer.contains(block_id) then
    buffer.insert(block_id);
    buffer[block_id].bitmap = blank bitmap;
    buffer[block_id].erase_flag = 0;
buffer[block_id].bitmap[page_offset] = 1;
if buffer is full then
    flush(buffer) ;
```

Algorithm 1: Insert an invalid page address into the buffer.

```
Input: block_id
if buffer.contains(block_id) = false then
    buffer.insert(block_id);
buffer[block_id].bitmap = blank bitmap;
buffer[block_id].erase_flag = 1;
if buffer is full then
    flush(buffer) ;
```

Algorithm 2: Insert the address of an erased block to the buffer.

GC Queries. A garbage-collection operation issues a GC query to determine which flash pages on the victim block are valid and must be migrated before erasing the victim. A GC query first searches the buffer and then the runs from smallest to largest for Gecko entries with keys matching the victim block's ID. It uses the run directories to only perform one flash read per run. Thus, its cost is $O(L)$ flash reads. It finishes once all runs have been searched or when it encounters an entry with an erase flag set to 1.

Figure 5 shows an example GC query for key 40. It first looks in the buffer and finds the first matching entry. It then probes the run directories for the location of the first run (at level 0), reads it from flash, and finds the second matching entry. It then probes the run directory for level 1, identifies flash page K as containing keys that overlap the target ID, reads it, and finds the third matching entry. Since this matching entry's erase flag is set to 1, the GC query terminates. Although there may be Gecko entries with key 40 in runs at higher levels, these entries must have been created before the last time that block 40 was erased, so they are ignored. The bitmaps of all found Gecko entries are finally merged using the bitwise OR operator.

```
Input: newEntry, oldEntry
if newEntry.erase_flag = 1 then
    return newEntry ;
else
    create resultEntry;
    resultEntry.bitmap = newEntry.bitmap OR oldEntry.bitmap;
    resultEntry.erase_flag = oldEntry.erase_flag;
    return resultEntry ;
```

Algorithm 3: Handling collisions during merge operations.

3.2 Analysis

The number of levels L in Logarithmic Gecko is $\lceil \log_T(\frac{K}{V}) \rceil$, where K is the number of flash blocks in the device and V is the number of Gecko entries that fit into one flash page. The reason is that the largest run consists of K/V pages, and subsequent runs have exponentially decreasing sizes, each by a factor of T. This gives way to a logarithmic number of levels.

Cost per Update. An update to Logarithmic Gecko involves inserting a Gecko entry to the buffer. This entry is later rewritten multiple times during merge operations. Although the overall IO cost of an update is indirect, we can capture it as follows. During a merge operation, each flash write copies V entries to the new run. The cost per entry per merge is thus $O(\frac{1}{V})$ of a flash read and write. Now, if an entry does not become obsolete earlier, then it participates on average in $O(T)$ merges per level and goes

Technique	Update		Garbage-Collection Operation		Integrated RAM
	flash reads	flash writes	flash reads	flash writes	
RAM-resident PVB	0	0	0	0	$O(B \cdot K)$
Flash-resident PVB	1	1	1	0	$O(\frac{B \cdot K}{P})$
Logarithmic Gecko	$O(\frac{T}{V} log_T(\frac{K}{V}))$	$O(\frac{T}{V} log_T(\frac{K}{V}))$	$O(log_T(\frac{K}{V}))$	$O(\frac{T}{V} log_T(\frac{K}{V}))$	$O(\frac{B \cdot K}{P})$

Table 1: Relative to a flash-resident PVB, Logarithmic Gecko offers cheaper updates and more expensive GC queries.

through $O(\log_T(\frac{K}{V}))$ levels before getting merged with the largest run. Thus, the amortized cost for an update is $O(\frac{T}{V} \cdot \log_T(\frac{K}{V}))$ flash reads and writes. Typically, V is much larger than $T \cdot \log_T(\frac{K}{V})$. Thus, the cost of an update is sub-constant (lower than 1); each update costs a small fraction of a flash read and a flash write.

Cost per Garbage-Collection Operation. A garbage-collection operation issues a GC query to Logarithmic gecko. A GC query traverses the levels from smallest to largest, issuing one flash read per level. The cost is $O(\log_T(\frac{K}{V}))$ flash reads.

A garbage-collection operation also inserts one entry to the buffer with the erase flag set to true. As we just saw, the cost of inserting a Gecko entry to the buffer is $O(\frac{T}{V} log_T(\frac{K}{V}))$ flash reads and writes.

Tuning. The parameter T controls a trade-off between the costs of updates and GC queries. The minimum value of T is 2. As T increases, there are less levels and so GC queries become cheaper. However, increasing T also increases the cost of updates, as each entry participates in $O(T)$ merge operations per level. In Section 5, we show how to tune T so as to minimize the overall IO overhead.

Comparison to PVB. Table 1 compares the IO costs and integrated RAM requirement of Logarithmic Gecko, a flash-resident PVB, and a RAM-resident PVB. The RAM-resident PVB has no IO overheads, but its integrated RAM requirement is high. The flash-resident PVB has a similar integrated RAM requirement to Logarithmic Gecko, but it is different in terms of IO costs. With a flash-resident PVB, the cost of an update is one flash read and one flash write while the cost of a GC query is one flash read. Compared to a flash-resident PVB, the cost of an update in Logarithmic Gecko is cheaper while the the cost of a GC query is more expensive. In Section 5, we show that this is a good trade-off because (1) flash reads are cheaper than flash writes, and (2) GC queries happen infrequently relative to updates to page validity metadata.

Space-Amplification. The existence of invalid entries leads to space-amplification in Logarithmic Gecko. However, because the largest run contains one of each entry, and since the runs at smaller levels are exponentially smaller, space-amplification is never beyond a factor of ≈ 2 for any value of T.

3.3 Entry-Partitioning

When a Gecko entry is inserted into the buffer, is contains a lot of space yet little information (i.e. most of the bits are set to 0). This waste of space limits V, the number of Gecko entries that fit into the buffer. As shown in Table 1, V is inversely proportional to the update cost in Logarithmic Gecko. We now show how to decrease the update cost by decreasing the amount of buffer space that is wasted.

Under the current design, $V \approx \frac{P \cdot 8}{key+B}$, where P is the size of a flash page in bytes, *key* is the size of a key of a Gecko entry in bits, and B is the number of page validity bits in a Gecko entry. Thus, the value of V depends on the value of B. In fact, since B tends to increase as flash devices grow in capacity, the dependency of V on B is a scalability problem. Our goal is to maximize V while eliminating its dependence on B.

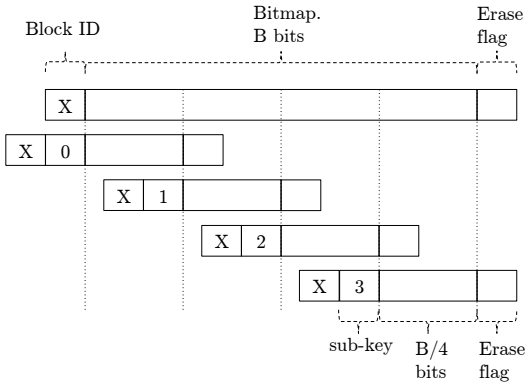


Figure 6: Entry-partitioning with a factor of 4.

To do so, we introduce entry-partitioning. The idea is to partition a Gecko entry into S equal sub-entries, where S is the partitioning factor. Figure 6 shows an example of entry-partitioning where $S = 4$. As shown, each of the partitioned sub-entries has an additional sub-key component to indicate which part of original bitmap it corresponds to. Thus, during an update to Logarithmic Gecko, we only need to insert one sub-entry for the part of the original bitmap that was updated.

An interesting question is how to tune S. To do so, it helps noting that the overall size of Logarithmic gecko in flash is $O(B \cdot K + S \cdot key \cdot K)$ bits, where $O(B \cdot K)$ is for the page validity bits and $O(S \cdot key \cdot K)$ is for the keys. One extreme is setting S to 1. This amounts to no entry-partitioning. In this case, $B \cdot K$ is typically larger than $key \cdot K$, and so the size of Logarithmic Gecko in flash simplifies to $O(B \cdot K)$ bits. The other extreme is setting S to B. In this case, the size of Logarithmic Gecko becomes dominated by the keys and simplifies to $O(B \cdot K \cdot key)$ bits. Thus, the space that Logarithmic Gecko takes up is amplified by a factor of $O(key)$. Space-amplification is harmful because it increases the number of levels in Logarithmic Gecko thereby increasing the costs of both updates and GC queries. It also increases the size of the run directories in integrated RAM. Thus, space-amplification should be bounded.

A good balance is setting $S = \frac{B}{key}$. This removes the dependence of V on B while restricting the size of Logarithmic Gecko to $O(B \cdot K)$ bits. For instance, if *key* is 32 bits and B is 128, than S is set to 4 so that each sub-entry contains a 32 bits key and a 32 bits chunk of the bitmap.

4. GECKO FTL

We now present GeckoFTL, a novel FTL whose design goal is to keep the integrated RAM requirement and recovery time practical for very large flash devices without paying a high price in terms of write-amplification. To do so, GeckoFTL makes three innovations. (1) It offloads page validity metadata to flash using Logarithmic Gecko. (2) It uses a garbage-collection scheme that prevents an increase in write-amplification as more metadata is offloaded to flash. (3) It uses a recovery approach for dirty mapping entries that does not exhibit a contention between recovery time and write-amplification.

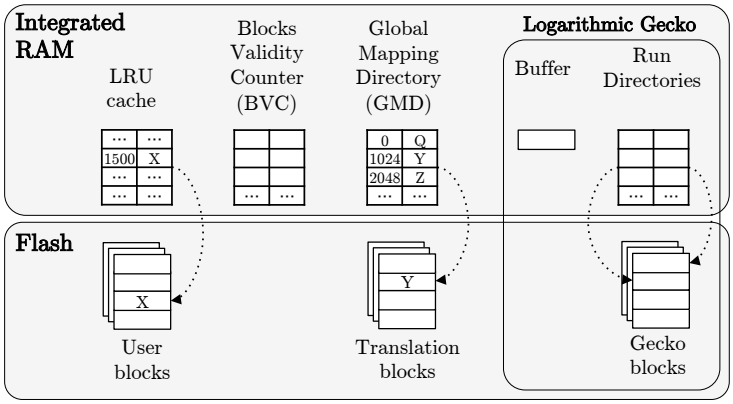


Figure 7: Overview of GeckoFTL.

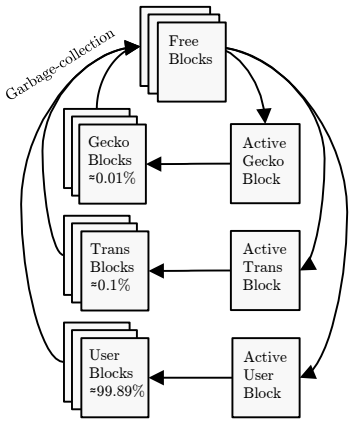


Figure 8: Lifecycle of a flash block.

The structure of the translation table in GeckoFTL is orthogonal to its core design. Thus, we use the simplest and original page-associative flash-resident translation table design first proposed in DFTL [22]: the translation table is stored in flash and a Global Mapping Directory (GMD) in integrated RAM keeps track of the physical location of every translation page. A RAM-resident LRU cache stores recently accessed mapping entries. The overall architecture of GeckoFTL is shown in Figure 7.

We now describe this translation scheme as well as how GeckoFTL lays out user data and metadata in flash. In Section 4.1, we describe how GeckoFTL and Logarithmic Gecko interact. In Sections 4.2 and 4.3, we describe the novel garbage-collection and recovery schemes respectively. We discuss wear-leveling in Appendix D.

Serving Application Reads. Figure 7 illustrates an example where the application issues a read to logical page 1500. To serve it, GeckoFTL first checks if the mapping entry is cached. If not, it finds the appropriate translation page in GMD, reads this translation page from flash, finds the mapping entry for logical page 1500 within the translation page, inserts this mapping entry into the LRU cache, and finally reads the user page.

Serving Application Writes. Now suppose the application updates logical page 1500. GeckoFTL immediately writes the new version of page 1500 on a free flash page Z. It then updates the cached physical address for the mapping entry to Z and marks the mapping entry as dirty using a bit flag.

Synchronization Operations. When the LRU cache fills up, the least-recently-used mapping entry is evicted. If this entry is dirty, a synchronization operation takes place. A synchronization operation identifies all dirty mapping entires in the LRU cache that belong to the same translation page as the evicted entry[6]. GeckoFTL then reads the translation page, updates the dirty cached mapping entries that it found on the translation page, and writes the updated translation page on a free page in flash. It then updates GMD to point to the updated translation page, and marks the cached dirty mapping entries that were included in the operation as clean.

Physical Layout. The flash pages in GeckoFTL have three types: (1) user pages, which contain user data, (2) translation pages, which store the translation table, and (3) Gecko pages, which store Logarithmic Gecko's runs. GeckoFTL separates these pages into different groups of flash blocks based on their types. This gives rise to 3 block groups, as illustrated in Figure 8. Each group has an active block to which updates are made in an append-only manner. When an active block runs out of free space, GeckoFTL allocates to the group a new active block from the pool of free blocks. The overall fraction of translation and Gecko blocks in a device is in the order of 0.1% and 0.01% respectively [7]. Thus, metadata consumes very little space in flash.

Garbage-Collection Operations. When the number of blocks with free space drops below a threshold, garbage-collection is triggered. The garbage-collection mechanism uses the Blocks Validity Counter (BVC) in Figure 7 to choose a victim. BVC keeps track of the number of valid pages in each flash block in the device. Once a victim is chosen, GeckoFTL issues a GC query to Logarithmic Gecko to identify which pages are still valid in the victim. These pages are migrated to a block with free space, and the victim block is erased. Note that garbage-collection migrations are treated like application writes; a dirty cached mapping entry is created for every page that is migrated.

4.1 Invalid Page Identification

We now show how Logarithmic Gecko efficiently identifies invalid pages during runtime. To do so, we denote a mapping entry as e = (logical_address, physical_address). As we just saw, when a logical page X is updated, a mapping entry $e_{cache} = (X,Z)$ is created for it and inserted into the cache. This entry is marked as dirty because the corresponding mapping entry $e_{flash} = (X,Y)$ in flash is now pointing to a before-image Y. Our goal is to guarantee that the invalid flash page Y is eventually identified and reported to Logarithmic Gecko. The greedy solution is to immediately read the translation page that contains e_{flash}. However, this would cost one extra flash read for every application write. We should avoid this extra cost. To do so, we augment synchronization operations to also lazily identify invalid pages.

UIP Flag and Synchronization Operations. We add one bit to each cached mapping entry called the Unidentified Invalid Page (UIP) flag. The UIP flag for entry $e_{cache} = (X,Z)$ is set to true if there is a flash page Y that contains a before-image of logical page X that has still not been identified and reported to Logarithmic Gecko as invalid. During a synchronization operation, before we replace $e_{flash} = (X,Y)$ by $e_{cache} = (X,Z)$, we report Y as invalid to Logarithmic Gecko if the UIP flag of e_{cache} is set to true.

Application Reads. When an application read fetches a mapping entry from the flash-resident translation table and inserts it

[6]The LRU cache is implemented as a tree to enable efficient range queries for mapping entries on a particular translation page.

[7] Assuming X mapping entries fit into each translation page, then the fraction of translation pages in the device is $\approx 1/X$. In practice, X is in the order of thousands, so the fraction of translation pages is in the order of 0.1%. Similarly, each Gecko page contains tens of thousands of page validity bits, and so the fraction of Gecko pages is in the order of 0.01%.

into the cache, we set both the dirty and the UIP flags for the new cached entry to false.

Application Writes. When a logical page X is updated by the application, we manage the cache as follows. If the mapping entry for page X is not already cached, we create a cached mapping entry for it and set both its dirty and UIP flags to true. On the other hand, if a mapping entry for page X is already cached, then we immediately report the old physical page as invalid to Logarithmic Gecko. We then set the entry's dirty flag to true (if it is not already) and leave the entry's UIP flag as it is (as there may be another before-image of logical page X that has still not been identified and reported to Logarithmic Gecko).

Garbage-Collection. Since invalid pages are identified lazily, it is possible that there are still unidentified invalid pages (UIPs) on a block when it is garbage-collected. These UIPs must be identified to avoid migrating invalid pages (doing so would overwrite valid pages). To identify such UIPs, we use the following policy. For every physical page Y in a victim block that Logarithmic Gecko reports as valid, we read the spare area of page Y and identify the logical page X last written on it. We then look up logical address X in the LRU cache. If there is a cached mapping entry for logical page X with the UIP flag set to true and with a different physical address than Y, then we know that the physical page Y is a UIP and do not migrate it as a part of the garbage-collection operation.

4.2 Garbage-Collection Victim-Selection

Since GeckoFTL stores metadata in flash, an interesting question is how metadata should be garbage-collected. Existing page-associative FTLs use a greedy victim-selection policy, which always targets the block with the least number of live pages in the device. The intuition is that this should minimize the number of garbage-collection migrations thereby minimizing their IO overhead. However, this intuition is wrong.

Flash-resident metadata is typically updated 2-3 orders of magnitude more frequently than user data. The key insight is that we should wait before garbage-collecting a block that contains frequently updated (hot) data because its valid physical pages will soon be invalidated anyways.

The question is by how much to defer garbage-collecting Gecko blocks and translation blocks. Our answer is to avoid garbage-collecting them altogether. Instead, GeckoFTL waits until all pages in a Gecko block or a translation block have become invalid and only then erases the block. In Section 5, we show that this policy significantly reduces garbage-collection overheads.

4.3 Recovery from Power Failure

When power fails, GeckoFTL must recover all RAM-resident data structures shown in Figure 7. The complete and detailed recovery algorithm is given in Appendix C. In this section, we focus on the bottleneck: recovering dirty cached mapping entries. We show how to quickly identify and recreate mapping entries for all non-synchronized pages. We also make the case for deferring synchronizing the recreated mapping entries with the translation table until after normal operation resumes.

Approach. The cache's capacity is C mapping entries. Thus, there are at most C dirty mapping entries when power fails. Our approach is therefore to recreate mapping entries for the C unique logical pages that were most recently updated. No mapping entry that was dirty before power failed can be outside of this set. To do this, we identify the most recently written user blocks based on timestamps in their spare areas. This takes K spare area reads, one per flash block. We then read the spare areas of these blocks' pages in reverse order starting from the most recently written block. For each new logical address that we encounter, we create a cached mapping entry.

Checkpoints. If the number of frequently updated logical pages before power fails is smaller than C, then some dirty entries can linger by the end of the LRU queue without ever getting evicted and synchronized. If so, the backwards scan must reach far back to find them. To bound the length of the backwards scan, we use runtime checkpoints. A checkpoint is taken every period of C inserts or updates to the LRU cache, and it synchronizes any mapping entry that has been in the LRU cache since the last checkpoint without getting updated itself. The checkpoints bound the backwards scan to $2 \cdot C$ spare area reads, as any logical page updated before the second last checkpoint must have been synchronized or updated again recently enough to be captured by the backwards scan.

In terms of implementation, a checkpoint inserts a dummy entry into the LRU cache called the checkpoint symbol. It then scans the cache's LRU queue from the end backwards until it finds and removes the symbol inserted by the last checkpoint. It synchronizes all dirty mapping entries encountered along the way.

Deferred Synchronization. Once GeckoFTL finishes recreating mapping entries for the most recently updated C unique logical pages, it still does not know which of these mapping entries are dirty. A natural approach is to access the flash-resident translation table to check. However, the cost would be at most $\frac{TT}{P}$ page reads, which amount to ≈ 36 seconds for the example values in Figure 2. Instead, GeckoFTL sets the dirty and UIP flags for all cached entries to true and corrects mistakes after normal operation resumes during regular synchronization operations. This is described in detail in Appendix C.3. Thus, synchronizing the recreated mapping entries does not elongate recovery time, and the penalty is amortized through synchronization operations that would be taking place anyways during normal operation.

5. EVALUATION

In this section, we present a detailed evaluation of GeckoFTL. We first evaluate Logarithmic Gecko in isolation to show how to optimally tune it and to demonstrate that its performance scales well with respect to different architectural parameters of a flash device. We then compare GeckoFTL to existing FTLs and show that it keeps the integrated RAM requirement and recovery time practical for very large flash devices while paying a lower price in terms of write-amplification.

Infrastructure: The FTL in commercial devices is an opaque black box because manufacturers complete with each other based on its design. Although it is possible to benchmark flash devices to determine the efficiency of different IO patterns [6, 7], it is impossible to disentangle the impact of different FTL components on performance let alone modify them. A part of the difficulty is that flash has no moving parts, so reverse-engineering a flash translation layer is impossible because there is nothing to observe (unlike for disks [2]). As a result, all work that we are aware of in this area relies on simulations [3, 22, 26, 24, 18]. Thus, we implemented GekoFTL and a few competitor FTLs within the flash simulation framework EagleTree [11]. This enables capturing the performance characteristics of different FTL components precisely.

Default Configuration: Unless otherwise stated, we simulate a 2 TB device with 4 KB flash pages and 128 pages per block. This is a standard architectural configuration [15, 3]. We set the ratio between the logical and the physical address spaces (a measure of over-provisioning) to 70%, which is common in practice [35]. We set the size of the LRU cache to 4 MB and assume that 8 bytes are needed per cached entry. Thus, the LRU cache accommodates $C = 2^{19}$ mapping entries. We denote the latency ratio between a

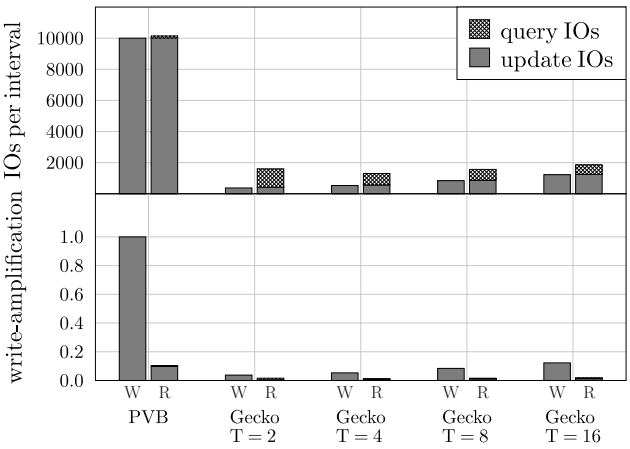


Figure 9: Logarithmic Gecko with a size ratio T of 2 minimizes write-amplification.

page write to a page read as δ and set it to 10, which is typical in practice [15]. We demonstrate throughout the evaluation how the performance of GeckoFTL scales with respect to all of these architectural parameters.

Metrics: We use three metrics to evaluate GeckoFTL: integrated RAM requirement, recovery time, and write-amplification. We measure write-amplification as $WA = (i\_writes + i\_reads/\delta)$ where *i_writes* and *i_reads* are the numbers of internal flash writes and flash reads that take place on average with respect to every logical page update issued by the application. Since write-amplification captures the impact on performance of garbage-collection and of updates to flash-resident metadata, it encompasses the entire impact of GeckoFTL's core design on performance. In terms of application reads, GeckoFTL performs the same as state-of-the-art FTLs [22, 26] because the structure of the translation table is the same. Thus, we do not consider application reads in the experimental workload. Nevertheless, our results are easily generalizable to a mixed workload. In particular, the slowdown factor for application reads throughput is the following expression, where *RA* is read-amplification caused by reading translation pages to fetch mapping entries, and *RW* is the ratio between application reads to application writes in the mixed workload.

$$\text{slowdown factor} = \frac{1}{RA \cdot RW + WA \cdot \delta}$$

5.1 Logarithmic Gecko vs. Flash-Based PVB

We first compare Logarithmic Gecko to a flash-resident PVB. Relative to this baseline, Logarithmic Gecko optimizes updates over GC queries because updates are at the critical path of performance. We now evaluate the impact of this design decision on write-amplification. We also show how Logarithmic Gecko performs under different tunings of the size ratio parameter T.

We evaluate Logarithmic Gecko under an adversarial application workload, which consists of uniformly randomly distributed page updates. Under this workload, Logarithmic Gecko's buffer absorbs as few updates as possible. This maximizes the overhead of merge operations. On the other hand, PVB is workload-insensitive; its IO overhead only depends on the number of page updates rather than the update pattern. Thus, evaluating PVB under this workload is fair.

The top part of Figure 9 shows the average number of internal flash reads and writes caused by updates and GC queries to PVB and Logarithmic Gecko over intervals of 10000 application writes. The bottom part of Figure 9 expresses these overheads in terms of write-amplification. Note that at this point, we do not capture the entire write-amplification in the device, which would also include garbage-collection and synchronization operations. We omit these overheads for now to enable an apples to apples comparison between Logarithmic Gecko and a flash-resident PVB.

As shown in Figure 9, PVB generates high write-amplification because each application update triggers one flash read and one flash write to PVB. Thus, PVB accumulates an overhead of 10000 flash reads and writes per interval. This results in a write-amplification of $\approx 1 + \frac{1}{\delta}$, which in this case is ≈ 1.1. Note that each garbage-collection operation also triggers one GC query to PVB, which involves one flash read. However, since garbage-collection operation are infrequent and since flash reads are inexpensive, the overhead of GC queries to PVB is negligible.

Logarithmic Gecko outperforms PVB under all tunings of the size ratio T. The reason is that the cost of updates is significantly reduced through buffering, and the time to execute GC queries remains fast and scalable because the number of runs is bounded and we only issue one IO per run during a GC query.

Finally, we observe that Logarithmic Gecko performs best when the size ratio T is set to 2. Recall that T controls a trade-off between the IO costs of updates and GC queries; as T increases there are less levels so GC queries become cheaper, but merge operations occur more frequently so updates become more expensive. Since 2 is the lowest value that T can take, the experiment shows that optimizing for updates as much as possible minimizes write-amplification. The reasons are that (1) updates are 1-2 orders of magnitude more frequent than GC queries, and (2) flash writes are an order of magnitude more expensive than flash reads. For all subsequent experiments, we set T to 2.

5.2 Logarithmic Gecko Scaling

We now show how Logarithmic Gecko's performance scales with respect to different architectural parameters of a flash device: the block size B, the number of blocks K, and the level of over-provisioning R.

Block Size and Entry-Partitioning. As the capacity of flash devices grows, the number of pages per flash block B tends to increase. In Section 3.3, we saw that this increases the size of a Gecko entry thereby reducing V, the number of entries that fit into the buffer, which in turn increases the cost of updates. To counter this trend, we proposed a technique called entry-partitioning to make the buffer size V independent of B (thereby also making the cost of updates independent of B). The idea is to partition a Gecko entry into S sub-entries, and to only insert to the buffer the sub-entry that corresponds to the part of the block that contains the page that was invalidated.

Figure 10 shows the impact of entry-partitioning on write-amplification as we vary the block size and the entry-partitioning factor. With no entry-partitioning (i.e. $S = 1$), write-amplification increases proportionally to the block size, whereas a moderate amount of entry-partitioning makes write-amplification independent of the block size. As the partitioning factor continues to increase, however, write-amplification begins to increase. The reason is that the keys of the partitioned entries lead to space-amplification. This increases the number of levels in Logarithmic Gecko thereby also increasing the cost of updates and GC queries. To choose the optimal value of S, the analytical technique at the end of Section 3.3 can be used. Overall, well-tuned entry-partitioning makes write-amplification independent of the block size.

Capacity. Logarithmic Gecko's goal is to enable flash devices to scale to multiple terabytes without incurring a significant performance degradation. However, as device capacity increases, the

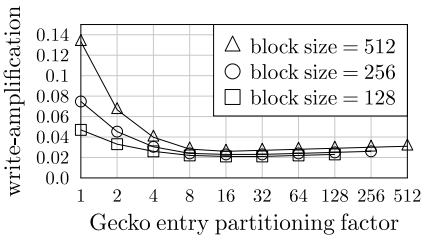


Figure 10: Entry-partitioning makes write-amplification independent of block size B.

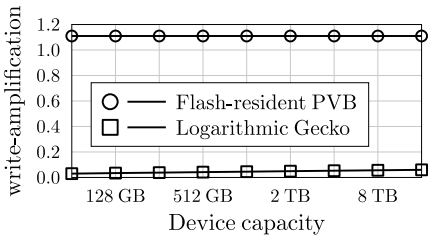


Figure 11: Logarithmic Gecko scales well for terabyte flash devices.

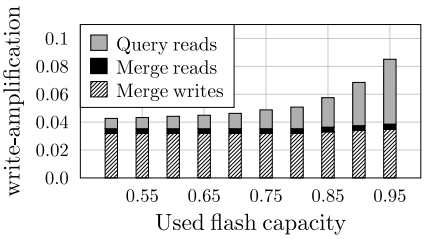


Figure 12: Over-provisioning level does not significantly affect write-amplification.

number of blocks K that are indexed in Logarithmic Gecko increases. This increases the number of levels, which, as we see in Table 1, increases the costs of updates and GC queries. It is therefore interesting to see the rate at which Logarithmic Gecko's performance deteriorates as device capacity increases.

In Figure 11, we compare write-amplification for Logarithmic Gecko and a flash-resident PVB as we increase the number of blocks K. For Logarithmic Gecko, write-amplification increases slowly with respect to device capacity because the costs of updates and GC queries are logarithmic with respect to the number of blocks K. On the other hand, for a flash-resident PVB, write-amplification is independent of device capacity since the cost of updates and GC queries each involve a fixed number of IOs. An interesting question is when the two curves cross. For this to happen, capacity has to increase by a large factor of $\approx 2^{100}$. Thus, Logarithmic Gecko outperforms PVB for any foreseeable device capacity.

Over-Provisioning. Since Logarithmic Gecko optimizes updates at the cost of more expensive GC queries, it is interesting to measure performance as the ratio between updates and GC queries changes. To do so, we vary over-provisioning, which affects the frequency of garbage-collection operations. As over-provisioning decreases, there are more live pages per flash block on average. Thus, garbage-collection victim blocks have more valid pages on average, so each garbage-collection operation reclaim less free space on average. As a result, garbage-collection operations take place more frequently relative to application writes.

In Figure 12, we vary the amount over-provisioning, expressed in terms of R, the ratio between the logical and the physical address spaces. As expected, the number of flash reads due to GC queries increases. Although more GC queries take place, the overall increase in write-amplification is low because flash reads are an order of magnitude cheaper than flash writes. Overall, Logarithmic Gecko works well for any reasonable level of over-provisioning.

5.3 GeckoFTL Vs. Existing FTLs

We now compare GeckoFTL to four state-of-the-art page-associative FTLs: DFTL [22], LazyFTL [26], μ-FTL [24] and Indexed-Based-FTL (IB-FTL) [18]. We compare these FTLs in terms of overall integrated RAM requirement, recovery time, and write-amplification. We show that GeckoFTL keeps the integrated RAM requirement and recovery time practical for very large flash devices without relying on a battery and while paying a lower price in terms of write-amplification than the other FTLs.

Competing FTLs. The FTLs we use differ in two critical ways from one another: (1) in how they store page validity metadata, and (2) in how they recover dirty cached mapping entries. In terms of page validity metadata, DFTL [22] and LazyFTL [26] both use a RAM-resident PVB, whereas μ-FTL uses a flash-resident PVB. IB-FTL logs the addresses of invalidated pages in flash in a page validity log (PVL) while maintaining a linked list of pointers between log entries of invalidated pages on the same block, where the first pointer for each chain is stored in integrated RAM [18]. We explain IB-FTL in detail in Appendix E.

In terms of recovery, DFTL and μ-FTL use a battery to recover dirty cached mapping entries. On the other hand, LazyFTL and IB-FTL restrict the number of dirty cached entries during runtime thereby navigating a trade-off between recovery time and write-amplification. In our experiments, we set the proportion of the cache that stores dirty mapping entries for LazyFTL and IB-FTL to 10% of C, the size of the cache.

(1) Integrated RAM Comparison. To compare the FTLs in terms of integrated RAM requirement, we modeled the sizes of their different data structures using the formulas in Section 2 and Appendix B. The top part of Figure 13 shows the amount of integrated RAM taken up by different data structures for each FTL. Note that the top part of Figure 1 in the introduction is simply the total integrated RAM requirement of LazyFTL under different device capacities.

DFTL and LazyFTL have the largest integrated RAM footprint because they both use a RAM-resident PVB. GeckoFTL, μ-FTL and IB-FTL avoid this overhead by storing page validity metadata in flash and instead only store BVC in integrated RAM to keep track of the number of valid pages in each block. IB-FTL stores additional metadata in integrated RAM to enable traversing and cleaning its page validity log.

GeckoFTL and μ-FTL achieve the lowest integrated RAM footprints. The bottleneck for both is BVC. This bottleneck can be alleviated by increasing the size of blocks, or by only storing a sample of BVC in integrated RAM. Note that μ-FTL achieves a slightly lower footprint than GeckoFTL since the translation table is a B-tree and so only the root has to be stored in integrated RAM rather than GMD. Indeed, in situations where integrated RAM is too scarce for storing GMD, structuring the translation table as a B-tree is a viable design decision.

(2) Recovery Time Comparison. To compare the FTLs in terms of recovery time, we modeled the number and types of flash IOs that are needed to recover each of their RAM-resident data structures. Appendix C shows the modeling in detail for GeckoFTL. In our models, reading a flash page takes 100 μs, reading a spare area takes 3 μs, and writing a flash page takes 1 ms [15]. The middle part of Figure 13 shows the results of our cost models. Note that the bottom part of Figure 1 in the introduction is based on the middle part of Figure 13 and captures the total recovery time for LazyFTL under different device capacities.

LazyFTL and IB-FTL exhibit significant bottlenecks to recovery time. Their shared bottleneck, denoted as "LRU cache", accounts for the time taken to recover and synchronize dirty mapping entries with the translation table. DFTL and μ-FTL avoid this bottleneck by relying on battery. In contrast, GeckoFTL eliminates this bottleneck by only identifying dirty entries during recovery and deferring their synchronization until after normal operation resumes.

LazyFTL's second bottleneck arises due to recreating the RAM-

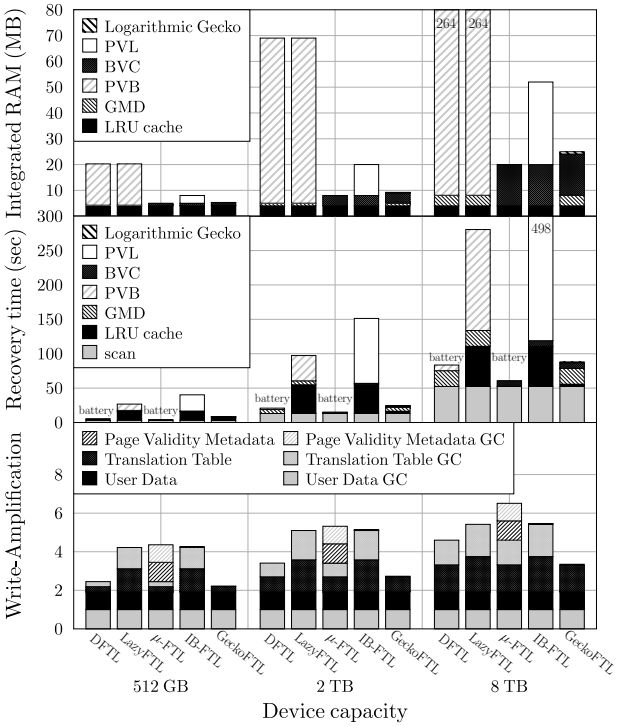


Figure 13: GeckoFTL achieves a better balance among integrated RAM, recovery time, and write-amplification than existing FTLs for large devices without relying on a battery.

resident PVB by scanning the translation table. DFTL avoids this bottleneck by using a battery to copy PVB to flash before power runs out. In contrast, μ-FTL and GeckoFTL avoid this overhead because they store page validity metadata in flash, and so it persists across failures.

IB-FTL's second bottleneck accounts for the time to recover metadata pertaining to its page validity log. Recovering this metadata entails scanning the whole log, whose size is proportional to device capacity.

Finally, Figure 13 shows that the time to initially scan the device to determine the types of blocks is emerging as a bottleneck for all FTLs. This bottleneck may be alleviated by using a larger block size or through parallelism, as a flash device typically consists of multiple logical units that can be accessed by the controller in parallel [3].

(3) Write-Amplification Comparison. To compare the FTLs in terms of write-amplification, we simulated each of them under uniformly randomly distributed writes and measured the number of IOs taking place for different purposes. The bottom part of Figure 13 shows write-amplification due to (1) application updates and garbage-collection of user data, (2) synchronization operations and garbage-collection of translation metadata, and (3) updates, GC queries and garbage-collection of page validity metadata.

The overhead of updating translation metadata is highest for LazyFTL and IB-FTL because they restrict the proportion of dirty mapping entries in the LRU cache. Thus, updates to the translation table are amortized to a lesser extent. For the other FTLs, the overhead of updating translation metadata is largely the same. Note that even though μ-FTL and IB-FTL structure the translation table as a B-tree, this does not increase the cost of updates to the translation table by much because the root and internal nodes of the B-tree are usually cached in integrated RAM.

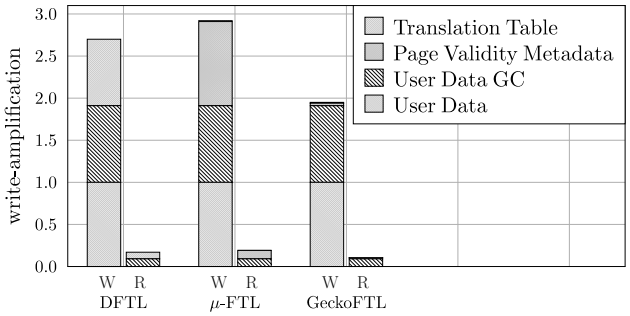


Figure 14: Even when integrated RAM is plentiful enough for storing PVB, GeckoFTL still improves performance by freeing integrated RAM to cache a larger proportion of the translation table.

Recall that GeckoFTL uses runtime checkpoints to bound recovery time by synchronizing dirty cached entries that have not been updated for a long time, as explained in Section 4.3. This leads to premature eviction and synchronization of dirty mapping entries and should result in an increase in the cost of updates to translation metadata in GeckoFTL. However, Figure 13 shows that the write-amplification induced by updating the translation table is largely the same in GeckoFTL as in DFTL and u-FTL, which rely on a battery. This demonstrates that the checkpoints in GeckoFTL increase write-amplification by a negligible amount.

In terms of page validity metadata, μ-FTL exhibits the highest write-amplification because it uses a flash-resident PVB. DFTL and LazyFTL avoid this overhead because they store PVB in integrated RAM. IB-FTL and GeckoFTL both achieve low write-amplification by virtue of buffering and later reorganizing page validity metadata in flash.

Finally, Figure 13 shows that garbage-collecting metadata blocks significantly increases write-amplification for DFTL, LazyFTL, μ-FTL and IB-FTL. GeckoFTL eliminates these overheads by not targeting blocks that contain metadata for garbage-collection, because such blocks contain frequently updated metadata that will soon be updated anyways.

Overall. Figure 13 shows that GeckoFTL achieves a better balance than existing FTLs among integrated RAM requirement, recovery time and write-amplification, and it does so without relying on a battery.

5.4 Better RAM utilization with GeckoFTL

Let us now suppose that integrated RAM is plentiful enough to store PVB, so storing page validity metadata in flash is not strictly necessary. We show that even in this case, using Logarithmic Gecko improves performance because the integrated RAM that it frees up is used to cache a larger proportion of the translation table.

In Figure 14, we compare three FTLs designs. All of these FTLs are given $\approx$ 70 MB of integrated RAM. The first is DFTL, which stores PVB in integrated RAM. It uses 64 MB of its integrated RAM for storing PVB, 4 MB for the LRU cache (as in the previous experiments), and $\approx$ 2 MB for the remaining RAM-resident data structures. The second FTL is μ-FTL, which stores PVB in flash. All of its remaining integrated RAM is allocated to the LRU cache, which has a size of 68 MB. The third is GeckoFTL. It also has a cache size of 68 MB, but it stores page validity metadata using Logarithmic Gecko rather than using a flash-resident PVB. To enable an apples to apples comparison, we give the implementations of DFTL and μ-FTL in this experiment the same garbage-collection scheme as GeckoFTL.

For DFTL, updating PVB involves no IO overheads since it is in integrated RAM, but the overhead due to updating translation meta-

data is significant. For μ-FTL, the overhead due to updating translation metadata drops to nearly 0 because the larger cache enables more amortization of synchronization operations, but the overhead due to updating PVB in flash is significant. In contrast, GeckoFTL achieves the best of both worlds. The cost of synchronization operations is nearly zero because of the larger cache, and the cost of updating page validity metadata in flash is low by virtue of using Logarithmic Gecko. Thus, GeckoFTL enables a more effective use of the available integrated RAM than existing FTLs even when integrated RAM is plentiful enough for storing PVB.

6. RELATED WORK

In this section, we describe how state-of-the-art page-associative FTLs maintain page validity metadata. We also discuss how they garbage-collect flash-resident metadata, as well as how they ensure quick recovery from power failure. We show how GeckoFTL advances the state-of-the-art in all of these respects.

Page Validity Metadata. DFTL and LazyFTL [22, 26] use one bit in the spare area of each flash page to indicate if the page is invalid. However, in modern flash devices the spare area of a flash page cannot be updated until the underlying block is erased [3, 24, 18]. Thus, an implementation of DFTL or LazyFTL would need to consolidate these bits into PVB and store it either in integrated RAM or in flash. On the other hand, μ-FTL [24] explicitly stores PVB in flash. The problem with these schemes is that storing PVB in integrated RAM is expensive and increases recovery time, whereas storing PVB in flash significantly increases write-amplification.

IB-FTL [18] logs the addresses of invalidated flash pages in flash while maintaining a linked list of pointers between log entries that correspond to invalid pages on the same blocks, and where the first link in each chain is stored in integrated RAM. Although it generates low write-amplification, the problem is that its RAM-resident metadata significantly increases the integrated RAM footprint and is slow to recover as the entire log must be scanned. Note that the log does not come with a cleaning mechanism, and so we extend it with one in Appendix E to enable an apples to apples comparison with other techniques in the evaluation.

In contrast, GeckoFTL uses Logarithmic Gecko, which generates low write-amplification while still enabling a great reduction in recovery time and integrated RAM.

Garbage-Collection. State-of-the-art page-associative FTLs [16, 26, 24] use a greedy garbage-collection policy that always chooses as a victim the block with the least number of valid pages in the device. In contrast, we propose a garbage-collection policy that never targets metadata. Instead, it waits for blocks containing metadata to become completely invalid and then erases them. This policy significantly reduces garbage-collection overheads.

Our policy in inspired by state-of-the-art garbage-collectors that separate logical pages into different groups of flash blocks based on update frequency [37, 8] and allocate relatively more over-provisioned space to hotter groups [35, 12]. However, state-of-the-art garbage-collectors rely on temperature detectors (e.g. [17, 34]) to determine the update frequency of a logical page. Temperature detectors exhibit a trade-off between accuracy and RAM-overheads, and in our environment integrated RAM is scarce. Moreover, temperature detectors fail when the workload changes or exhibits periodicity. In contrast, our approach leverages the type of data (i.e. user data vs. metadata) to infer its update frequency. This approach avoids the above-mentioned pitfalls of temperature detectors.

Recovery. State-of-the-art FTLs cache recently updated mapping entries in integrated RAM to amortize updates to the flash-resident translation table. When a failure occurs, the FTL need to recover these so-called dirty mapping entries.

The best known recovery algorithm for dirty mapping entries, proposed in LazyFTL, exhibits two shortcomings that GeckoFTL improves upon. In LazyFTL, whenever a dirty mapping entry is evicted from the cache, the cache is scanned and all other dirty entries that belong on the same physical block are also synchronized. This is done so that non-fully-synchronized blocks can be quickly identified during recovery. The problem is that scanning the cache for each eviction is computationally expensive. In contrast, GeckoFTL only scans the cache once per checkpoint thereby significantly reducing the computational overhead.

The second problem of LazyFTL's recovery approach is that it limits the number of dirty entries in the cache to bound the time it takes to synchronize all of them with the translation table. This decreases the amount by which updates to the translation table can be amortized, and so write-amplification during runtime increases. In contrast, GeckoFTL defers synchronizing the recreated mapping entries until after normal operation resumes. Thus, GeckoFTL does not need to bound the number of dirty entries in the cache. This removes the contention between recovery time and write-amplification.

Logarithmic Method. Various write-optimized key-value stores [32, 25, 4, 38] use the Logarithmic Method [5, 29], which involves logging updates and later reorganizing them through merge operations to guarantee logarithmic access time for queries. In this paper, we apply the logarithmic method outside of its traditional usage for key-value storage. We show how to use the logarithmic method to aggregate information about objects and report the aggregations through queries. Applying the logarithmic method for write-optimized aggregations in secondary storage is generalizable beyond Logarithmic Gecko.

7. CONCLUSION

In this work, we show that as flash devices scale in capacity to the order of terabytes, the metadata space requirement and the recovery time needed by state-of-the-art Flash Translation Layers are growing at an unsustainable rate. We identify the Page Validity Bitmap as the main bottleneck as it takes up 95% of all RAM-resident metadata and a large proportion of recovery time. We show that persisting PVB in flash is a poor solution because it increases write-amplification thereby harming performance and device longevity.

To solve this problem, we design a novel FTL called GeckoFTL. The central innovation of GeckoFTL is using an alternative data structure called Logarithmic Gecko, which logs updates about page validity in flash and later reorganizes them to ensure fast and scalable access time for queries. Logarithmic Gecko involves significantly cheaper updates yet costlier garbage-collection queries relative to the baseline. We show that this is a good trade-off since garbage-collection queries occur infrequently relative to updates to page validity metadata, and since flash reads are cheaper than flash writes. Logarithmic Gecko reduces write-amplification relative to the baseline by 98% while still enabling a 95% reduction in integrated RAM consumption and at least a 51% reduction in recovery time.

8. REFERENCES

[1] Small-block vs. large-block nand flash devices. *Technical Report, TN-29-07, Micron*, 2007.

[2] A. Aghayev, M. Shafaei, and P. Desnoyers. Skylight-a window on shingled disk operation. *ACM Transactions on Storage (TOS)*, 11(4):16, 2015.

[3] N. Agrawal et al. Design tradeoffs for SSD performance. In *USENIX*, pages 57–70, 2008.

[4] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 81–92. ACM, 2007.

[5] J. L. Bentley and J. B. Saxe. Decomposable searching problems i. static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[6] L. Bouganim, B. Jonsson, and P. Bonnet. *uFlip: Understanding Flash IO Patterns*, pages 1–12. CIDR. 2009.

[7] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 181–192. ACM, 2009.

[8] M.-L. Chiang, P. C. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for flash memory. *SOFTWARE-PRACTICE & EXPERIENCE*, 29(3):267–290, 1999.

[9] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. System software for flash memory: a survey. In *Embedded and Ubiquitous Computing*, pages 394–404. Springer, 2006.

[10] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. A survey of flash translation layer. *Journal of Systems Architecture*, 55(5):332–343, 2009.

[11] N. Dayan, M. K. Svendsen, M. Bjørling, P. Bonnet, and L. Bouganim. Eagletree: Exploring the design space of SSD-based algorithms. *Proc. VLDB Endow.*, pages 1290–1293, Aug. 2013.

[12] P. Desnoyers. Analytic modeling of SSD write performance. In *SYSTOR*, pages 12:1–12:10, 2012.

[13] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[14] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, June 2005.

[15] L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[16] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS*, pages 229–240, 2009.

[17] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang. Efficient identification of hot data for flash memory storage systems. *ACM Transactions on Storage (TOS)*, 2(1):22–40, 2006.

[18] P.-C. Huang, Y.-H. Chang, and T.-W. Kuo. An index-based management scheme with adaptive caching for huge-scale low-cost embedded flash storages. *ACM Trans. Des. Autom. Electron. Syst.*, 18(4):56:1–56:26, Oct. 2013.

[19] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *USENIX*, pages 13–13, 1995.

[20] H.-J. Kim and S.-G. Lee. An effective flash memory manager for reliable flash memory space management. *IEICE Transactions on Information and Systems*, 85(6):950–964, 2002.

[21] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Trans. on Consum. Electron.*, 48(2):366–375, May 2002.

[22] Y. Kim, A. Gupta, and B. Urgaonkar. A temporal locality-aware page-mapped flash translation layer. *Journal of Computer Science and Technology*, 28(6):1025–1044, 2013.

[23] S.-W. Lee, W.-K. Choi, and D.-J. Park. FAST: An efficient flash translation layer for flash memory. In *EUC Workshops*, pages 879–887, 2006.

[24] Y.-G. Lee, D. Jung, D. Kang, and J.-S. Kim. μ-ftl:: a memory-efficient flash translation layer supporting multiple mapping granularities. In *EMSOFT*, pages 21–30. ACM, 2008.

[25] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *Proceedings of the VLDB Endowment*, 3(1-2):1195–1206, 2010.

[26] D. Ma, J. Feng, and G. Li. Lazyftl: a page-level flash translation layer optimized for nand flash memory. In *SIGMOD*, pages 1–12. ACM, 2011.

[27] D. Ma, J. Feng, and G. Li. A survey of address translation technologies for flash memories. *ACM Comput. Surv.*, 46(3):36:1–36:39, Jan. 2014.

[28] J. M. Marshall and C. D. H. Manning. Flash file management system, Nov. 3 1998. US Patent 5,832,493.

[29] U. Meyer, P. Sanders, and J. Sibeyn, editors. *Algorithms for Memory Hierarchies: Advanced Lectures*. Springer-Verlag, Berlin, Heidelberg, 2003. pages 33-34.

[30] I. Micron Technology. Nand flash memory - mlc+. 2013.

[31] I. Micron Technology. Wear-leveling techniques in nand flash devices. 2013.

[32] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[33] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications. *ACM Trans. Embed. Comput. Syst.*, 7(4):38:1–38:23, Aug. 2008.

[34] D. Park and D. H. Du. Hot data identification for flash-based storage systems using multiple bloom filters. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–11. IEEE, 2011.

[35] R. Stoica and A. Ailamaki. Improving flash write performance by using update frequency. *Proc. VLDB Endow.*, pages 733–744, July 2013.

[36] S. Wells. Method for wear leveling in a flash eeprom memory, Aug. 23 1994. US Patent 5,341,339.

[37] M. Wu and W. Zwaenepoel. envy: a non-volatile, main memory storage system. In *ACM SigPlan Notices*, volume 29, pages 86–97. ACM, 1994.

[38] X. Wu, Y. Xu, Z. Shao, and S. Jiang. Lsm-trie: An lsm-treebased ultra-large key-value store for small data. In *USENIX Annual Technical Conference*, 2015.

APPENDIX

A. MULTI-WAY MERGING

As mentioned in Section 3, a merge operation in Logarithmic Gecko may continue recursively. This is wasteful in terms of flash writes, as it involves rewriting Gecko entries from lower levels mul-

tiple times. We can reduce this overhead by foreseeing a recursive merge, and merging all the runs from the onset using a multi-way sort merge. This reduces the IO overhead of merge operations by a factor of $\approx 1/T$. To achieve this, we use the following policy. The run at level i participates in a commencing merge if (1) it is not already participating in another merge, and (2) there is at least one run at level $i-1$ participating in this merge. The downside of this policy is that it increases the number of input buffers needed in integrated RAM from 2 to L, the number of levels in the tree. Thus, this technique is only applicable for devices for which integrated RAM is larger than $P \cdot L$.

B. RAM BREAKDOWN

We now derive formulas for the amount of integrated RAM needed by the different data structures in GeckoFTL that are not already given in Section 2. The terms are given in Figure 2, and we denote I_X as an X byte integer.

Logarithmic Gecko's Run Directories

- *Number of entries:* The run directories contain one entry for every Gecko page in the device. Thus, we first derive the number of Gecko pages. Every Gecko entry takes up I_4 bytes for the key and $B/8$ for the page validity bitmap. Thus, the number of Gecko entries that fit into a flash page is $P/(I_4 + B/8)$. The largest run contains K entries, one for each block, and so its size in flash pages is $K/(P/(I_4 + B/8))$. The cumulative size of the rest of the runs is at most the size of the largest run. Thus, the total number of flash pages in Logarithmic Gecko is $(2 \cdot K)/(P/(I_4 + B/8))$.
- *Entry size:* Each entry in the run directories is a mapping from a block ID to the physical address of a flash page. This takes up $2 \cdot I_4$ bytes.
- *Total:* $2 \cdot I_4 \cdot (2 \cdot K)/(P/(I_4 + B/8))$ bytes.

Blocks Liveness Counter (BVC)

- *Number of entries:* There are K entries in BVC, one for each block.
- *Entry size:* I_2 bytes are needed to store the number of invalid pages in each block.
- *Total:* $I_2 \cdot K$ bytes.

Logarithmic Gecko's Buffers

- *Number of entries:* Logarithmic Gecko has one input buffer to which Gecko entries are inserted when physical pages become invalid. In addition, the multi-way merge strategy in Appendix A requires at most L input buffers, one for each level, and one output buffer.
- *Entry size:* The size of each buffer is P.
- *Total:* $P \cdot (2 + L)$ bytes.

C. RECOVERY

The recovery algorithm of GeckoFTL, which we call GeckoRec, comprises eight steps. In this section, we explain each of them and quantify their costs in terms of flash IOs.

Step 1. GeckoRec commences by creating a temporary data structure in integrated RAM called the Blocks Information Directory (BID). An entry in BID consists of 2 fields: the type of the block (Gecko, translation, user, or free), and a timestamp of when the first page in the block was written. To create BID, GeckoRec reads the spare area of the first flash page of each flash block, where the timestamp and the block's type are stored. Creating BID takes K spare area reads.

GeckoRec creates entries in BID for all translation, Gecko and free blocks. Since most flash blocks contain user data, creating a BID entry for every user block may cause BID to exceed the integrated RAM capacity. Thus, GeckoRec only creates BID entries for the most recently written $\max(\lceil \frac{2 \cdot C}{B} \rceil, V)$ user blocks. The reason this number must be at least $\lceil \frac{2 \cdot C}{B} \rceil$ is to enable scanning the last $2 \cdot C$ user pages to recover dirty mapping entries, as explained in Section 4.3. The reason it must also be at least as large as V, the size of Logarithmic Gecko's buffer, is to enable finding recently erased blocks that were lost from Logarithmic Gecko's buffer when power failed, as explained in Appendix C.2.

Step 2. Next, GeckoRec scans the spare areas of all translation pages, where the ID and last-update-timestamp for each of them are stored. It uses this to recover GMD by finding the most recent version of every translation page. This takes $O(\frac{K \cdot B}{P})$ spare area reads because the number of translation pages is $O(\frac{K \cdot B}{P})$.

Step 3. GeckoRec then recovers Logarithmic Gecko's run directories by scanning the spare areas of all flash pages in Gecko blocks. This takes $O(\frac{K \cdot B}{P})$ spare area reads because the number of Gecko pages is $O(\frac{K \cdot B}{P})$. Further technical details about recovering the run directories are given in Appendix C.1.

Step 4. Next, GeckoRec recovers Logarithmic Gecko's buffer, as explained in detail in Appendix C.2. This takes at most $2 \cdot V$ page reads.

Step 5. GeckoRec next recreates the Blocks Liveness Counter (BLC), which keeps track of the number of valid pages in each block in integrated RAM. It does this by scanning Logarithmic Gecko, reconstructing a page validity bitmap for every flash block, and computing its hamming weight to give the number of invalid pages in that block. This takes $O(\frac{K \cdot B}{P})$ flash reads, since Logarithmic Gecko consists of $O(\frac{K \cdot B}{P})$ flash pages.

Step 6. Next, GeckoRec recovers dirty mapping entries, as explained in Section 4.3. This step takes at most $2 \cdot C$ spare area reads.

Step 7. For all recreated mapping entries, GeckoRec sets the dirty and UIP flags to true. Mistakes will be fixed after normal operation resumes, as explained in Appendix C.3. The IO cost introduced by fixing mistakes is at most $O(\frac{K \cdot B}{P})$ flash reads as well as $O(C)$ spare area reads and redundant insertions into Logarithmic Gecko, but these overheads take place after recovery is complete and so they do not elongate recovery time.

Step 8. GeckoRec disposes of BID, and normal operation resumes.

C.1 Restoring the Run Directories

When power fails, we lose Logarithmic Gecko's run directories. We now show how to recover them. The challenge is that there may be multiple obsolete runs on Gecko blocks, but we must only recover the directories for the runs that were valid at the moment of power failure. Our approach relies on adding some metadata to each run to enable determining which run is the most recently created in each level.

We pad each run in flash with a preamble and a postamble. The preamble stores the run's level, a creation timestamp, and a unique ID. The postamble stores a copy of the run directory for this run. Every other page in the run begins with a header that contains the ID of the run that it belongs to.

During recovery, we examine the preamble and postamble of every run. We discard any run without a postamble as it is only partially written. We then use the creation timestamps to identify the most recently-created run in each level. For these runs, we recover the run directories from the postambles to integrated RAM.

C.2 Restoring the Buffer's Contents

When power fails, we lose the content of Logarithmic Gecko's buffer, which contains the addresses of recently erased blocks and recently invalidated flash pages. In this section, we show how to recover them.

C.2.1 Restoring Addresses of Erased Blocks

First, GeckoRec identifies and recreates Gecko entries for all blocks that were erased since the last time that Logarithmic Gecko's buffer flushed. To do this, it identifies the smallest run in Logarithmic Gecko and reads its preamble to find its creation timestamp. It then searches BID for all blocks that are free or whose first page was written after this timestamp. For each of these blocks, it invokes Algorithm 2, which inserts a Gecko entry into the buffer with the erase flag set to true. Since at most V blocks can be erased between two times that Logarithmic Gecko's buffer flushes, BID must contain entries for at least the most recently erased V user blocks to guarantee that GeckoRec does not miss any recently erased blocks.

C.2.2 Restoring Addresses of Invalidated Pages

Next, GeckoRec identifies and recreates Gecko entries for all flash pages that were invalidated since the last time that Logarithmic Gecko's buffer flushed. To do so, it exploits the property that an invalid flash page is only reported to Logarithmic Gecko during a synchronization operation, which updates a translation page. Thus, GeckoRec finds all translation pages that were updated since the last time that Logarithmic Gecko's buffer flushed. It compares each of these translation pages to the most recent previous version of the same translation page[8]. Every mapping entry in the previous translation page that mismatches the entry in the current translation page corresponds to a flash page that was invalidated since the last buffer flush. For each such mapping entry $e = (X, Z)$, we check if the physical address Z is still invalid by reading the spare area of flash page Z and checking if the logical address last written on it is still X. If so, GeckoRec inserts Z into Logarithmic Gecko's buffer via Algorithm 1.

For this approach to work, we must ensure that recently invalidated translation pages are not erased. To do so, we maintain a list of blocks in integrated RAM that cannot be erased. When a translation page is updated, we insert the ID of the block that contains the invalidated version of the translation page into the list. When Logarithmic Gecko's buffer is flushed, we clear the list.

The dominant cost of this phase is reading and comparing translation pages. Since the buffer contains up to V Gecko entries, then V flash pages are typically invalidated in-between two times the buffer flushes. In this case, as many as V translation pages could have been synchronized, so at most $2 \cdot V$ translation pages must be read and compared during this phase of recovery. Note that technically, if multiple logical pages that all correspond to different translation pages are stored on the same flash blocks and are updated at the same time, then the buffer can absorb as many as $V \cdot B$ inserts before flushing, in which case $2 \cdot V \cdot B$ translation pages must be read and compared during this phase of recovery. Although this scenario is unlikely, it is possible to restrict recovery time to $2 \cdot V$ by limiting the number of insertions that the buffer can absorb to V before it flushes.

[8] As long as $C > V$, the same translation page cannot be updated more than once in-between two times that Logarithmic Gecko's buffer flushes, so there is always at most one previous version for each updated translation page that we must examine.

C.3 Correcting Restored Cached Entries

GeckoRec defers synchronizing the recreated cached mapping entries with the flash-resident translation table until after normal operation resumes in order to shorten recovery time. Instead, it simply sets the dirty and UIP flags for each restored cached mapping entry to true. We now show how GeckoFTL corrects these flags after recovery during regular synchronization operations.

C.3.1 Dirty = false, UIP = false

We first consider a restored cached mapping entry for which the corresponding mapping entry before power failed had its dirty and UIP flags both set to false. When the restored mapping entry participates in a synchronization operation, GeckoFTL compares its physical address to the flash-resident mapping entry's physical address. If these addresses match, it means that the cached mapping entry was wrongly marked as dirty, and so GeckoFTL sets its dirty and UIP flags to false and omits it from the synchronization operation. If all cached entries participating in a synchronization operation are omitted, GeckoFTL aborts the synchronization operation thereby saving one flash write. The overhead introduced is $O(\frac{K \cdot B}{P})$ page reads because there are $O(\frac{K \cdot B}{P})$ translation pages and so the number of aborted synchronization operations is $O(\frac{K \cdot B}{P})$. However, this IO price is only paid after regular operation resumes, and so it does not lengthen recovery time.

C.3.2 Dirty = true, UIP = false

We now consider a restored cached entry for which the corresponding mapping entry before power failed had its dirty and UIP flags set to true and false respectively. In this case, only the UIP flag must be corrected after recovery. As before, when the cached entry participates in a regular synchronization operation after recovery, GeckoFTL compares the physical addresses of the cached entry and the flash-resident entry. Their addresses do not match (this is true by assumption that the cached entry before power failure was dirty), and so GeckoFTL replaces the flash-resident entry by the cached entry. After doing so, GeckoFTL would normally also report the physical address Z of the replaced entry as invalid to Logarithmic Gecko. However, address Z was already reported to Logarithmic Gecko before power failed (this is true by assumption that the cached entry's UIP flag was false before power failed). There is a danger in reporting page Z to Logarithmic Gecko a second time as invalid; page Z may have already been erased and rewritten before power failure, and so reporting it a second time as invalid can lead to losing live data currently written on it. To avoid this problem, GeckoFTL first checks in the spare area of page Z if the logical page written on it is the same as in the restored cached entry. It only reports Z as invalid to Logarithmic Gecko if these logical addresses match. Using this approach, GeckoFTL may still report a physical address to Logarithmic Gecko as invalid a second time after recovery, but the above check at least guarantees that GeckoFTL does not report a valid page as invalid.

Cost. This check introduces an overhead of one spare area read for every entry participating in a synchronization operation. This is not a large overhead since spare area reads are extremely cheap compared to flash writes. However, this overhead can also easily be restricted. The simplest method is to add an uncertainty flag to each cached mapping entry to signal that we are not sure if the UIP flag should be true or not. The uncertainty flag is set to true for any cached entry created during recovery. Otherwise, it is always set to false. Thus, we only perform the spare area check during a synchronization operation if the cached entry's uncertainty flag is set to true. After the check, we set the uncertainty flag of the entry

to false. Since at most C entries can have their uncertainty flag set to true, then the overhead is $O(C)$ spare area reads and redundant insertions to Logarithmic Gecko.

We can also achieve the same effect without introducing an uncertainty flag. The dirty and UIP flags during normal operation can only ever be (false, false), (true, false) and (true, true). Thus, we can set the dirty and UIP flags for all restored cached mapping entries to (false, true), and use this combination to mean that the dirty flag and UIP flag are currently assumed to be true but that we are uncertain of this, so we should perform all the appropriate checks during a synchronization operation after power resumes.

D. WEAR-LEVELING

In this Appendix, we show how GeckoFTL performs wear-leveling. We show that the only necessary metadata in integrated RAM includes a few global statistics comprising $30-40$ bytes at most. The rest of the wear-leveling metadata can be stored in the spare areas of blocks.

Wear-Leveling Statistics. GeckoFTL relies on two statistics for each block to perform wear-leveling. The first statistic is erase-count, which is the number of times a block has been erased (used to identify blocks that are exceptionally unworn [36, 20, 3]). The second statistic is age, which is the number of erases that has taken place in the device as a whole since the last time that a given block was erased (used to identify blocks that contain static data [19]).

GeckoFTL maintains the erase-count for each block in one of its spare areas. An erase-count comprises 2 bytes per block because every block has a lifetime of at most a few thousands of erases [3]. The solution in [28] is used to maintain the erase-count in the presence of power failures.

To maintain the age of blocks, GeckoFTL uses a global erase counter (4 bytes) in integrated RAM. When any block is erased, the value of the global counter at that moment is saved as an erase-timestamp in one of the spare areas of the block, and the global counter is incremented. The age of any block can thus be calculated by subtracting its erase-timestamp from the global counter. The erase-timestamp comprises 4 bytes for each block.

Finding Wear-Leveling Victims. GeckoFTL only stores a few global statistics for the erase-counts and ages of blocks such as minimum, maximum and average (24 bytes). It periodically scans the device, issuing one spare area read per block, to update its global statistics and to identify wear-leveling targets based on how their erase-count and age compare to the global statistics. The scan takes place gradually with respect to flash writes. In particular, for every flash write that takes place, we read the spare area of the next block in the scan. When a scan is finished, we restart it. Since spare area reads are 3 orders of magnitude less expensive than flash writes, this ensures that scans do not introduce a significant performance overhead.

Scan Cost Analysis. Let us define a period as $K \cdot B$ flash writes, where K is the number of blocks in the device and B is the number of pages per block. By definition, B full scans take place during one period. Furthermore, let us assume that $1/X$ flash blocks contain non-static data. Thus, the blocks that contain non-static data each get erased X times on average during one period.

Clearly, as long as $X < B$ the policy does not fall behind. Under this condition, static blocks are inspected by the scan more frequently than the rate at which non-static blocks get erased. For example, suppose that X is 2, meaning that 50% of the blocks are non-static, and suppose that B is 128. During one period, the non-static blocks are each erased 2 times on average whereas the static block are inspected 128 times. It is therefore extremely easy to discover and address erase-count discrepancies as they arise.

This policy works well even when $X > B$. For example, suppose X is set to 1000, meaning that 0.1% of the blocks are non-static. In this case, the rate at which non-static blocks are erased is $1000/128 \approx 8$ times higher than the rate at which static blocks are scanned. Thus, each static block may be erased ≈ 8 times on average while we complete an entire scan to identify wear-leveling victims, and so the erase-count discrepancy will be no less than ≈ 8. This is acceptable seeing as the number of erases each block can undergo is in the hundreds or even thousands for modern flash devices [15]. In a stress case where $X >> B$, it is possible to increase the frequency of scans (i.e. read more than 1 spare area for each flash write) to ensure that the wear-leveling policy does not fall behind.

In summary, scans obviate the need for storing wear-leveling statistics for each block in integrated RAM in order to find wear-leveling victims. The frequency of scans is easy to tune such that they neither impact performance nor fall behind.

Dynamic Wear-Leveling. GeckoFTL uses a complementary dynamic wear-leveling technique to write frequently updated pages on unworn blocks (to accelerate their wearing rate) and seldom updated pages on worn blocks (to decelerate their wearing rate) [31]. To do this, GeckoFTL needs to know the update frequency of pages. It does this by keeping an update timestamp for each page in its spare area. When a page is updated or migrated, we can then tell based on the timestamp how long it has been since the last time it was updated relative to the average.

Overall, GeckoFTL uses a range of wear-leveling techniques, which only require storing a few bytes' worth of global statistics in integrated RAM.

E. PAGE VALIDITY LOG

IB-FTL [18] proposes to log the addresses of invalidated flash pages while maintaining a linked list of pointers between log entries that correspond to invalid pages on the same blocks, and where the first link in each chain is stored in integrated RAM. However, the original design in [18] does not include a cleaning mechanism, and so the page validity log (PVL) can grow indefinitely. To enable a fair comparison of Logarithmic Gecko and PVL in our experiments, we extended the design of PVL with a cleaning mechanism.

Our extension involves adding a timestamp to every log entry corresponding to when the page was invalidated. We also add a timestamp for every block in integrated RAM of the last time it was erased. We then bound the size of the log to X log entries. When the buffer is flushed and the log grows beyond X entries, we find the page that was inserted into the log the longest time ago. For every entry in this page, we compare its timestamp to the erase-timestamp of the corresponding block in integrated RAM. If the entry's timestamp is larger than the block's timestamp, it means the log entry was created after the last time the block was erased, and so we reinsert it into the log. On the other hand, if the log entry's timestamp is smaller, we discard it, because it was created before the last time the block was erased and is now obsolete.

A natural question is how to tune X, the size of the log. Our insight is that at any point, the number of invalid pages in the device is at most the difference between the sizes of the physical address space and the logical address space. We denote this difference as D. We propose to set X to double the size of D. Thus, when we reclaim the oldest page in the log, at least half of its log entries are discarded on average. Each log entry is reinserted into the log on average one time, and so the cost in terms of write-amplification is $O(\frac{1}{V})$, where V is the size of the buffer. In Section 5, we compare our version of PVL to Logarithmic Gecko.