Design Tradeoffs in a Flash Translation Layer

Garth Goodson
NetApp
495 East Java Drive
Sunnyvale, CA, USA
garth.goodson@netapp.com

Rahul Iyer
NetApp
495 East Java Drive
Sunnyvale, CA, USA
rahul.iyer@netapp.com

ABSTRACT

Although flash has better random access characteristics than disk, it has a number of undesirable complexities, including the inability to overwrite a block in place. A layer, called the *flash translation layer* is used to vitalize flash and presents it as a block device. Although many such designs have been proposed in the past, they often do not focus on the requirements of large-scale enterprise storage, nor do they examine the trade-offs between different approaches. In this paper, we use trace-based simulation to analyze the trade-offs and designs of two popular classes of flash translation layers.

1. INTRODUCTION

Driven by the performance requirements of low latency and delivering orders of magnitude more random I/Os per second (IOPs), flash is beginning to make serious inroads within commercial storage systems. The ever-dropping price per gigabyte combined with the increasing densities and low cost per IOPs have propelled flash from hand-held devices to the enterprise market. Although flash offers a huge performance improvement, making it as easy and efficient to use as today's disk drives presents a challenge.

Although it is possible for storage systems to directly access raw flash, the storage system must be aware of the complexities involved in doing so, in particular, the inability to update data in place. One approach to integrating flash is through the use of flash-based file systems [16, 15, 22]. Although this approach ultimately provides the tightest integration and best performance, it can require an extensive rework of the storage system's existing infrastructure and file system. The other, more popular, technique is to package flash as a solid state disk drive (SSD) that exports a standard block interface (for example, SATA, SAS, FC). As flash memory does not support overwriting in place, flash devices require a translation layer to map logical blocks to their locations within physical flash memory. This layer is called the flash translation layer (FTL) [7, 14]. Using an FTL allows file systems and SSDs to maintain the block interface of disks without sacrificing the tighter integration and control over how the flash is managed. This paper explores the trade-offs in designing a flash translation layer and the impact of proposed changes to the existing block interfaces.

In this paper we describe the design and implementation of two FTLs and analyze their overheads. The two FTLs are chosen based on the two most common classes of translation layers discussed in the literature. The first FTL uses a log-based approach where new updates are appended to the end of a log; the second is based on mapping consecutive ranges of logical blocks to consecutive sets of flash blocks. We implement both FTLs in a trace-based simulator and evaluate the overheads of overwriting, and the effectiveness of cleaning and wear leveling using synthetic traces and real block traces. We also analyze the impact of proposed changes to the ATA and SCSI protocols [11, 18] that allow the flash device to be notified of file-level block de-allocations.

The paper is organized as follows: Section 2 gives an overview of flash translation layers and describes the two main classes of FTLs. Section 3 describes the design implementation of the two FTLs we implement and evaluate. Section 4 evaluates the two FTLs. Section 5 compares related work. Finally, Section 6 concludes.

2. FLASH TRANSLATION LAYERS

This section describes the purpose of flash translation layers and provides an overview of the different generic classes of FTLs.

2.1 Overview

A flash translation layer hides the complexity of flash by providing a logical block interface to the flash device. Since flash does not support overwriting flash pages in place, an FTL maps logical blocks to physical flash pages and erase blocks.

2.1.1 Mapping pages to logical blocks

There are many ways to perform the mapping from flash pages to logical blocks. It can be algorithmic or it can store mapping information persistently on flash. Because algorithmic mappings are static in nature, they are seldom solely used for high-performance flash devices. Persistent mapping information can be spread across the metadata regions of each page. However, metadata stored in this way requires a scan of the entire device to reconstruct the full mapping table, which is too expensive for large devices and used only by much smaller

HPCA WEST 2010 Bangalore, India

consumer devices. An index or multilevel table can be used for faster metadata retrieval.

The FTL design also dictates the granularity of the mapping tables. For the most flexibility a *direct map*, where an entry exists for each logical block, can be used. However, it requires on the order of 4 GB of map space per 1TB of flash (assuming a 4KB page and an 8 byte entry).

2.1.2 Managing wear

In addition to maintaining mapping tables, an FTL must also manage the wear across the set of erase blocks. Wear leveling is either performed dynamically or statically. *Dynamic wear leveling* occurs when wear leveling is performed during the course of normal operation. *Static wear leveling* occurs out-of-band to move cold, low-wear blocks back into circulation. In the course of managing wear, it might be necessary to move the contents of cold erase blocks to more-worn erase blocks. Allowing the cold blocks to store hotter data allows them to become hotter, thus evening out the device's wear. The inability to overwrite in place, together with the granularity of erase, often necessitates the *copy out* of live data pages from an erase block before it can be reused.

2.2 FTL inefficiencies

Virtualizing flash as a new layer below the file system introduces inefficiency because the FTL block interface does not allow for the communication of file system block de-allocations. Delaying block de-allocations results in copying out more data during cleaning and wear leveling than is necessary. Modifying existing block protocols, shows the most promise for reducing inefficiencies. Changes are being introduced that allow for the communication of file-system-level block de-allocations down to the storage subsystem (for ATA [18], for SCSI [11]).

2.3 Common FTL Classes

There are two common classes of FTLs that are used in enterprise scale devices. The first FTL class uses a log-based layout where each new write request gets written to a new flash page, regardless of the logical block being written. The second FTL class assigns a fixed set of erase blocks to each logical block range.

2.3.1 Log-based FTLs

Similar to log-based file systems [5, 17], log-based FTLs treat the flash device as a log. Each new write is appended to the end of the log. Because each request is written to a new location, log-based FTLs match the characteristics of flash. However, because logical blocks can be written anywhere, the mapping layer must maintain an entry per logical block leading to a large metadata overhead.

2.3.2 Range-mapped FTLs

Range-mapped FTLs divide the logical block space into a set of fixed-sized ranges, each mapping a contiguous set of logical block numbers to a set of contiguous erase blocks called a *range container*. Because an entire logical block range can be mapped by a single entry, the amount of metadata required to locate a given logical block is significantly reduced. These mappings are not static, but change as the range containers fill and new ones are allocated.

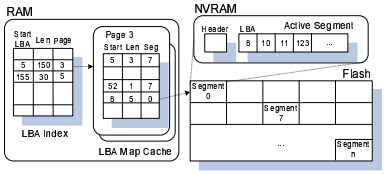

Figure 1: Log-based metadata overview.

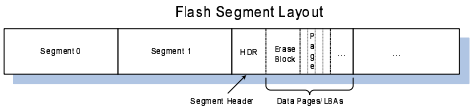

Figure 2: Log segment header.

3. DESIGN AND IMPLEMENTATION

This section outlines the design and implementation of two flash translation layers.

3.1 Design Assumptions

We make a number of assumptions in the design of our flash translation layers.

- The minimum logical block size is one page.
- A small amount of NVRAM is available to speed up write operations while ensuring consistency in the face of failures.
- The FTL must be able to map at least 1TB worth of logical block space.
- Metadata for mapping LBNs must pageable.

Given these assumptions, a 1TB flash device holds 2^{28} (256M) 4KB flash pages and 2^{22} (4M) 256KB erase blocks. A direct map with 8 byte entries uses 4 GB of RAM.

3.2 Log-based FTL

This section describes the design of a novel log-based FTL. Important distinctions of our design include: the incorporation of a small amount of NVRAM to speed updates and maintain consistency in the face of failure, the organization of the metadata mapping structures, and the algorithm used to combine wear leveling and cleaning.

The flash storage is logically broken into fixed-sized segments. There is only one segment, known as the *active segment* being written to at any one time. Logical blocks are written in log order as complete flash page writes. Each segment header contains a page map that maps the logical block number (LBN) of each logical block to its location within the segment. To locate a logical block the *LBA Map* maps logical block extents to segments. Because the LBA Map can be too big to fit in memory, there is a smaller, fully resident *LBA Index* that maps logical block extents to the appropriate section of the LBA Map. An overview of these structures is shown in Figure 1.

3.2.1 Log segments

As shown in Figure 2, each segment contains a segment header and a set of data pages. Segments are striped across multiple flash devices.

Pages containing logical blocks are written sequentially to the end of the active segment. Active segment header updates are collected in NVRAM until the segment is full. The seg-

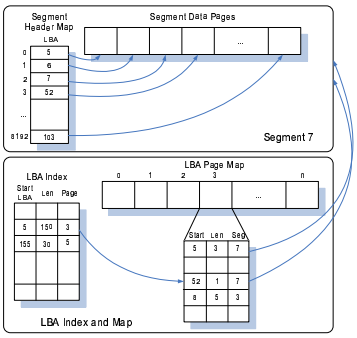

Figure 3: Log metadata indexes.

ment header is then written and a new active segment is selected from the free segment-list based on wear statistics.

If a logical block that exists in an older segment is updated, then the older segment header is not invalidated. Instead, another data structure, the LBA Map, tracks the segment that contains the latest version of a logical block.

3.2.2 LBA Map

A second data structure, the LBA Map is required to find the segment in which the logical block resides. We make use of block-extents to reduce the amount of metadata required to map logical blocks. For cases where the LBA Map does not fit within main memory, a third structure, called the *LBA Index*, is used to index into the LBA Map. The process of looking up a logical block is shown in Figure 3.

The LBA Map is used to find the segment storing a given logical block. The segment header's page map is then consulted to determine the exact location of the page within the segment. In case of inconsistencies, the LBA Map can be reconstructed by a scan over all segment header page maps. If a logical block is present in multiple segments, a timestamp within the segment header determines which was written latest.

The worst case size of the LBA Map is an extent length of one for each entry. This causes the extent map to degenerate into a direct map. In practice, the LBA Map is expected to be significantly smaller, because logical block extent-to-segment mappings, rather than individual block mappings are stored for each individual logical block and we expect most writes to contain runs of sequential logical blocks.

Updates to the LBA Map are logged to NVRAM and are periodically checkpointed to flash. On flash, LBA Map entries are ordered by extent range and stored in fixed-sized buckets; each bucket is a multiple of a flash page. When buckets fill, they are split, and half of the entries are moved to a new bucket.

3.2.3 LBA Index

Because the LBA Map might not fully fit within RAM, an index is built over the LBA Map to improve the search time for a given logical block. The LBA Index maintains maps LBA extents to the flash pages in which the LBA Map for that LBA extent range is stored. The LBA Index fits fully in RAM. The LBA Index need not be stored persistent on flash—it can be wholly reconstructed by scanning the LBA Map. Changes are logged to NVRAM and checkpointed to flash.

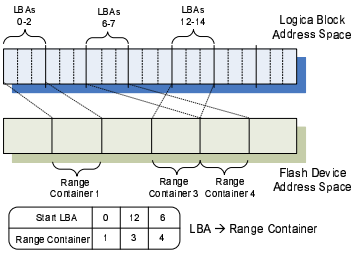

Figure 4: Range-mapped FTL example.

3.2.4 Cleaning data structures

In order for cleaning to be effective, metadata is required to determine 1) which segment to clean, and 2) which pages within the segment are live. On an overwrite, the old segment header is not updated to reflect that it no longer holds the most-recent copy of the block—all that is required is that the LBA Map point to the newest version of the block.

By examining the segment header's page map, the cleaner cannot determine which pages are valid within the segment being cleaned. The simplest approach to solve this problem is to maintain a per-segment bitmap corresponding to the state of each flash page (live/dead). For a 1TB flash device, a 2^{28} bit bitmap consumes 32MB of memory space. However, if an in-memory bitmap is used, it must be made persistent across power cycles or the bitmap must be reconstructed on startup by scanning the LBA map. The latter can be avoided by logging to NVRAM and checkpointing it periodically.

3.2.5 Cleaning and wear leveling algorithm

To prevent uneven wear within a segment, all live pages of a segment are copied out into a new segment when cleaned. The cleaning algorithm relies on a weighting function based on wear. This parameter, combined with the live page utilization of the segment, is used by the cleaner to select the next segment for cleaning. By varying these weighting factors, one can control the degree to which wear plays in the cleaner's selection of a segment.

The cleaning algorithm is a function of two ratios:

- Ratio of live blocks within a segment: $\%valid\_blocks = \frac{segment\_valid\_blocks}{segment\_size}$
- Ratio of the segment's wear to the maximum wear: $\%wear = \frac{segment\_wear\_count}{max\_wear\_count}$

We use two variables, α, β, as weights for each of these components, such that: $0 \leq \alpha, \beta < 1$. The general form of the segment selection function is: $f(\alpha, \%valid) * g(\beta, \%wear)$. Segment selection is based on minimizing this function. After substituting two simple linear functions for $f()$ and $g()$ weight: $[1-\alpha(\%valid)]*[1-\beta(\%wear)]$. To simplify things further, we constrain α and β such that $\beta = 1 - \alpha$. When the cleaner runs, this function is calculated for each dirty segment.

The cleaning function serves to make heavily worn segments with high churn less desirable to be selected as the active segment. When $\alpha = 0$, cleaning is solely based on segment wear and when $\beta = 0$, cleaning is solely based on segment utilization.

3.3 Range-based FTL

The range layout divides the logical block address-space into a set of contiguous fixed-sized ranges. Each range is

mapped to a *range container*. A range container is a set of contiguously allocated erase blocks, as shown in Figure 4.

3.3.1 Mapping metadata

In our range-layout, range containers have a minimum size of a single erase block and may grow to encompass several erase blocks. Even using only a single erase block per range container results in a huge metadata overhead savings. For example, mapping 1TB of flash with a logical block size of 4KB and an erase block size of 256KB requires only 4 M entries.

3.3.2 Wear leveling

Both static and dynamic wear leveling are required for this FTL. We maintain a *free-list* that holds all clean containers, and an *active-list* that holds all in-use (or *active*) containers. The set of active containers can be further broken into a set of cold containers that have not been written to in some time, and a set of hot containers that are actively updated.

Our static-wear leveling algorithm periodically swaps the least-worn cold-active range containers with the most-worn free range containers. This requires the sorting of both the free-list and the active-list. Periodically, a static wear leveling process examines the difference between the least-worn active range container and the most worn free range container. If their percent difference is beyond some threshold, then the contents of the cold-active range container are copied into the most-worn free range container and the old cold range container is erased and added to the free-list.

3.3.3 Optimizations

Optimizations are required to minimize the number of copy outs for certain classes of workloads, by delaying the need for copy out to the maximum extent possible.

Write-buffering.

In the evaluation (Section 4) we explore the effects of a small NVRAM write buffer.It operates by absorbing overwrites that occur close in time. Once the write buffer is full, I/Os to the same logical block range are collected and flushed to the appropriate range container. If the remaining free-space within the range container cannot hold the incoming I/O, the live pages from the container are written along with the write buffer data into a newly allocated range container.

Overprovisioning.

Copy outs can also be reduced by increasing the amount of flash storage dedicated to a range container, without increasing the number of logical blocks it maps. We explore two basic methods of doing this. The first method equally divides the overprovisioned free-space across all range containers. The second method allocates additional free range containers, to hot logical block ranges. Thus, hot block-ranges are given a proportionally larger share of the overall write buffer than cold ranges.

4. EVALUATION

We built a trace-driven FTL simulator in C++ to explore the trade-offs in FTL layouts and optimizations. The simulator models a flash device as a contiguous set of erase blocks. It models both the log-based and range-mapped FTLs.

4.1 Trace workloads and metrics

In order to evaluate the FTLs being discussed, we use both synthetically generated traces and real traces captured from a NetApp file server, and an HP Labs file server.

4.1.1 Synthetic workloads

- **Uniform**: This workload simulates a uniformly random workload by uniformly distributing the starting LBN across the device. The run length is uniformly distributed. The uniform workload represents the best case for wear leveling, because every segment or range container is accessed uniformly. However, it results in a high number of copy outs due to the uniform fragmentation.
- **X/Y**: This workload simulates a workload which has a set of hot and cold logical blocks; X% of the I/Os go to Y% of the LBNs. Run length is uniformly distributed. This workload tests the effectiveness of wear leveling, because wear is not evenly distributed. We use a 90/10 distribution.
- **LOGX**: This workload simulates a log file system workload, where the parameter X approximates the internal fragmentation. The workload cycles sequentially across the device. Each LBN is selected with a probability equal to X%. Sequential LBN runs are issued as a single I/O.

4.1.2 Real workloads

We also use traces we collected from an enterprise NetApp file server running the log-based WAFL file system [5] and one month of the Cello 1999 trace [3] from HP Labs.

- **NetApp TPC-C**: TPC-C is an online-transaction processing workload [21]. We ran the open-source TPCC-UVA benchmark [13] over NFS against a PostgreSQL 8.3 [20] database for eight hours, overwriting the device twice.
- **NetApp SPC-1**: SPC-1 simulates business-critical applications characterized by random I/O [19]. SPC-1 was run over iSCSI for 48 hours, overwriting the device over five times.
- **HP Cello 1999**: The Cello 1999 traces are from a file server at HP Labs hosting home directories and other general material for the storage group. We extracted traces from the month of April for a single disk.

4.1.3 Evaluation criteria

To better quantify wear leveling overhead, we use the *write amplification* metric. Write amplification is the ratio between the number of bytes written due to page copy outs versus the number of bytes written due to I/O. A write amplification of 0 indicates no overhead, and a write amplification of 1 indicates an overhead of 100%. Unless specified, the synthetic traces have a size of 10 GB, with 20% over-commit and write ≈ 25 million logical blocks.

4.2 Log-based FTL

This subsection evaluates the Log-based FTL. Unless specified, we use a segment size of 32MB and $\alpha = 0.8$.

4.2.1 Overprovisioning

Figure 5 shows the effects of overprovisioning. For the Uniform and the LOG20 workloads overprovisioning allows for more overwrites before cleaning starts, resulting in a larger number of dead pages enabling more efficient cleaning, thus lowering the write amplification.

Overprovisioning has little effect for the 90/10 and the LOG80

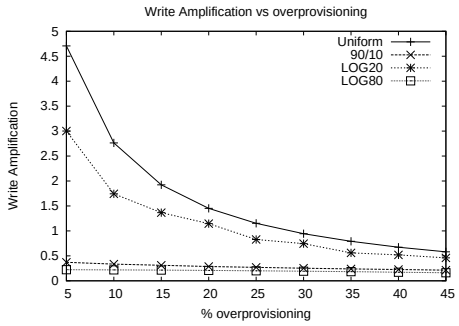

Figure 5: Log: Write amplification vs over-provisioning.

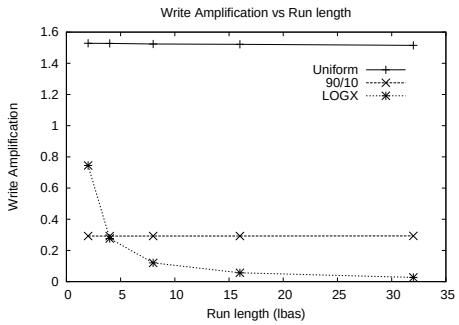

Figure 6: Log: Write amplification versus run length.

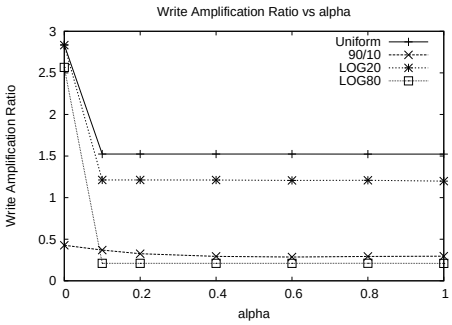

Figure 7: Log: Write amplification versus alpha.

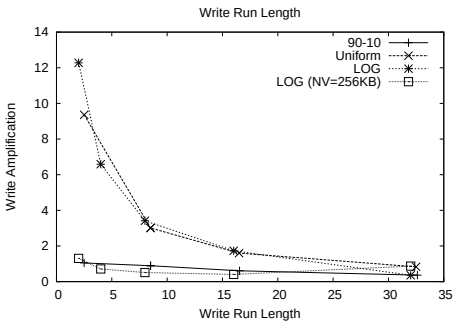

Figure 8: Range container: run length sensitivity.

workloads, as a number of segments have a large dead page count due to the high temporal locality of these workloads.

4.2.2 Run length

Figure 6 shows the effect of run length on write amplification. As run length has no effect on the distribution of overwrites in the Uniform and 90/10 traces, there is no change in write amplification. The probability of a logical block being overwritten is similar for the LOG and Uniform workloads, the sequentiality of the LOG workload results in overwrites that are much more localized, both in time and space. This results in more overwrites to the same segments and a low write amplification. Thus, run length significantly helps only when it results in more overwrites to the same logical blocks.

4.2.3 Segment-size

The Uniform and LOG20 workloads distribute their overwrites evenly across all the segments, increasing the segment size results in a segment having more live pages increasing write amplification. The large number of overwrites in the 90/10 and the LOG80 workloads cancels the effect of increasing the segment size.

4.2.4 Wear leveling: effects of α

Figure 7 shows the effect of α on write amplification. Because the Uniform and LOG workloads have no hot spots, the segments wear evenly even at high values of α. However, within a segment, there is a large standard deviation in the number of live pages. Because the 90/10 trace is not uniform, write amplification decreases with increasing α. As α increases, the emphasis on utilization increases which results in the hotter segments being selected for cleaning. Because hot data is more likely to be written to these segments, they remain in use longer, which adversely affects wear leveling.

When only wear is considered, at $\alpha = 0$, all segments are cleaned with equal likelihood, which results in a higher write amplification. Since the wear of all segments is similar, any non-zero α value prioritizes segments based on utilization, lowering write amplification.

4.3 Range-mapped FTL

This subsection evaluates the range FTL. By default, each range container holds 32 4KB logical blocks.

4.3.1 Run length

Figure 8 shows the impact of average write run length on write amplification. The general trend shows write amplification decreasing as run length increases. Because more of the range container is invalidated on each I/O, fewer live pages are copied out. The 90/10 workload exhibits low write amplification, because it has a higher locality of overwrite. LOG and Uniform exhibit similar write amplifications because their I/Os are distributed uniformly across the ranges. Notice how a small amount of NVRAM (256KB) can drastically improve the performance of the LOG-based workload.

4.3.2 Range container size

All results increase linearly as range container size increases due to an increased number of live blocks per copy out; the slope differs by workload. The 90/10 workload exhibits the lowest write amplification due to the high locality of overwrites. The LOG20 workload shows a steep slope, because

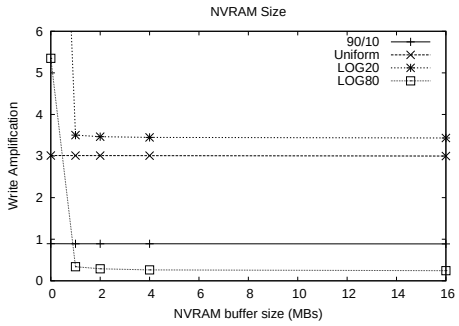

Figure 9: Range container: NVRAM write buffering.

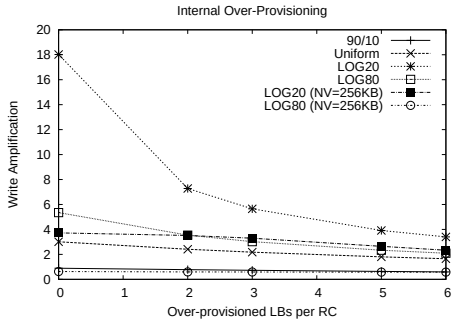

Figure 10: Range container: internal overprovisioning.

each sequential write I/O to a range container causes a copy out of that range container. This is amplified by the small run length per write I/O, thus the LOG80 workload, with a higher average run length, shows a dramatically decreased slope.

4.3.3 Write buffering

Figure 9 examines the effects of adding a small amount of NVRAM acting as a write buffer. Similar to the flash-log, a NVRAM write buffer has a huge impact on reducing the write-amplification for the LOG20 and LOG80 workloads. However, unlike the flash-based log, the 90/10 or uniform workloads suffer no ill effects.

Recall that the LOG workloads write mostly sequential data, interspersed by holes. The write buffer gathers the individual small write I/Os and issues them as a single large I/O that invalidates a large portion of the range container, resulting in a low write amplification. When issued individually to a full range container, the small individual I/Os invalidate a very small portion of the range container, resulting in a much higher write amplification.

4.3.4 Overprovisioning flash

We examine two methods of overprovisioning range containers to delay copy outs. The first, assigns extra pages within each range container, this is shown in Figure 10. The second, assigns unused range containers to hot ranges.

Overprovisioning within a range container evenly distributes the extra space across all range containers. As expected, all workloads are improved. However, the LOG20 shows a dramatic improvement due to the buffering of the tiny write I/Os.

Assigning overflow range containers to hot ranges concentrates the overprovisioned flash at hot-spots. However, doing so is only useful if the workload is non-uniformly distributed. As expected, the 90/10 workload benefits the most.

		TPC-C		*SPC-1*		*Cello*
		no del	del	no del	del	no del
Log	Write Ampl.	0.01	0.006	0.28	0.05	0.51
	Mean Wear	2.06	2.05	6.3	5.2	48
	4KB Metadata Pages	230	1947	19k	37k	1165
Range	Write Ampl.	0.34	0.01	0.70	0.09	2.3
	Mean Wear	2.75	2.07	8.8	5.7	69
	4KB Metadata Pages	750		2207		104

Table 1: Real-workloads summary.

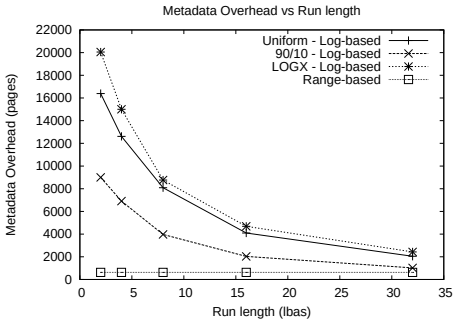

Figure 11: Metadata overhead versus run length.

4.4 Real workloads

We ran the TPC-C, SPC-1 and the HP Cello traces against the FTLs. TPC-C and SPC-1 are similar to the LOGX workload due to WAFL. Cello is similar to an 85/15 (X/Y) workload. However, unlike their respective synthetic traces the run lengths are not uniformly distributed.

The results are summarized in Table 1. For SPC-1 and TPC-C the first column shows the impact of not having block de-allocations, and the second includes de-allocations. Cello has no block de-allocations. The range FTL uses a 256KB NVRAM buffer, without which write amplification doubles.

The TPC-C trace has an average run length of 31.4 with a maximum of 32 and high temporal locality. Because it was run on a clean log-structured file system, there is very little internal fragmentation. This is arguably the best case for the log-based FTL.

The SPC-1 trace has a maximum run length of 32 with an average of 11.25. The longer run time causes aging, resulting in fragmentation. Fragmentation makes it harder to find long chains of sequential logical blocks, thus reducing run length.

Of the traces, Cello has the smallest average run length of 2.9. This is not unexpected, because it utilizes a write-in place file system, rather than a log-based file system.

4.5 Metadata Overhead

The log FTL's reduction of copy outs comes with a high metadata overhead. In general, the increased metadata of the log FTL requires more NVRAM buffering and more I/Os for metadata lookups, which impacts both cost and performance. Figure 11 shows the metadata overhead of the two FTLs versus

run length. An increase in run length decreases the metadata overhead, due to the larger extent sizes; this is clearly seen in the TPC-C workload. The range-based FTL has a much lower, fixed metadata footprint.

5. RELATED WORK

Kawaguchi et al. first proposed using a log-structured FTL to provide a block interface to flash [9] with a cleaner similar to LFS's [17].

Kim et al. propose the BAST FTL for small compact flash devices. Using an overflow scheme that chains log blocks off the primary data container, similar to the range container overflow containers [10]. FAST [12] optimizes BAST by creating a single log partitioned into random and sequential segments.

FTLs often use data temperature for wear leveling. Gal and Toledo survey a number of algorithms and data structures used by FTLs [4] for wear leveling, metadata-management and cleaning, including algorithms that use temperature. Jung et al. [8] reduce metadata overheads by dividing the device into larger groups. For wear leveling they swap hot blocks with cold blocks in different groups. Hsieh et al. use multiple hash functions to track hot-data blocks [6]. Wu and Zwaenpoel proposed a log-based flash virtual memory layer that used SRAM as a write buffer and for storing metadata [23].

More recently, Agrawal et al. have described a flash device-access model for use in high-performance SSDs [1]. Some of their design is based on the work done by Birrell et al., who found that most available SSDs have poor write performance [2]. They use fine-grained mapping tables in addition to storing data structures in RAM. However, their design requires a scan of flash at startup to regenerate the mapping tables.

6. CONCLUSIONS

In this paper, we present the design and implementation of two common FTL class. We use trace-based simulation using both synthetic and real traces to analyze the trade-offs of these FTLs and to explore the impact of each workload. We find that spatial and temporal locality have a huge impact on the wear and number of copy outs a device endures. Additionally, we explore the impact of proposed changes to existing block protocols to allow for a tighter coupling between the file system and the FTL by passing down block de-allocation events to the FTL. Using these new interfaces, we find that write overheads can be more than halved.

7. REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *Proceedings of the USENIX Annual Technical Conference*, June 2008.

[2] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *Operating Systems Review*, 41(2):88–93, 2007.

[3] SNIA IOTTA repository: block I/O traces. http://iotta.snia.org/traces/list/BlockIO, 1999.

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

[5] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter Technical Conference*, January 1994.

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

[7] Intel Corporation. Understanding the flash translation layer (FTL) specification, December 1998.

[8] D. Jung, Y.-H. Chae, H. Jo, J.-S. Kim, and J. Lee. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 160–164, New York, NY, USA, 2007. ACM.

[9] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proceedings of the Winter USENIX Technical Conference*, pages 155–164, 1995.

[10] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for compact-flash systems. *IEEE Transactions on Consumer Electronics*, 48(2), May 2002.

[11] F. Knight. SBC-3 thin provisioning commands. Technical Report 08-149r7, INCITS Technical Committee T10, December 2008.

[12] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *Trans. on Embedded Computing Sys.*, 6(3):18, 2007.

[13] D. R. Llanos. TPCC-UVa: an open-source TPC-C implementation for global performance measurement of computer systems. *SIGMOD Rec.*, 35(4):6–15, 2006.

[14] M-Systems. Flash-memory translation layer for NAND flash (NFTL).

[15] C. Manning. YAFFS: yet another flash file system. http://www.aleph1.co.uk/yaffs, 2004.

[16] K. W. Parker. Portable electronic device having a log-structured file system in flash memory. US Patent 6,535,949, 2003.

[17] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[18] F. Shu and N. Obr. Data set management commands proposal for ATA8-ACS2. Technical Report e07154r6, INCITS Technical Committee T13, December 2007.

[19] Storage Performance Council. SPC benchmark official specification, revision 1.3. http://www.storageperformance.org, March 2003.

[20] The PostgreSQL Global Development Group. PostgreSQL 8.3.3 documentation. http://www.postgresql.org/docs/8.3.

[21] Transaction Processing Performance Council. TPC benchmark C standard specification, revision 5.9. http://www.tpc.org, June 2007.

[22] D. Woodhouse. JFFS: The journaling flash file system. In *Ottawa Linux Symposium*, July 2001.

[23] M. Wu and W. Zwaenepoel. envy: a non-volatile, main memory storage system. In *Proceedings of the 6th international conference on Architectural support for programming languages and operating systems*, pages 86–97, 1994.