LSDM: Improving the Performance of Mobile Storage with a Log-Structured Address Remapping Device Driver

Aviad Zuck
Tel-Aviv University
Email: aviadzuc@tau.ac.il

Oren Kishon
Tel-Aviv University
Email: orenkish@tau.ac.il

Sivan Toledo
Tel-Aviv University
Email: stoledo@tau.ac.il

***Abstract*—Mobile devices use low-cost SSDs such as microSD cards and eMMC devices for persistent data storage. However, The controllers of low-cost SSDs are optimized for reads and for sequential writes and they perform poorly under random writes. In this paper, we show that it is possible to overcome this limitation using a novel device driver *on the host*. Our driver, called LSDM, uses design techniques normally used in the firmware (FTL) of high-end SSD to transform random writes to sequential ones. Our driver is a generic kernel module that interfaces an existing file system to the block device that represents the SSD, transforming the arbitrary read/write request sequence of the file system to a sequence with long streams of sequential writes; even low-cost SSDs perform well on such sequences. The use of an existing unmodified file system allows users and administrators to benefit from mature, feature-complete file systems. Our prototype implementation speeds up all `filebench` workloads used, by up to a factor of 6. Our experiments show that a flash-friendly file system that generates long streams of sequential writes delivers performance that is similar to a conventional file system mounted on LSDM. The high complexity of a completely new file system, relative to the simplicity of LSDM, favors our solution.**

I. Introduction

Mobile devices (i.e. smartphones, tablets etc.) use low-cost flash-based Solid-State Disks (SSDs) , such as eMMCs (Embedded MultiMediaCards) and microSD cards, to store data. These storage devices are a major factor determining the pricing and performance of mobile devices [18]. Vendors use them due to their low price, combined with their shock-resistent nature, and low power consumption. However, some price/performance tradeoff is inevitable. The tradeoff manifests itself mostly in terms of the device's IO/s rate (IOs per second). Sequential write and all read access patterns usually achieve good performance. However when writing random blocks performance deteriorates significantly.

The position of a low-cost SSD on the price/performance spectrum is determined to a large extent by the small amount of RAM that their controller has. There are other determining factors, such as the raw performance of the NAND chips that are used, but the size of the device's RAM is very significant [4]. Most importantly, the amount of RAM is responsible for essentially all the difference between random-write and sequential write performance.

It has been widely recognized that sector/page-granularity log-structured mapping structures are key in any solution to this problem [16], [4], [26], [29], [25]. Table I lists different ways to apply this technique. The different solutions differ in the software layer at which they apply the technique (the SSD's firmware, the block device driver, or the file system) and whether and where mappings are cached. The number and variety of the solutions, some on the market and some proposed, demonstrates how important this problem is; it must be solved. However, it is not yet clear which one will win.

In this paper we generalize several such solutions [28], [24], [21], [20]. We show that our solution is easily applicable to mainstream operating systems and SSDs and that it performs well under a variety of workloads.

The key idea in these solutions (and in our system, LSDM) is to deploy a log-structured address-translation layer between the file system and the block device driver that manages a low-cost SSD. This layer transforms the random writes that the file-system layer emits into long streams of sequential writes using a log-structured approach. This technique was originally proposed for magnetic disks [11]; it did not gain much acceptance on disks, probably because it can hurt spatial locality, which slows down reads. SSDs usually deliver similar performance on sequential and random reads, so they are better candidates for this technique.

Current solutions for low-cost SSDs are specialized; they either require two storage devices, using a special user space library, maintaining large data structures in RAM, perform redundant writes that weare out the device, or they are designed for a specific application on a platform with an unusual NAND flash storage. Our paper makes several important contributions.

- A mapping mechanism that works well with only one NAND storage device (typically a low-cost SSD whose random write performance can be arbitrarily bad) without impacting overall system performance. We demonstrate the mechanism using a Linux device driver; we expect that similar drivers for other operating systems should be easy to implement.
- Our mapping mechanism uses an amount of RAM that is proportional to the capacity of the SSD, and can be easily reduced even further to improve performance. This is particularly significant for large SSDs (or for systems with small RAM, like phones and tablets).
- Our design writes the data once, and does not require redundant copies of the data from a staging area on the device like some other recently-proposed schemes.

Table I.

	Storage Device	Firmware	File System	Flash Hardware	Random Writes
Low-end SSDs	SSD, small RAM	no change	no change	standard	slow
High-end SSDs	SSD, large RAM	no change	no change	standard	fast
FusionIO	PCI Express card	custom	no change	custom	fast
Specialized File System [25], [18], [1]	SSD, small RAM	no change	custom	standard	fast
Hinting [5]	SSD, small RAM	custom	no change	standard	fast
LBS [28]	SSD, small RAM	no change	no change	custom	fast
Advil [21]	SSD, small RAM	no change	no change	custom	fast
FlashLite [20]	SSD, small RAM	no change	interface with library	standard	medium
ReSSD [24]	SSD, small RAM	no change	no change	standard	medium
LSDM (this paper)	SSD, small RAM	no change	no change	standard	fast

- Our garbage collection mechanism better matches the performance characteristics of low-cost SSDs.
- We demonstrate that LSDM improves the performance of systems that use a low-cost SSD under a variety of common workloads. The improvements achieved with LSDM and a mature file system (ext4) currently used in popular mobile devices, are better than or similar to the performance improvements obtained by switching to a log-structured and flash-friendly file systems. This is important because log-structured and flash-friendly file systems for mainstream operating systems are either not available or are immature.

The design of LDSM is described in Section 3; it is driven by performance measurements summarized in Section 2. Our prototype is described in Section 4, and its performance in Section 5; Section 6 reviews related work and Section 7 discusses our results and presents our conclusions.

II. Performance Characteristics of Low-Cost Solid-State Disks

The performance of random writes on SSDs with small RAMs is typically three orders of magnitude slower than sequential writes, and reads (of any pattern). The performance of repeated aligned write requests (say blocks of 4 KB on 4 KB boundaries) in an SSD can often be modeled quite accurately by a simple affine model

$$t_{\text{write}} = \alpha n + \beta$$

where t_{write} is the time the SSD spends on serving the request, n is the size of the request in bytes, α is the throughput of the SSD in seconds/byte on large requests, and β is a latency parameter. The throughput parameter α is the same for sequential and random writes, but in low-cost SSDs the β parameter is small for sequential writes and huge for random writes. Misaligned writes incur an additional penalty that is irrelevant to our discussion.

Figure 1 shows the actual write performance and the predicted performance according to an affine model on typical low-cost SSDs - 16 GB Kingston USB Data Traveler, 8 GB Sandisk USB Cruzer, 16 GB Sandisk Extreme SDHC, and 16 GB Patriot LX Pro SDHC. The model was generated using weighted least-squares, where the weights were chosen in a way that minimizes the sum of squares of the *relative* errors. The figure shows that the model is fairly accurate for the Sandisk Cruzer; the maximum relative prediction errors were 7% for random writes and 15% for sequential writes. The throughput α was modeled as 22 MB/s for random writes and 23 MB/s for sequential; quite similar. The latencies (β's) were dramatically different: 222 ms for random writes but only 1 ms for sequential writes. The performance of the rest of the devices is non-monotonic, so it cannot be accurately modeled by a simple monotone model. The non-monotonicity manifests itself in better performance for medium-length bursts of sequential writes (64 KB) than for longer bursts, or bursts of small 4 KB writes. This is essentially a defect. The maximum relative prediction errors were 86% for random writes and 80% for sequential writes for these devices. We demonstrate in the figures that removing outliers improves the fit of the model.

The latency of sequential writes is determined by the latency of the interconnect and the latency of raw NAND program operations.

The high latency of random writes in low-cost SSDs is mostly due to a coarse-grained mapping mechanism in the FTL of the SSD. Coarse-grained mappings map large aligned consecutive ranges of LBAs to consecutive aligned ranges in the physical addresses space of the SSD (consisting of NAND chips or planes, erase units, and pages). The size S_{FTL} of these ranges is usually an erase unit or constant number of erase units, depending on how much RAM the SSD uses for the mapping table.

SSDs with relatively large RAM can use a page-level mapping, in which any page-size range of LBAs can map to any flash page. The random-write latency of these SSDs is much smaller than in low-cost SSDs, on the order of the sequential-write latency [5], [4]. Such SSDs are typically expensive.

Some low-cost SSDs utilize both a coarse-level mapping and a page-level mapping [23], [17], [16]. A coarse-level mapping maps most of the address space. A page-level mapping maps a relatively small amount of storage. Such schemes are useful when a only a small subset of the address space is subject to random writes. There are many ways to decide how to map a particular LBA, but all of them perform poorly when the random-write access pattern persists for long stretches of time and covers a significant part of the address space.

The size of individual requests influences performance significantly, but only up to a point. Figure 2 presents the throughput of writing random aligned blocks of m bytes, where each such block is written using m/n I/O requests of size n each, sequentially. For example, one particular data point in Figure 2 (Sandisk 8 GB) shows the throughput achieved when writing 8 MB sequentially in chunks of 64 KB, jumping to a different, random 8 MB block and filling it using sequential 64 KB requests, and so on. The data shows that as long as individual I/O requests transfer 64 KB or more, the performance is about the same as when each request transfers

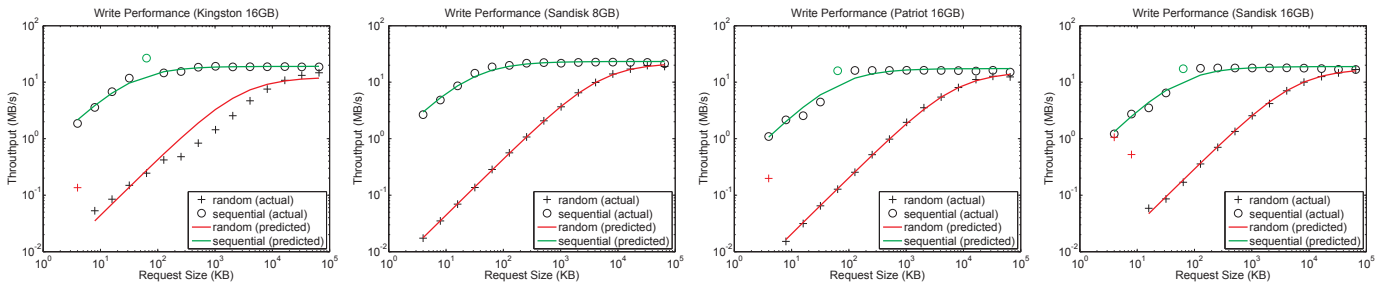

Figure 1. The effect of request size (in blocks) on sequential and random write throughput. Random writes throughput increases as we use larger the request size, while sequential throughput is almost unaffected. The colored dots are outliers that we removed to improve the fit.

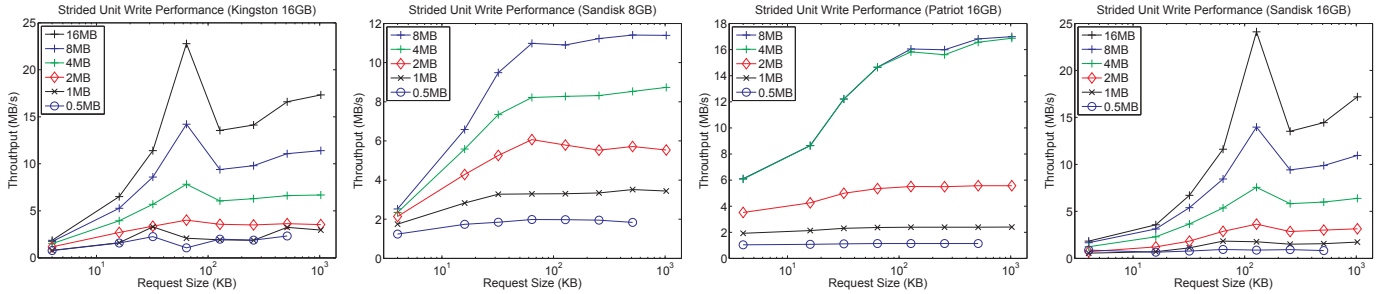

Figure 2. The throughput of a low-cost SSD when writing chunks of size n sequentially within random blocks of size m, where n (the x axis) ranges from 4 KB to 1 MB and where m ranges from 0.5 MB to 16 MB (the different data series).

the entire block of size m. I/O requests smaller than 64 KB degrade performance significantly.

This behavior was observed by others, such as Chen [9], who exploited it in database logging, an application that generates small sequential write requests.

III. The Design of LSDM

LSDM is an address-mapping block device driver. It translates block addresses presented to it by a file system to block addresses that it requests from a lower-level block device, which we assume represents a (low-cost) solid-state disk, as shown in Figure 3. The performance properties of LSDM require it to be the sole client of the SSD, so the lower-level device should normally represent an entire SSD rather than a partition. Read and write commands to the SSD specify *logical block addresses* (LBAs). These are translated again to NAND flash physical devices by the FTL of the SSD.

LSDM exposes to the kernel a block-device interface. This device is virtual (in the sense that LSDM performs address translation and buffering), so we call the addresses that it exposes *virtual addresses*[1]. From the kernel's perspective, these addresses are also LBAs (they form the address space of a block device), but in this paper we reserve the term LBA to the addresses that the SSD exposes.

Block devices serve read and write requests for blocks whose size is a multiple of a sector size and that are aligned on sector-size boundaries. Magnetic disks expose 512-byte sectors. LSDM also exposes a 512-byte block size, but its data structures are configured to support efficiently aligned

[1]The term *virtual addresses* is used in this paper in a way that has nothing to do with virtual memory.

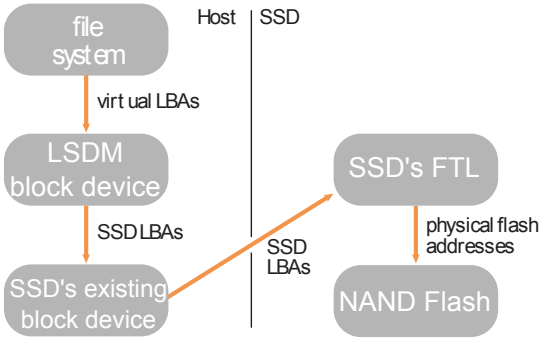

Figure 3. The role of LSDM in the storage stack.

requests of a block size that matches the file-system's request size (4 KB). We refer below to this size as the *sector size*.

LSDM maintains an arbitrary mapping of aligned sector-sized blocks of the virtual address space (*virtual sectors*) to aligned sector-sized blocks of the LBA address space (*LBA sectors*). To ensure high performance, the mapping is represented by a flat table in RAM. We also maintain in RAM a bitmap that specifies whether each LBA sector is valid or obsolete. The LBAs mapped to virtual sectors are not overwritten in place. When the mapping of a virtual sector changes, the LBA sector that used to store it becomes obsolete.

LSDM tries to avoid the performance penalty associated with small random writes. To avoid this penalty, LSDM partitions the LBA-sector space into fixed-sized units that we call *garbage-collection units* (GC units). Each garbage collection unit is further partitioned into *write units*. The size m of garbage collection units should be large enough that $\alpha m/\beta$ is large, where α (throughput) and β (latency) are the

model parameters of random writes. A large m amortizes the performance penalty of random writes (the large β). At an $\alpha m/\beta \approx 1$ we expect LSDM to achieve a throughput of about 50% of peak; higher ratios can yield even higher throughput. The size n of write units should be larger than the location of the knee in the graph of Figure 2 for the GC unit size that was selected (if the graph is non-monotonic, n should be close to a maximizer in the graph; for our Kingston device, n should be 64 KB). A large n eliminates the performance penalty associated with small I/O requests.

Beyond some point, making m and n larger does not improve the I/O-transfer performance much, but it can degrade the overall performance in other ways that we explain below. Therefore, the best performance of LSDM is achieved when m and n are selected based on estimates of α, β, and the location of the knee in the throughput graph. A simple utility can test the SSD and estimate these values.

LSDM serves write requests by appending the data to be written to a buffer of size n. When the buffer contains n data bytes minus one sector, LSDM prepends a metadata sector to the buffer and writes it to the SSD. Obviously, both the mapping table and the valid-obsolete bitmap of LSDM are updated accordingly. The metadata sector contains log records that specify the changes to these two RAM data structures, as well as a monotonically-increasing sequence number. Taken together, the metadata sectors in all the write units on the SSD constitute a log that LSDM can read after a crash to recover its data structures. By writing these log records to the beginning of each write unit we avoid the need to support a secondary stream of small sequential writes (which would be required if the log was stored separately) and we avoid the need to checkpoint LSDM's data structures to the SSD. During normal shutdown, the data structures are written to a fixed LBA address; we assume that the SSD performs wear leveling, so this simple checkpointing scheme does not cause wear imbalance. A checksum of the entire write unit contents is stored in the metadata sector; during crash recovery, this checksum tells LSDM whether the most recently-written unit was written completely or incompletely.

The client of a block device can instruct it to flush pending writes to persistent storage. LSDM responds by filling in the rest of the write buffer with valid sectors from the GC unit that is currently being garbage collected (see below) and then flushing the write buffer to the SSD even if it is not full.

Writing the buffer to LBA addresses $\ell_i, \ldots, \ell_{i+k}$ overwrites the data in these LBA sectors. Therefore, these sectors must all be obsolete (except if the buffer contains virtual sectors that are currently mapped to $\ell_i, \ldots, \ell_{i+k}$). As all log-structured storage systems must do, LSDM must garbage collect in order to create obsolete CG-unit sized ranges of LBA sectors (recall that LSDM uses all the LBAs in a GC unit sequentially, not only the LBAs within a write unit).

LSDM currently uses a greedy lazy garbage collection strategy. At any given time one of the GC units is being garbage collected and another is used for all the writing. We denote the first one by g and the second by w. LSDM starts collecting garbage by reading the valid pages in the first write unit of g into the write buffer. It then waits for write requests from its client (typically a file system). These requests are added to the buffer. When the buffer fills, it is appended to w and LSDM reads the valid pages from the next write unit in g. After m/n such phases, g is completely obsolete and w has been written in its entirety. LSDM now sets g to w (data will now be written to g) and it selects a new g greedily: it selects the GC unit with the most obsolete pages.

Collecting lazily (one write unit at a time rather than a full GC unit at once) enhances garbage-collection performance by giving sectors more time to become obsolete. This strategy is mostly greedy, but it does trade greed for sequentiality within GC units. That is, after collecting one write unit, LSDM does not select the next most obsolete write unit to collect; it continues sequentially within the GC unit in order to ensure that writes of entire GC units are sequential. This design decision is justified by the performance characteristics of low-cost SSDs, which we have presented in Figure 2.

A. Prototype Implementation

We implemented LSDM as a driver for the Linux 3.X kernel. The driver processes block IO requests from a simple IO scheduler. It emits different block IO requests to the underlying driver of the SSD. The requests that LSDM emits to the underlying driver do not pass through the kernel's buffer cache, so blocks are only cached once, using the LSDM device and the virtual LBAs.

B. Possible Enhancements

Several known techniques can make LSDM more flexible or more efficient. These techniques and the tradeoffs that they entail are mostly well understood and outside the scope of our research, so we have not implemented them in LSDM and we do not explore them. We list them here for completeness.

Currently our design requires about 1 MB of RAM per 1 GB of SSD capacity . Most of this RAM is used to store the virtual-to-LBA mapping. However, the design of LSDM can be easily modified to store the mapping on flash and to cache only a subset of the mappings in RAM, using a scheme similar to that of DFTL [16].

Another issue is booting after a shutdown (whether planned or unexpected). This requires LSDM to reconstruct its mapping data structures by reading all metadata sectors from the SSD. A typical write-unit size is 64 KB, so assuming a sector size of 4 KB, this means that to recover the entire state from flash we will have to read 6% of the entire device. If necessary, a checkpointing mechanism can be added to modify the tradeoff between write performance and post-crash boot time.

Copying valid pages from GC unit g to unit w eagerly when no client requests are pending can improve responsiveness in some settings. It can also be triggered to minimize the number of write buffers being flushed to the device in response to synchronous requests. Eager garbage collection tends to perform more writes than lazy collection, because an eager collector copies pages that might have become obsolete under a lazy collector. Background collection is a tradeoff, not a straightforward enhancement.

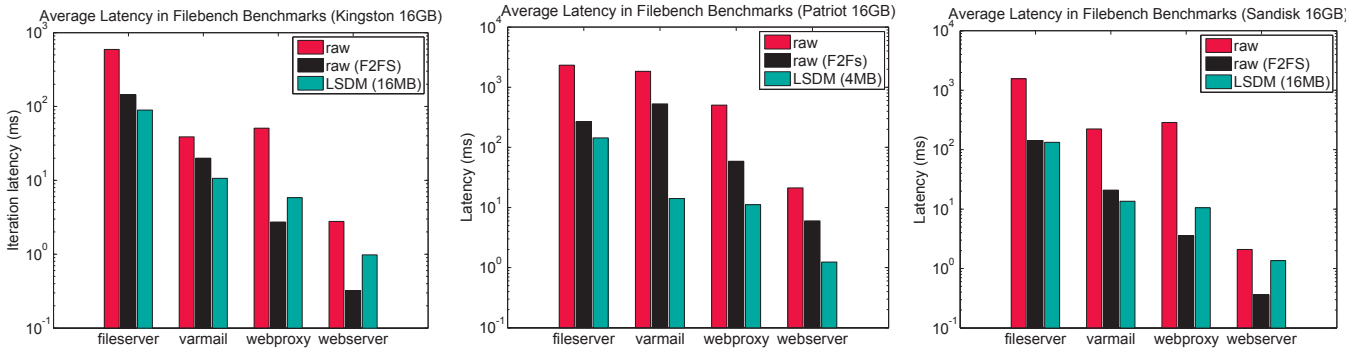

Figure 4. The average latency of file system IO operations under a variety of `filebench` workloads. The raw performance was measured once when the ext4 file system was mounted directly on the SSD and again with the nilfs2 file system. The performance of LSDM was measured with ext4 mounted over LSDM, which was mounted on the SSD's block device. In LSDM runs the size of garbage-collection units was 16 MB (4 MB for the Patriot device) and its write buffer size was 64 KB.

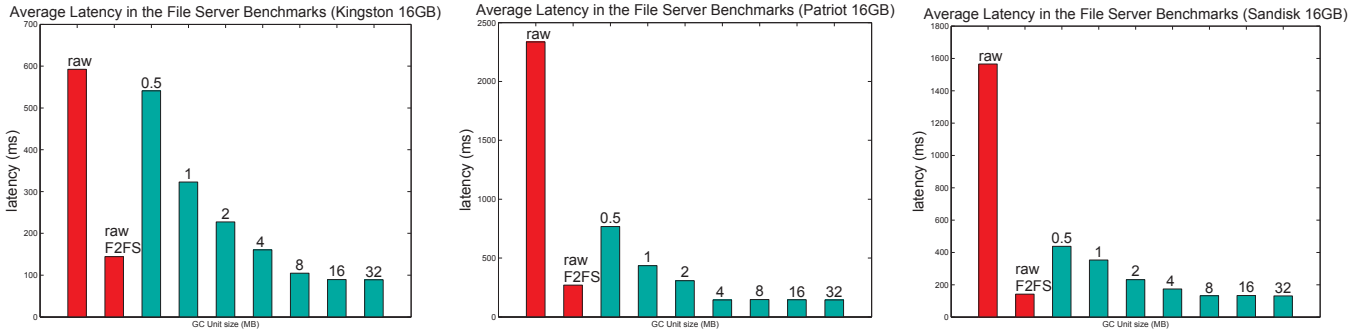

Figure 5. The average file system IO latency of file system IO operations in `filebench`'s fileserver workload. The setups are the same as in Figure 4, except that in LSDM runs the size of garbage-collection units was varied from 0.5 MB to 32 MB. The results for the webserver, webproxy and varmail benchmarks were similar.

IV. Evaluation

A. Experimental Setup

We performed the experiments on all devices except the 8 GB Sandisk Cruzer[2]. We used an Intel Atom 330 machine running at 1.60 GHz and using 2 GB of RAM. The machine ran Ubuntu Raring Ringtail with a Linux version 3.9.0 kernel.

We evaluated performance using `filebench` [2], a popular file system benchmarking tool. `filebench` measures several metrics, including average latency, throughput of file system requests, and CPU cycles. The ones that we presented here are the the average latency of file system I/O operations, which is closely related to overall wall-clock time with Figures 4 and 5). Due to lack of space of omitted throughput results, though they were similar. To minimize the effects of the buffer cache the file system was mounted using a commit time of 1s (the default is 5). We compared the performance of the workload under three setups:

- An `ext4` file system that is mounted directly on the SSD's block device. Ext4 is the default file system for Android-baesed mobile devices [18]. We refer to this setup as a *raw* device.
- An `f2fs` [22], [1] file system that is mounted directly on the SSD's block device. We refer to this setup as a *raw (f2fs)* device. This file system is flash friendly; as such, it generates mostly sequential write requests to the block device, even if the workload accesses files and/or locations within files randomly. F2FS is not completely mature, but it appears to be a promising alternative to legacy file systems designed for magnetic media.
- A `nilfs2` [22] file system that is mounted directly on the SSD's block device. We refer to this setup as a *raw (nilfs2)* device. This file system is log structured; as such, it generates mostly sequential write requests to the block device, even if the workload accesses files and/or locations within files randomly. nilfs2 is not completely mature and stable, but it is considered to be more stable and better supported than other log-structured file systems for Linux (e.g., [3]).
- An `ext4` file system that is mounted on an LSDM block device which uses the SSD as its backing store.

We began each experiment with a priming phase that is designed to ensure that the file system, LSDM, and the SSD are in realistic steady states, rather than particularly clean. The priming phase creates a new file system and then repeatedly creates and deleted large files. We then create large files that are never deleted which take up 35% of the file system's capacity. At this point we run the workload under test. Every experiment was repeated 5 times.

[2]The 8 GB SanDisk Cruzer whose out-of-the-box performance was reported in Figure 1 and 2 was too worn out for the experiments reported here and we did not find an identical device to test.

In all experiments, LSDM was configured to 25% overprovisioning, and to the optimal write-unit and garbage-collection unit-size for each device (64 KB write-unit size on all devices, 16 MB/4 MB/16 MB garbage-collection unit size for the Kingston/Patriot/Sandisk devices accordingly). The size of the garbage-collection units is justified by Figure 1 (and by Figure 5) and the size of the write units is justified by Figure 2.

Due to the hardware limitations of our machine and the small capacity of the SSD being used, we modified one of `filebench`'s workloads (the file-server workload) in minor ways. The original workload would not complete without running out of memory or sufficient storage to run the tests.

B. Results

Figure 4 compares the performance of the three setups under various `filebench` workloads that emulate behaviors of common applications. LSDM improves performance significantly in all workloads over the raw setup with ext4, and delivers comparable performance to that of F2fs and outperforms it in many of the workloads, especially those that are more I/O-intensive. nilfs2 could not sustain the stress of most workloads and crashed, therefore we opted not to include it in our figures.

Both LSDM and F2fs have the same overall effect on the access pattern that the SSD sees. In both cases, the SSD sees long streams of write requests (interspersed perhaps by read requests). We do not know why the performance of LSDM differs from that of F2fs; we believe that this is due to slight variations in the access patterns that they produce, or to differences in their garbage collection mechanisms.

In some cases F2fs outperforms LSDM on the *webserver* and *webproxy* benchmarks. In these workloads all setups display a relatively low throughput. These workloads are dominated by accesses to small files, which are read constantly by "clients" (threads). Due to the small request sizes these workloads are mostly dominated by open, close, delete and create operations. These operations result in synchronous metadata block requests which contribute to the slightly worse latency.

Figures 5 shows how the size of GC units affects performance in the fileserver benchmark. We did not include similar figures for the webserver, webproxy and varmail benchmarks, since the results were similar. As evident from the figures, LSDM outperforms the raw setup even when using only a 0.5 MB GC unit. Performance improves appreciably as the GC unit size grows from 0.5 MB to the the optimal GC unit size for every device, but does not improve much beyond that.

This experiment demonstrates the benefit of large GC units. Small GC units make the greedy collection policy more effective, but they reduce overall performance because of the poor mapping strategies of low-cost SSDs. Overall, large GC units perform better; even 2 MB is clearly too small, and less is terrible.

V. Related Work

Various other solutions have been suggested to improve the random-write performance of SSDs. High-end SSDs have a large built-in RAM that allows them to use a page-level translation layer (FTL) [4], [16] that performs a log-structured or write-anywhere mapping and delivers high performance. A large RAM can also improve performance through caching and de-duplication [8], [19]. SSDs connected to the host's memory bus, such as Fusion-io's PCI Express devices (e.g. [14]), can use the host's RAM for their translation layer, but they are typically even more expensive than high-end SSDs with storage interconnects (SATA, SCSI, FC, etc).

The host's RAM can be used to accelerate page-level address translation in SSDs with little RAM and with relatively-slow storage interconnects by having the host attach mapping hints to IO requests [5]. This solution, however, requires modifying both the SSDs firmware and the host's device driver.

Log-structured file systems [27], [22] have been shown to deliver high performance on SSDs [25], [26]. However, given the enormous investment in existing mature file systems such as NTFS [10] and ext4 [6], it is unclear whether log-structured file systems would actually replace them.

LLD [11] proposed to put the log-structured address translation in a new layer in the storage stack, below the file system but above the disk. LLD was aimed at magnetic disks and the motivation for it was the same as for LFS, the original log-structured file system [27]. All log-structured mapping structures improve the spatial locality of write requests (they are mostly sequential), but they can hurt the spatial locality of read requests. On magnetic disks, any random request is expensive, whether it is a read or a write request. This appears to be the main reason that neither LFS nor LLD became popular; they hurt read performance too often.

On SSDs, the spatial locality of writes is important, because erase blocks are larger than pages (the write unit), but the spatial locality of reads is not a significant performance determinant. Therefore, several solutions using log-structured mapping structures above the storage stack were suggested.

LBS [28] was conceived as a solution to a rather specialized problem, that of running multiple virtual machines on a smartphone or tablet running Android. Android devices have two non-volatile flash devices, a so called internal storage and a so-called external SD card (the external card is removable but fits within the device's enclosure). The authors of LBS assumed that VM images would be stored on the external storage. They designed LBS to improve the performance of random writes to this device. Because they also had access to a second flash device, the internal one, they used it to store the mapping metadata of LBS, thereby avoiding several complications. LBS does not run on computers with only flash storage device. Even on computers with more than one flash device, users would probably not want to dedicate one to the metadata of another one, so LBS is not a general purpose solution.

A commercial solution called Managed Flash Technology [12] has also been suggested to improve the random write performance of SSDs. However, there are no details about its architecture and garbage collection mechanism, and the problem of sync commands. It is also known to have a strict requirement of about 1 MB of kernel memory for every 1 GB of capacity, probably for its large non-flexible mapping data structures.

FlashLite [20] is a user-level library designed to eliminate random writes generated when downloading large files through P2P protocols. Writes to the file are remapped and written sequentially to temporary files. FlashLite maintains mapping information in memory for the original locations of these writes. When the download is complete the data is re-written sequentially to its original location.

ReSSD [24] partitions the address space of the SSD to a large area dedicated for data and sequential writes, and a small "reserved" staging area used as a sequential log for small random writes. ReSSD remaps these writes, and occasionally writes them back to their original location. This approach eliminates most small random writes and would perform well for large SSDs with multiple chips, but not as well for low-cost SSDs since it generates repeated writes to various random locations on the device.

Advil [21] is an improvement on ReSSD which also takes into account the temporal locality of data. Advil is aimed at low-cost SSDs on mobile devices by placing the so-called "reserved" area on the external memory of a mobile device, and the data area on the so-called "internal" memory card.

Our paper takes the approach of eliminating small random writes by using a remapping layer above the storage stack, but takes it one step further and generalizes this solution to the more common single-SSD setup. LSDM also fixes two other problems not addressed in previous solutions; we explain this in the next section.

VI. Discussion and Conclusions

Low-cost SSDs are a major factor in the performance and pricing of mobile devices. However they perform poorly under a random-write access pattern. The literature already describes several techniques to address this challenge; some are already on the market. In this paper, we show that an address mapping device driver on the host can solve the problem, without requiring hardware or firmware changes and without replacing the file system.

Our driver, LSDM, improves performance by up to a factor of 6 when coupled with a low-cost SSD and a mature industrial strength file system (ext4) used by default in Android-based mobile devices, when compared to a setup in which the file system is mounted directly on top of the SSD. Switching from ext4 to a log-structured file system (nilfs2) resulted in numerous crashes; this does not imply that the ideas behind nilfs2 are flawed, of course, but it does show that implementing a complete file system is a major engineering challenge. Switching to a flash-friendly file system (F2fs) mounted directly on top of the SSD also improves performance, but not as much for IO-intensive workloads. F2fs appears to be more mature than nilfs2, but replacing the file system remains a significant engineering and system administration concern.

We claim that LSDM improves performance significantly over conventional file systems mounted on top of low-cost SSDs. It has an inherent performance advantage over log-structured file systems, and it has proven stable and on par with the performance of a new flash friendly file system. LSDM is a driver that is significantly simpler than file systems, making our solution more usable than F2fs and similar efforts.

LSDM has several important advantages over similar designs. First, it runs on a single SSD, so it targets a much wider range of systems than LBS [28] and Advil [21] which require using a secondary storage device. Second, LSDM's garbage collection mechanism leads to very long stretches of sequential writes (many MB), which helps the SSD deliver high performance; in comparison, other designs occasionally jump to random write locations [28], [21] which may degrade performance significantly, or they perform redundant copies of all data from a staging area to their actual location [24]. Third, LBS [28] uses about 12 MB of metadata for each 1 GB of SSD capacity, whereas LSDM uses only about 1 MB of RAM for each GB of flash. This is not a lot to pay for a 2x to 6x performance improvement on realistic workloads and makes LSDM much more memory efficient, which is particularly important on systems with small RAM or with large SSDs. Lastly LSDM is implemented as a generic kernel driver without any modification to existing applications and file systems. In contrast, FlashLite [20] requires applications to use a special user-space library and also suffers from any underlying file system fragmentation.

The utility of a log-structured page-level mapping to achieve high performance in flash storage has already been widely recognized. Do we want to have enough RAM in each SSD to store the page-level mapping? Judging from the measured cost/performance characteristics of existing low-cost SSDs used for mobile storage, the answer is no; SSDs with enough RAM are expensive. Do we want to switch to log-structured or flash-friendly file systems? Probably not yet. Our investment in mature and stable file systems may be too high to forgo. This may be particularly important since flash devices for mobile devices will not be the only non-volatile storage systems that we will need to deploy in the next few years. Despite many predictions in the past, magnetic disks are still the mainstream choice for primary storage, and many predict they will continue to be useful in the near future [15]. Magnetic disks do not deliver high performance when log-structured file systems are mounted on them, and are agnostic to of flash-friendly file-systems. Moreover, emerging technologies such as PCM-based disks [13] and/or shingled magnetic disks [7] may become prevalent, and log-structured or flash-friendly file systems may not be appropriate for them either. We better stay with our current file system and adapt the software stack below it. Solutions such as LSDM may emerge as the most appropriate solutions.

VII. Acknowledgments

This work was supported by the Israeli Ministry of Science, Technology and Space.

References

[1] F2fs homepage. https://wiki.archlinux.org/index.php/F2fs

[2] Filebench benchmarking tool project homepage. http://sourceforge.net/apps/mediawiki/filebench. Original tool by Richard McDougall

[3] Lfs: A log structured file system for linux that supports snapshots. http://logfs.sourceforge.net

[4] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J.D., Manasse, M.S., Panigrahy, R.: Design tradeoffs for SSD performance. In: USENIX Annual Technical Conference, pp. 57–70. USENIX Association (2008). URL http://www.usenix.org/events/usenix08/tech/fullpapers/agrawal/agrawal.pdf

[5] Budilovsky, E., Toledo, S., Zuck, A.: Prototyping a high-performance low-cost solid-state disk. In: Proceedings of the 4th Annual International Conference on Systems and Storage (SYSTOR), p. 10 pages. ACM (2011)

[6] Card, R., Tso, T.Y., Tweedie, S.: Design and implementation of the second extended file system. Proceedings of the Amsterdam Linux Conference (1994)

[7] Cassuto, Y., Sanvido, M., Guyot, C., Hall, D., Bandic, Z.: Indirection systems for shingled-recording disk drives. In: Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, pp. 1 –14 (2010). DOI 10.1109/MSST.2010.5496971

[8] Chen, F., Luo, T., Zhang, X.: CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In: Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST), pp. 77–90 (2011). URL http://www.usenix.org/events/fast11/tech/techAbstracts.html#Chen

[9] Chen, S.: Flashlogging: exploiting flash devices for synchronous logging performance. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 73–86 (2009)

[10] Custer, H.: Inside the Windows NT File System. Microsoft Press (1994)

[11] De Jonge, W., Kaashoek, M., Hsieh, W.: The logical disk: A new approach to improving file systems. ACM SIGOPS Operating Systems Review **27**(5), 15–28 (1993)

[12] Dumitru, D.: Optimizing flash storage with linearization software (2009). Flash Memory Summit

[13] Eilert, S., Leinwander, M., Crisenza, G.: Phase change memory: A new memory enables new memory usage models. In: Memory Workshop, 2009. IMW '09. IEEE International, pp. 1 –2 (2009). DOI 10.1109/IMW.2009.5090604

[14] Fusion-io: ioDrive data sheet (2012). http://www.fusionio.com/data-sheets/iodrive-data-sheet/

[15] Grupp, L.M., Davis, J.D., Swanson, S.: The bleak future of nand flash memory. In: Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST'12, pp. 2–2. USENIX Association, Berkeley, CA, USA (2012). URL http://dl.acm.org/citation.cfm?id=2208461.2208463

[16] Gupta, A., Kim, Y., Urgaonkar, B.: DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. ACM SIGPLAN Notices **44**(3), 229–240 (2009)

[17] Kang, J., Jo, H., Kim, J., Lee, J.: A superblock-based flash translation layer for NAND flash memory. In: Proceedings of the 6th ACM/IEEE International conference on Embedded Software (EMSOFT), pp. 161–170 (2006)

[18] Kim, H., Agrawal, N., Ungureanu, C.: Revisiting storage for smartphones. In: Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST), p. 14 pages (2012)

[19] Kim, H., Ahn, S.: BPLRU: a buffer management scheme for improving random writes in flash storage. In: Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST), pp. 239–252 (2008)

[20] Kim, H., Ramachandran, U.: Flashlite: A user-level library to enhance durability of ssd for p2p file sharing. In: Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on, pp. 534–541. IEEE (2009)

[21] Kim, J.M., soo Kim, J.: Advil: A pain reliever for the storage performance of mobile devices. In: Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on, pp. 429–436 (2012). DOI 10.1109/ICCSE.2012.66

[22] Konishi, R., Amagai, Y., Sato, K., Hifumi, H., Kihara, S., Moriai, S.: The Linux implementation of a log-structured file system. ACM SIGOPS Operating Systems Review **40**(3), 102–107 (2006)

[23] Lee, S., Shin, D., Kim, Y., Kim, J.: LAST: locality-aware sector translation for NAND flash memory-based storage systems. ACM SIGOPS Operating Systems Review **42**(6), 36–42 (2008)

[24] Lee, Y., Kim, J.S., Maeng, S.: Ressd: a software layer for resuscitating ssds from poor small random write performance. In: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10, pp. 242–243. ACM, New York, NY, USA (2010). DOI 10.1145/1774088.1774138. URL http://doi.acm.org/10.1145/1774088.1774138

[25] Min, C., Kim, K., Cho, H., Lee, S.W., Eom, Y.I.: SFS: Random write considered harmful in solid state drives. In: Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST) (2008)

[26] One, A.: YAFFS: Yet another flash filing system (2002). `http://www.yaffs.net`

[27] Rosenblum, M., Ousterhout, J.: The design and implementation of a log-structured file system. ACM Transactions on Computer Systems **10**(1), 26–52 (1992)

[28] Tuch, H., Laplace, C., Barr, K.C., Wu, B.: Block storage virtualization with commodity Secure Digital cards. In: Proceedings of the 8th International Conference on Virtual Execution Environments (VEE), pp. 191–202. ACM (2012)

[29] Woodhouse, D.: JFFS: The Journalling Flash File System (2001). Ottawa Linux Symposium, available at `http://sourceware.org/jffs2/`