Prototyping a High-Performance Low-Cost Solid-State Disk

Evgeny Budilovsky
The Blavatnik School of Computer Science
Tel-Aviv University
Tel-Aviv, Israel
budevg@gmail.com

Sivan Toledo
The Blavatnik School of Computer Science
Tel-Aviv University
Tel-Aviv, Israel
stoledo@tau.ac.il

Aviad Zuck
The Blavatnik School of Computer Science
Tel-Aviv University
Tel-Aviv, Israel
aviadzuc@tau.ac.il

ABSTRACT

We present a design for a high-performance low-cost solid-state disk (SSD). Ignoring garbage-collection costs, our SSD performs only $1+\varepsilon$ physical accesses to NAND flash pages for every request of a page-size block by the host, for some small ε. This is true for all access patterns, including random writes, which are usually slow on low-cost SSDs. Garbage collection in all SSDs is determined primarily by how full the SSD is, and its cost is similar in most SSDs. The unique feature in our design is that it achieves high performance even with when the SSD contains only a small amount of RAM. In most SSD designs, this would imply low performance; in ours, it does not. A small RAM lowers the cost of an SSD with a given flash array. Our design achieves high performance with a small RAM using two innovative ideas. One is the use of a clever mapping data structure. The second is a host-assisted hinting mechanism that uses RAM on the host to compensate for the small amount of RAM within the SSD. This mechanism is implemented as an enhanced SCSI driver (kernel module). Our prototyping methodology is also a significant contribution. We simulate the SSD in software, using files to represent the flash array, but the resulting prototype is a working SCSI device that file systems can be mounted on.

Categories and Subject Descriptors

D.4.2 [**OPERATING SYSTEMS**]: Storage Management—*Allocation/deallocation strategies, Garbage collection*

General Terms

Design, Algorithms , Performance, Experimentation

Keywords

Flash, NAND flash, iSCSI, Hints, Host Assisted, Page Mapping

1. INTRODUCTION

Solid-state disks still provide dismal poor random write (and sometimes random read) performance. Benchmarks of SSDs have shown that many of them perform poorly on random-write workloads [5, 2]. Random read performance is often excellent, on par with sequential reads. This poor random-write performance is caused by a coarse-granularity mapping of host sector addresses (or linear block addresses, LBAs) to physical page addresses.

For example, the specification of the SiliconEdge Blue SSD [9] lists sustained sequential read and write performance of 250 and 140 MB/s, respectively, but random read and write performance of only 20 MB/s. Benchmarks show even worse slowdowns for random accesses: 13 and 15 MB/s for random reads and writes, respectively[1], versus 229 and 176 MB/s for sequential reads and writes [21].

One proposed solution to this problem relies on a log-structured design that writes pages sequentially within flash erase blocks even when the host writes to random LBAs. This requires a fine-grained mapping mechanism that admits arbitrary mappings of host sectors (or small contiguous clusters covering 4KB or 8KB) to flash pages [5]. However, this solution assumes that the SSD has enough RAM to store a full sector-to-page mapping table.

For a 256GB Silicon Blue SSD, a table that maps each 4KB host sector to an arbitrary flash address requires about 256MB of RAM; alas, the SSD only has 64MB of RAM. Even so, the SSD, which is designed for laptops, is at least 10 times more expensive than a high-end magnetic hard disk (HDD) for laptops . The SSD performs 15 times better on random I/O and about 2 times better on sequential I/O than an HDD.

It is clear that a large RAM would improve the random-I/O performance [5] of SSDs. Still, the designers of recent SSD have opted for a small RAM (still much larger than the RAM buffer of a comparable HDD) and relatively poor performance. Assuming that the cost of NAND flash chips will continue to drop, the cost of a large RAM will become even more significant.

Moderately-priced SSDs from other vendors also perform relatively poorly on random writes [14, 13] (These Intel drives perform well on random reads, which is also the case for many other SSDs [2]; Intel's high-end SLC and low-end MLC drives differ only in sequential performance, not in random-I/O performance).

High-end SSDs perform better, but at a much higher cost. The Zeus-IOPS SSD [12] performs sequential writes at 115MB/s sequentially and random writes at 64MB/s; but it is 30 times more expensive per GB than [13, 14, 9].

SYSTOR'11, May 30–June 1, 2011, Haifa, Israel.
Copyright 2011 ACM 978-1-4503-0773-4/11/05 ...$10.00.

[1] The relatively poor random I/O performance of the SiliconEdge is not due to garbage collection (GC) overhead. When the benchmark wrote randomly to the entire drive, performance dropped to 2-3MB/s, similar to an HDD, due to the overhead of GC. The specified random-write performance was achieved only when the writes were limited to a small portion of the drive, which reduces the overhead of GC. The fact that random reads are also slow also suggests that the slowdown is due to mapping overhead, not to GC.

This dramatic price-performance may seem fundamental, but it is not.

In this paper we describe the prototype of a high-performance SSD that uses a small amount of RAM. Our design relies on two ideas to achieve high performance. One, described in Section 3, is a two-level log-structured mapping data structure that can deliver good sequential and random performance with the amount of RAM that is now used in low-end SSDs. A host-assisted hinting mechanism, described in Section 4, improves performance from good to excellent by caching mapping information on the host's large and inexpensive RAM. The relationships between our techniques and other issues in SSD design are explained in Section 5.

The prototype is implemented in software, using simulated NAND flash chips that store information in files. The prototype is a SCSI device, which allows us to test it under real file systems and real workloads. The host side of the hinting mechanism is a kernel module that modifies the behaviour of the SCSI disk driver in Linux. The SSD prototype can work with existing SCSI device drivers; the hinting mechanism is not used, but the SSD works and its fine-grained mapping mechanism delivers good performance. The prototype, which is freely available under the GNU Public License[2], is described in Section 6.

Section 7 presents experimental results that support our performance claims and Section 8 presents our conclusions. The next section discusses related work.

2. RELATED WORK

The mapping mechanisms of commercial SSDs are proprietary, but the literature does describe several mapping schemes. Some designs use a block-level mapping, sometimes with mechanisms that avoid a read-modify-write cycle on every random write [18, 16]. Kang et al. [15] partition the sector space into super-blocks that are mapped into erase-block groups; each group contains data blocks and log blocks that are merged once in a while. LAST uses a block-level map-ping for most of the data and a page-level mapping for a small subset of the data [19]. Birrell et al. [5] proposed a page-level mapping with a flat mapping table in RAM; this requires a large RAM in the SSD.

Our design is a generalization of DFTL [11], which also uses a two-level page mapping. DFTL uses page-sized mapping chunks, which create a very large overhead for random writes; its authors assume that the access pattern exhibits locality. Mapping chunks are cached in RAM, and if the hit-rate is low, performance degrades. Our small chunks work well even under a totally random workload with no locality. Another difference is that DFTL caches mapping entries in the SSD's RAM whereas we also cache them on the host, where RAM is plentiful.

Arpaci-Dusseau et al.[3] proposed recently to eliminate entirely the mapping from the SSD. In their proposed design, the SSD informs the host where it stored a particular block; the host is responsible for keeping appropriate mapping information. In other words, the host and the SSD communicate using physical flash addresses rather than LBAs. Such designs must cope with address changes that result when the SSD reclaims blocks or performs wear levelling. Our design is somewhat similar in that the host maintains the mapping to physical addresses, but because the host's mappings are treated as hints, we do not need to keep the host and the SSD completely consistent. On the other hand, our prototype does store the mapping on the SSD. In Arpaci-Dusseau et al.'s [3] design, the SSD decides on the address of a new data block; NANDFS [24] uses a similar nameless writing scheme for the interface between the file-system layer and block layer.

Parallelism has emerged as a major concern in recent papers [20, 1]. Our prototype is event driven and concurrent; it can drive multiple flash chips on multiple buses, but parallelism is not the main concern of this paper.

Hints are pieces of information that are usually correct, that improve performance when correct, but are allowed to be incorrect. The utility of hints has been known for a long time (see [17] and the references there pertaining to hints). We are not aware of uses of hints in SSD designs.

The SCSI framework that we use to prototype our SSD, tgt [10], has been used in several research projects [8, 7]. SCST [6] is a similar framework, but it runs in the kernel rather than in user mode.

3. THE MAPPING DATA STRUCTURE

The mapping structure that we propose maps every 4 KB cluster (which we refer to as a sector below) to an arbitrary 4 KB page on flash. We store each entry in 4 bytes, which allows for SSDs of up to 16 TB. The mapping entries are stored in contiguous *chunks* of c entries; c is a parameter of the design. The chunks are stored on dedicated erase units, but usually not in order. An array in RAM, called the root array, stores a pointer to the on-flash physical address of each chunk, and a 4-byte version number associated with each chunk; version numbers are discussed later. The total size of this array is $8n/cp$, where n is the total size in bytes of all the sectors stored in the SSD and p is the size of each sector. In a 1 TB SSD with 4 KB sectors and 64-entries chunks, the root array occupies 32 MB.

The design also uses a bitmap in RAM that specifies whether each sector-size portion of flash is in use. Its size is $n/8p$; For the 1 TB unit, the size is also 32 MB. This amount of RAM required is reasonable and it remains so even if the SSD's capacity is scaled up (it is up to 32 times smaller than the amount required to keep the entire mapping in RAM). In SSDs that do not have enough RAM for the bitmap, the bit map can be replaced by an array of counters, one for each block. The counters specify the number of obsolete pages in a block, to allow the garbage collector to choose blocks for cleaning; the counters require only a $1/32$ fraction of the space of the bitmap. The disadvantage of the counters is that they force the garbage collector to determine the current mapping of each sector on a page that is being cleaned, which increases the number of read latencies incurred during cleaning. The host-assist mechanism described below can also be used to represent an approximation of the bitmap with little RAM usage on the SSD, and to assist in verifying the validity of reclaimed physical pages without having to read the relevant mapping chunks from flash.

To find sector s on flash, the controller fetches entry $\lfloor s/c \rfloor$ of the root array; it points to the physical flash address of the chunk containing the mapping of s. The controller reads the chunk. Because chunks are read without reading an entire page, each chunk is protected by a separate error-correction code (ECC). We note that on most modern NAND chips, reading a small fraction of a page is significantly faster than reading the entire page (because transfer times on the NAND bus are relatively long).

To serve a read request, the controller now reads the requested sector from flash and sends it to the host. To serve a write request, the controller uses the mapping of s to invalidate the old copy of s (in the bitmap), queues the new sector to be written to an arbitrary erased page, modifies the chunk containing s to show the new mapping, and queues the chunk to be written to flash as well.

The mapping data structure is illustrated in Figure 1; Figure 2 shows how the SSD changes it when the host writes data to an LBA.

[2] The code is available at `http://www.tau.ac.il/~stoledo/research.html`.

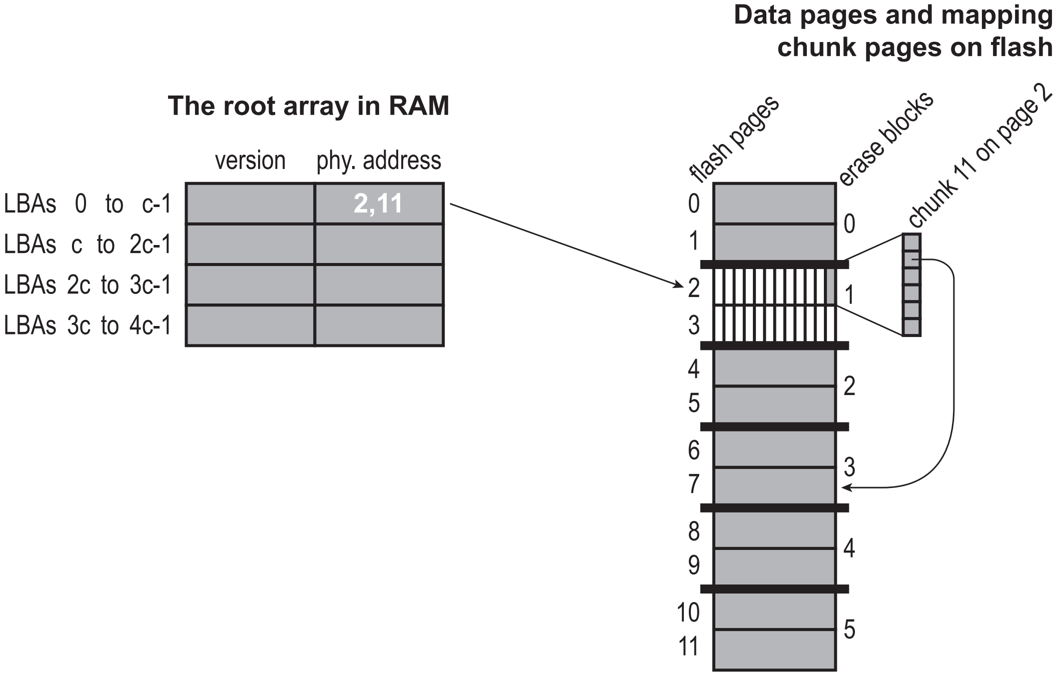

Figure 1: An illustration of the two-level mapping mechanism. On the left we see the root array in RAM, and on the right the flash array, consisting of 6 erase units containing 2 pages each. The example shows how LBA 1 is mapped to flash. This LBA is mapped through the second entry in the 12th (and last) mapping chunk. The root array indicates that the chunk is the 12th one stored on physical page 2. The second entry in this chunk points to physical page 7. This page stores LBA 1.

We write chunks to flash lazily. Each chunk that we write to flash contains not only *c* pointers, but also a version number and their index in the root array.

Sectors written to flash are self-identifying: they are written with metadata containing an ECC and the host sector number. If the SSD crashes before writing all the queued chunks, it reads recently-written pages, determines which sectors they contain, and updates the mapping.

4. HOST-ASSISTED MAPPING

Low-cost SSDs often lack enough RAM to store an arbitrary page-mapping in RAM. But the host often has enough RAM to store this mapping, or at least a fraction with a high hit rate.

We have developed a host-assist mechanism for SSDs. This mechanism allows the host to assist the SSD in various tasks, and in particular in mapping sectors.

The mechanism allows the SSD and the host to exchange information using what is essentially an out-of-band channel that is not used for the normal read/write requests.

We have prototyped the mechanism over iSCSI. The SSD, which we prototyped using the *tgt* user-space SCSI target framework, exposes one disk LUN and one custom LUN that is used for the assist mechanism. The host sends hints to the SSD by writing to the custom LUN. The host also keeps at least one read request pending on this LUN at all times, allowing the SSD to send mapping chunks to the host whenever it chooses.

When the SSD reads or modifies a mapping chunk, it sends the chunk to the host as a response to the outstanding read request on the custom LUN. On the host, a kernel thread receives the chunk and caches it. This cache of chunks on the host is used by an interceptor function. The interceptor is invoked by the SCSI driver on every request to the SSD. For every request directed at the hint-enabled SSD, the interceptor attempts to find relevant mapping chunks. If it finds any, it writes them to the custom LUN, concatenated with the original read or write SCSI request. This constitutes a hint. The interceptor then sends the original request to the data LUN. If the host can't find the chunk, the interceptor sends only the original request. This structure is illustrated in Figure3.

When the SSD receives a hint before the actual read/write request, it uses the mapping in the hint instead of reading the mapping from flash. This saves one read latency for most random reads and writes.

To work correctly without coupling the SSD and the host too tightly, the SSD attaches a version number to chunk pointers in the root array and to the chunks sent up to the host. It uses a chunk from the host only if its version number is up to date; otherwise it ignores it.

Figure 4 illustrates how the hinting mechanism changes the mapping process, using the example of Figures 1 and 2. Note that since the version in the hint matches the version in the root array, one read operation from flash is eliminated. The figure also illustrates the fact that mapping chunks are written to flash lazily when the SSD accumulates an entire page worth of dirty mapping chunks.

The hints are small relative to the data pages; therefore, they do not consume much bandwidth on the SCSI interconnect. The default size of chunks are 256 bytes long (and map 16 LBAs; they can be configured to be larger or smaller). Other fields in the hint occupy 28 bytes.

Hints are not necessary for the SSD to function correctly. If the SSD is used with an existing SCSI driver (on any operating system), the driver will not attach to the custom LUN, will not read mapping chunks, and will not send hints. Random reads and writes would be slower than on a host that implements hinting, but the SSD would work (and would deliver good but not excellent performance).

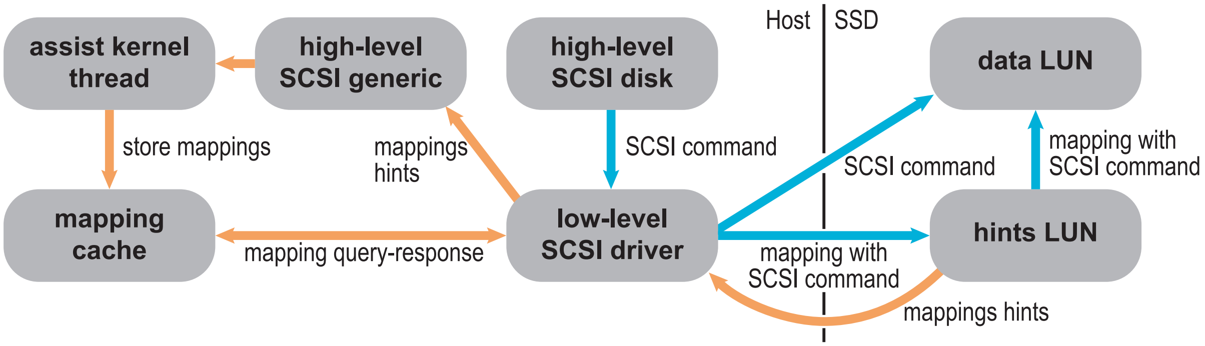

Figure 3: The hinting mechanism.

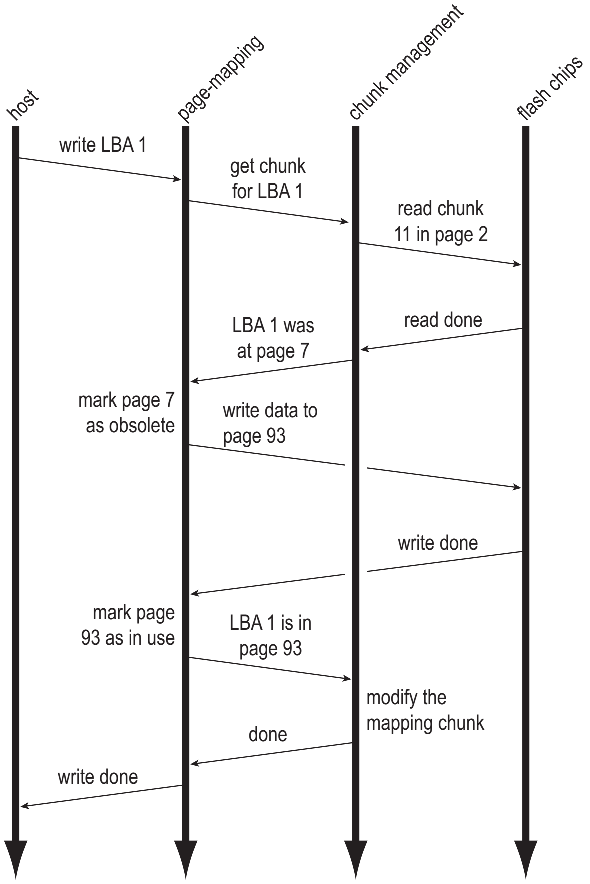

Figure 2: The sequence of requests and responses that takes place when the host writes to LBA 1, assuming that the mapping data structure starts at the state shown in Figure 1. The hinting mechanism is not active. Time progresses from top to bottom.

5. ADDITIONAL DESIGN ISSUES

Our work focuses on one aspect of SSD design, namely efficient mapping mechanisms. The design of SSDs must address a few more issues. Some of them are essentially orthogonal to the mapping and some are not. In this section, we list these issues and explain if and how they interact with our mapping mechanism.

Checkpointing and Startup.

All SSDs map LBAs to physical flash addresses using a nontrivial data structure. Some representation of the mapping must be stored on flash. Upon startup, the SSD must locate this information, read at least part of it, and initialize its mapping data structures in RAM. Like other non-volatile storage systems that rely on RAM data structures (e.g., file systems), an SSD must be able to startup even if it did not shut down properly. The checkpointing and startup mechanisms can be designed so that they are essentially orthogonal to the mapping mechanisms. We have designed such a checkpointing mechanism and verified that it works correctly with our mapping design. This checkpointing and recovery mechanism also supports mapping bad erase blocks to functional ones. The design uses conventional persistence techniques. Our analysis shows that the startup and recovery take much less than a second, even on a large SSD (512 GB). The details are described in Appendix A.

Reclamation and allocation.

Because SSDs do not overwrite sectors in place, every write request from the host turns one page into garbage, obsolete data that can be erased. Modern operating systems can also inform the SSD that a certain sector will not be read before it is written again, using the so-called *trim* command [22]. This also marks the page containing the sector as garbage. When the number of erased blocks drops below some threshold (which in theory can be 1), the SSD must select a block (or several), copy the non-obsolete data on it to erased storage, and erase it. This reclaims the storage that obsolete data occupied. For reclamation to be effective, the selected block should have as much obsolete data as possible. If the physical capacity of the SSD is much larger than the total size of LBAs that it stores and if the reclamation threshold is low, then the SSD can guarantee a lower bound on the number of pages that are reclaimed every time a block is cleaned. In other cases, reclamation might be ineffective.

The SSDs allocation policy is responsible for deciding on which erase block to write each sector. In some workloads, the allocation policy can have an effect on the effectiveness of reclamation. For example, in a workload dominated by sequential reads and writes, allocating nearby LBAs on nearby pages benefits reclamation, because many pages on a block are likely to become obsolete within a short span of time, when the relevant sectors are being sequentially overwritten. On the other hand, in a random-write workload any attempt to maintain spatial locality will only increase overheads without delivering any benefits. This would be the case, for example, in a block-level mapping mechanism.

Our design contributes to the allocation and reclamation aspects of the SSD in that it decouples them completely from the map-

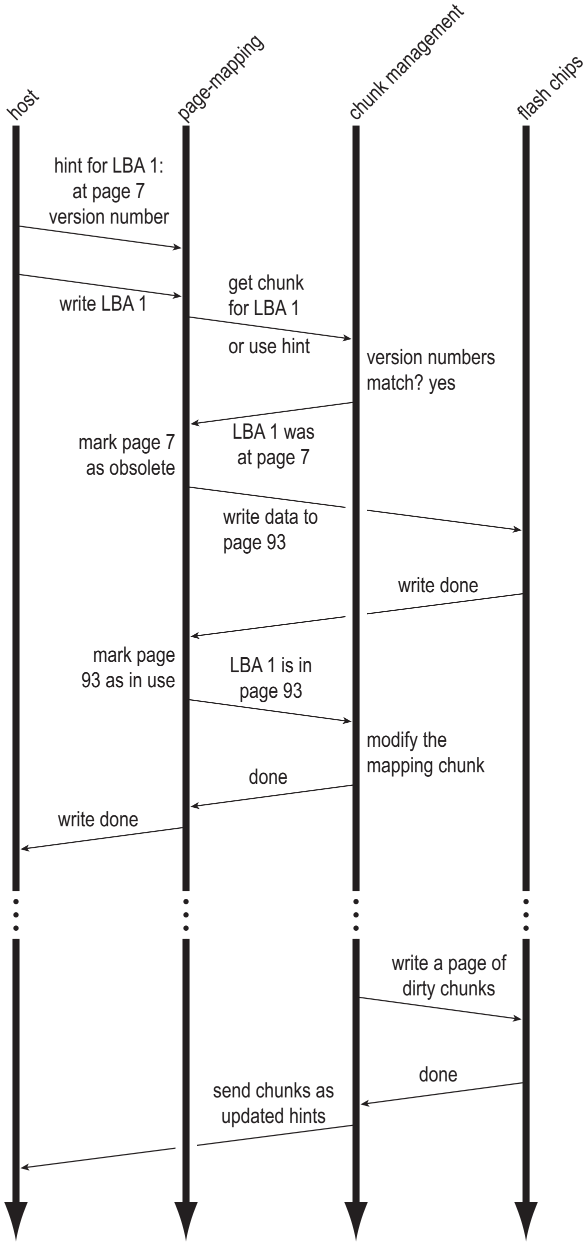

Figure 4: This example shows how the hinting mechanism eliminates one flash-read operation from the sequence of actions shown in Figure 2.

ping aspect. Our results below show that our mapping mechanism is almost as effective on random accesses as it is on sequential access. Reclamation costs are higher under random-write workloads, as they are in any SSD, but the mapping overheads (reading/writing mapping chunks) are the same. This gives the SSD designer the freedom to allocate sectors to erased pages in any way that is likely to enhance the effectiveness of reclamation. The designer can choose to co-locate nearby LBAs, or to co-locate LBAs that belong to the same application object (file or directory, if the SSD can detect this kind of information). Our mapping mechanism does not constrain the allocation and reclamation policies.

Wear Levelling.

It was previously shown that wear levelling can be completely decoupled from the mapping, allocation, and reclamation policies and mechanisms [4, 24].

6. IMPLEMENTATION

The core of our SSD prototype is a concurrent SSD simulator. The simulator simulates individual NAND chips, using a software API to represent NAND flash operations (read, program, and erase). We currently simulate only basic operations; our code currently does not support advanced operations like dual-plane reads and writes, copy-back writes, and so on, but they can be easily added. The simulator uses files to represent the non-volatile storage of flash chips. Data is transferred to and from simulated chips on simulated NAND buses. Each bus can support any number of flash chips. The use of the chip-select-don't-care feature allows one chip to perform a flash-array action (erasure, reading, or programming) while the bus transfers data and/or commands to/from another chip. The buses are driven by a controller component that handles SCSI requests, maintains the sector-to-page mapping, and performs garbage collection.

A typical SSD today has around 10 NAND flash chips and several NAND buses. To ensure that the concurrency is designed and implemented correctly, each flash chip is simulated by a separate thread. Buses are protected by locks, to ensure that a single bus is not used for two transfers at the same time.

A controller code executes SCSI requests and drives the buses and chips. The controller consists of the bulk of the SSD prototype. The controller is single-threaded but concurrent. It can drive many buses, chips, and DMA channels concurrently.

We note that even though we use a separate thread to simulate each chip, actual running times do not accurately reflect the SSD's performance. For example, when the files that represent the chips are stored on a single magnetic disk, different simulated chips cannot access their non-volatile storage concurrently, and random reads are very slow. Therefore, the simulator can theoretically keep track of simulated time, but the timing of responding to SCSI requests is not correct.

Our SSD implementation performs on-demand garbage collection. When the SSD runs out of storage, it selects the block with the largest number of obsolete pages and reclaims their storage. Idle-time reclamation would have improved performance. Our prototype does not perform checkpointing, orderly shutdown, and normal or post-crash startup. When the code starts, it initializes the SSD to an empty state. We have designed the details of checkpointing, recovery, and wear levelling using standard techniques [4, 24], but we have not implemented them.

The SSD code runs under *tgt*, a user-space SCSI framework [10]. To the host, the SSD appears as a normal iSCSI disk device. The interface between our SSD code and the rest of tgt is fairly simple: tgt calls our code when it receives a SCSI request to either the disk or the custom LUN. After processing the SCSI request inside our code, we invoke a tgt completion function to provide responses to SCSI requests.

Our host-side code consists of a single kernel module that must be loaded after the SCSI driver. We didn't change any existing kernel or SCSI drivers sources. Instead, we dynamically patch relevant callback tables to refer to our own functions. Our SSD-assist module contains the kernel thread that receives the chunks, the interceptor function that sends hints to the SSD, and the chunk cache that both of them use. When the module is loaded, it writes a pointer to the interceptor into an existing function table in the SCSI driver. This causes the interceptor to be invoked on each iSCSI request.

The interceptor overrides a function in the SCSI driver by first invoking new functionality and then calling the overridden function, to preserve the SCSI driver's functionality.

The FTL implementation and the kernel module together consist of about 2,500 lines of code.

7. EXPERIMENTAL RESULTS

We performed two main types of experiments with the SSD prototype. Both experiments used the same setup. A VirtualBox virtual machine ran a Linux kernel into which our hinting device driver was loaded. The SSD prototype ran on the same machine under tgt and it exported an iSCSI disk. The iSCSI disk was used either as a block device by a benchmark program or the ext4 file system was mounted on it.

The SSD was configured to include 8 NAND flash chips connected through 4 buses. The total capacity was kept small, 4GB, to allow us to run experiments quickly. In this configuration the amount of RAM required by our controller simulator was less than 1 MB. The small total size of the SSD should not make a significant difference in our experiments.

In one set of experiments, we mounted a file system on top of the SSD block device and ran Linux applications from that file system (mainly a kernel build). We repeated the experiments with and without hints. This verified that the prototype works correctly.

The main set of experiments is designed to evaluate the performance of the SSD. Each experiment starts by filling the block device sequentially. This brings the SSD to a state in which it must perform garbage collection. We then run `vdbench` [23] to generate block-device workloads: random or sequential reads or writes. Each request is for 4 KB, or one flash page. During the run of `vdbench`, the SSD prototype counts the number of pages written to or read from flash, the number of erasures, and the number of bytes transferred on NAND buses. We count separately operations that are triggered by SCSI requests and operations that are triggered by garbage collection. When the SSD writes pages of mapping chunks, the pages include both SCSI-triggered chunks and garbage-collection-triggered chunks. We consider all of these page write operations to be SCSI related (this makes our results seem a little worse than they really are).

The metric that we use to evaluate different SSD configurations is the average flash-chip time consumed by each SCSI operation. We compute this average by multiplying the number of pages read by amount of time NAND chips take to read a page, the number of page written by the program time, and so on, and then divide by the total number of blocks requested through the SCSI interface. The graphs below also show a breakdown of the average time to different components.

We repeated the main set of experiments 5 times, and the results presented here are the averages. The results were stable, and the largest variance exhibited between the runs was 1.5%.

The structure of the prototype does not allow us to make accurate real-time measurements, as explained in the previous section.

Figure 5 summarizes the performance benefits of our SSD design over a DFTL-like approach. The DFTL performance was evaluated by using page-size chunks and by tuning off the host-assisted hinting mechanism. We can see that reading mapping chunks accounts for a significant portion of the running time on random reads and writes. Writing chunks constitute a significant overhead for all writes because we commit the chunks immediately. Delaying chunk writes would have improved sequential-write performance significantly but not random-write performance. The performance of our SSD design is significantly better.

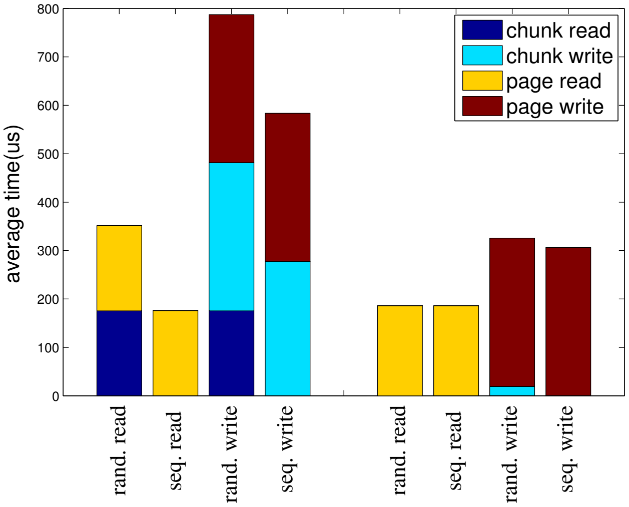

Figure 5: The performance of a DFTL-like scheme (four leftmost bars) compared to the performance of our SSD (four rightmost bars) on 4 different access patterns, all generated by `vdbench`. Each bar is broken into constituents. The average running times are estimates based on the timing of NAND flash operations and on the NAND bus throughput. We simulated each setup 5 times; the variations were 1.5% or less; this also applies to subsequent figures.

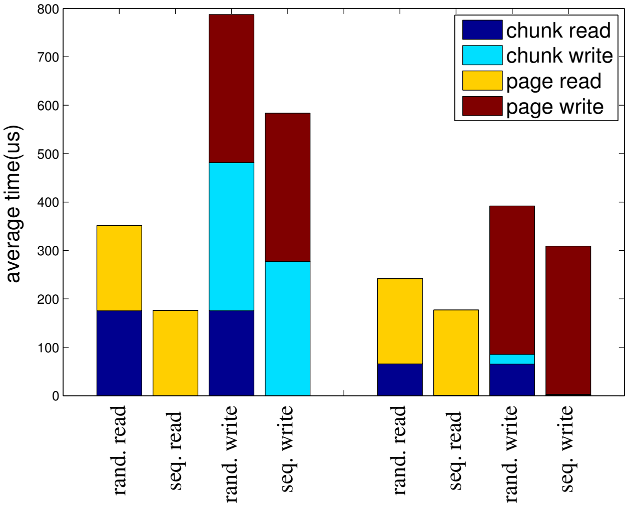

Figure 6: The benefit of using small mapping chunks. The graph compares the DFTL-like results on the left (same data as the leftmost bars in Figure 5) to our SSD's performance without the hinting mechanism.

The next graphs drill down to explain where the performance improvement comes from. Figure 6 shows the performance improvement that is due to the use of small mapping chunks. The time to write chunks is reduced because we write them only when the SSD fills a full page. The time to read chunks is also reduced because the transfer time on the NAND bus is much lower when we read a single chunk rather than a full page.

Figure 7 presents the performance benefits of using host-assisted mapping. The time to both read chunks is significantly reduced. This improves the performance of random reads and writes. In

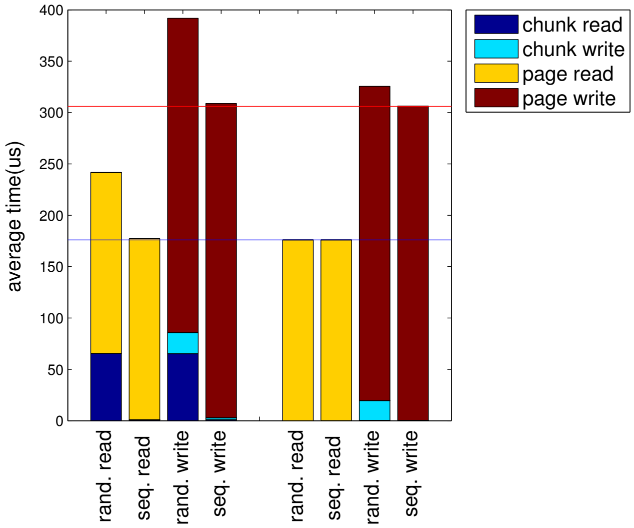

Figure 7: The effect of mapping hints on our SSD. The four bars on the left show the performance without hints (and with small mapping chunks; same data as the rightmost bars in Figure 6), and the four on the right show the performance with hints. The time to read hints almost disappears. The red horizontal line shows the cost of transferring a page from the controller to the NAND chip and programming a page. The blue line shows the cost of reading a page and transferring the data to the controller. With small chunks and hinting, performance is almost optimal for all access patterns.

particular, the use of hinting with small mapping chunks brings the cost of every read and write close to the minimum dictated by the NAND flash timing; random reads and writes are as fast as sequential ones, and they take essentially no time beyond the time to transfer a page and execute the NAND operation on the flash chip. This is the most important result of this paper.

Figures 8 and 9 explore the effects of hinting on flash devices with alternate timing characteristics. Figure 8 shows that when flash read operations are relatively slow, which is the case on some MLC NAND flash chips, the impact of the hinting mechanism is even grater. Some 3Xnm chips take about 200*us* to read a page and about 1.5ms to program a page or erase a block. The bus transfer times are similar to those of older chips. On such chips, the high cost of page read operations makes hinting particularly useful in random-read operations; writes are so slow that the use of hinting does not make a big difference. Figure 9 explores a possible future enhancement to NAND flash chips, in which the bus timing improves. In this graph, we assume a hypothetical transfer time of 5ns per byte. Again, the use of hinting becomes more significant, because the faster NAND bus makes reading chunk pages relatively more expensive.

Figure 10 demonstrates the effect of limiting the capacity of hints stored on the host. We repeated our random write benchmark with varying amounts of storage allocated to the hints cache on the host. The amounts of hint storage ranged from 20% of the amount required to keep the full mapping on the host to 100% (about 3 MB). We then measured the percentage of SCSI requests that triggered a chunk read from flash, i.e. that were not preceded by a useful hint. The data shows that there is a clear and simple correlation between the effectiveness of the hinting mechanism and the size of memory allocated in the host for storing hints. On workloads with more temporal or spatial locality, a small hints cache would be much more effective than on a random-access workload.

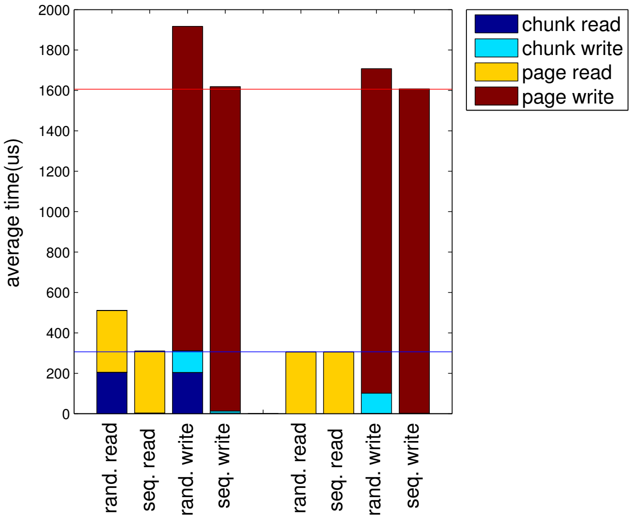

Figure 8: The impact of hints on our SSD when flash read and program times are longer (200*us* and 1.5ms, typical of some 3Xnm MLC devices) but bus transfer times remain the same. The impact on writes becomes smaller, because of the long latency, but the impact on reads becomes larger.

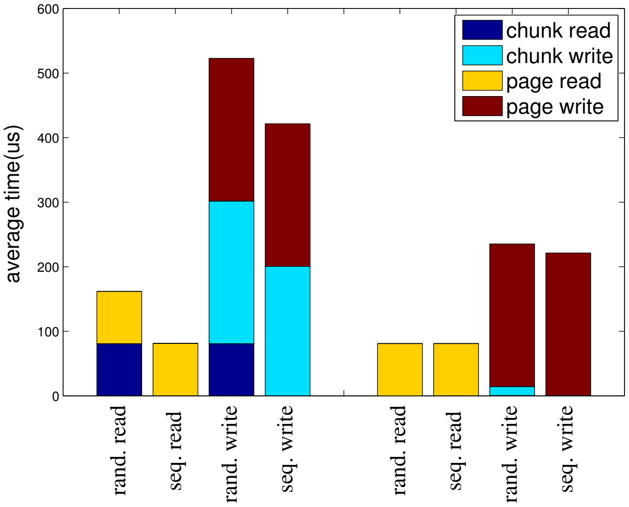

Figure 9: The impact of hints when bus transfer times drop to 5ns per byte.

8. CONCLUSIONS

This paper makes two contributions. The main contribution consists of two orthogonal techniques that together reduce mapping-related read and write amplification in NAND flash solid-state disks (SSDs). SSDs must use a complex mapping of LBAs to flash addresses, because flash erasures have large granularity and because wear levelling is used to ensure high endurance. The mapping is stored on flash and reading and updating it require performing flash operations. These operations increase the number of flash operations that are performed beyond the number required to actually

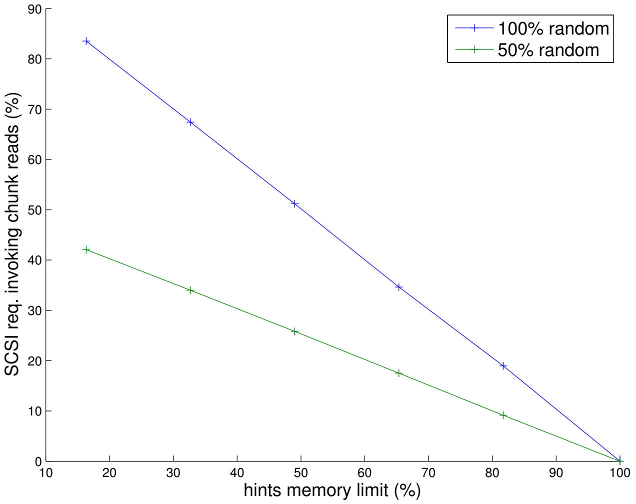

Figure 10: The effect of limiting the memory allocated for hints storage on the host-side device driver. The 100% point allows the entire page-level mapping to be stored on the host. The blue line displays the behaviour in a purely random workload, and the green line in a workload where only 50% of the requests are random and the rest are sequential.

access the data. This increase is called amplification.

The use of two-level mapping with a large root node in RAM and small internal nodes stored on flash is the first innovation of this paper. This is an enhancement of the DFTL mapping scheme, which also uses two levels. The internal mapping nodes in DFTL, however, occupy a full page. The DFTL scheme has a much larger overhead than ours on random access with poor temporal locality. It has a slightly smaller overhead on sequential accesses, but the difference is tiny.

The second technique that we use reduces read amplification. Fundamentally, we rely on a large cache in RAM for the mapping entries. The innovation in our design lies in that the cache is stored on the SCSI host, not on the SSD. In general, RAM on the SSD is expensive; most SSDs do not have enough RAM to store an entire page-granularity mapping. RAM on the host is cheaper and it can be used much more flexibly (that is, the cache can be shrunk to make RAM available to other uses).

Two ideas make a host-based cache effective. One is the fact that the SCSI device driver understands the mapping mechanism of the SSD, so it can send mapping entries with each SCSI request. By pushing mappings to the SSD when they are going to be used, we eliminate the round-trip latency that a request-reply protocol would have required. That is, it is better for the host to push the mapping than for the SSD to request it.

The other idea that makes host-based caching effective is the use of the pushed mappings as hints rather than as definitive mappings. The use of hints allows our design to avoid the complexity that would be required if the host and the SSD tried to keep consistent views of a mapping that changes all the time.

The two techniques together mean that our SSD performs only $1+\varepsilon$ physical accesses to NAND flash pages for every request of a page-size block by the host, for a very small ε. This is true for all access patterns, including random writes, which are usually slow on low-cost SSDs. The SSD is that efficient even when it has a small RAM. This shows that there is no reason to accept a memory/performance tradeoff or a sequential-access/random-access tradeoff in the design of mapping schemes for SSDs. We can have the best of both worlds.

The other main contribution of this paper is the prototyping methodology that we used to evaluate the design. By implementing the SSD as an iSCSI device we were able to achieve three objectives simultaneously. First, we were able to prototype the SSD using software alone; any prototype that uses SCSI or SATA interfaces would have required hardware design or FPGA design. Second, the SSD prototype appears as a normal block device, which allows us to mount file systems and run applications on top of it. Third, the fact that the SSD appears as a SCSI device allowed us to implement the modified SCSI driver that includes hinting support and to demonstrate that this can be done without any changes to existing Linux SCSI drivers. We believe that similar implementation techniques are possible in other operating systems. We also believe that the same principles are applicable to other SSD interfaces, such as SATA or Fibre Channel.

9. REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70. USENIX Association, 2008.

[2] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo. Characterizing the performance of flash memory storage devices and its impact on algorithm design. In *Proceedings of the 7th international conference on Experimental algorithms*, pages 208–219. Springer-Verlag, 2008.

[3] A. Arpaci-Dusseau, R. Arpaci-Dusseau, and V. Prabhakaran. Removing the costs of indirection in flash-based SSDs with nameless writes. In *Proceedings of the 2nd USENIX conference on Hot topics in storage and file systems*, pages 1–1. USENIX Association, 2010.

[4] A. Ben-Aroya and S. Toledo. Competitive analysis of flash-memory algorithms. To appear in *ACM Transactions on Algorithms*, Jan. 2007.

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

[6] V. Bolkhovitin. Generic scsi target middle level for linux. http: //scst.sourceforge.net/, 2003.

[7] D. Dalessandro, A. Devulapalli, and P. Wyckoff. Non-contiguous i/o support for object-based storage. In *Proceedings of the 2008 International Conference on Parallel Processing - Workshops*, pages 95–102, Washington, DC, USA, 2008. IEEE Computer Society.

[8] A. Devulapalli, D. Dalessandro, P. Wyckoff, N. Ali, and P. Sadayappan. Integrating parallel file systems with object-based storage devices. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 27:1–27:10, New York, NY, USA, 2007. ACM.

[9] W. Digital. *WD SiliconEdge Blue*. Lake Forest, CA, 2010. p. 2, Specifications, Available in PDF format from www.wd.com.

[10] T. Fujita and M. Christie. tgt: Framework for storage target drivers. In *Proceedings of the Linux Symposium*, 2006.

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

[12] S. Inc. *Zeus-IOPS Solid-State Drive*, 2008. p. 2, Product Brochure.

[13] Intel. *X18-M/X25-M SATA Solid State Drive*, 2009. p. 28, Product Manual.

[14] Intel. *X25-E SATA Solid-State Drives*, 2009. p. 24, Product Manual.

[15] J. Kang, H. Jo, J. Kim, and J. Lee. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 161–170. ACM, 2006.

[16] J. Kim, J. Kim, S. Noh, S. Min, and Y. Cho. A space-efficient flash translation layer for compactflash systems. *Consumer Electronics, IEEE Transactions on*, 48(2):366–375, 2002.

[17] B. Lampson. Hints for computer system design. *ACM SIGOPS Operating Systems Review*, 17(5):33–48, 1983.

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

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

[20] S. Park, E. Seo, J. Shin, S. Maeng, and J. Lee. Exploiting Internal Parallelism of Flash-based SSDs. *Computer Architecture Letters*, 9(1):9–12, 2010.

[21] T. Sullivan. Western digital siliconedge blue ssd review. in www.storagereview.com, 2010.

[22] I. T13. Data set management commands proposal for ata8-acs2 (revision 6). in www.t13.org, 2007.

[23] H. Vandenbergh. *Vdbench: a disk and tape i/o workload generator and performance reporter*, 2010.

[24] A. Zuck, O. Barzilay, and S. Toledo. NANDFS: a flexible flash file system for ram-constrained systems. In *EMSOFT '09: Proceedings of the seventh ACM international confere nce on Embedded software*, pages 285–294, 2009.

APPENDIX

A. ADDITIONAL DESIGN DETAILS

In this appendix we provide additional details on our SSD design. These details involve issues that do not affect performance to a significant degree. We have not implemented the mechanisms described here.

A.1 The Block Allocator and the Bad Block Mapper

The block allocator and the bad-block mapping mechanism use an in-RAM array that stores the state of each erasure block: *in use*, *in the process of being erased*, *erased*, or *garbage* (it needs to be erased). This array, like all the other data structures of the SSD, is committed to flash lazily; we explain the commit mechanism later.

The bad-block mapping mechanism is a very low-level one, because all the other modules require its services. Some blocks are marked on-flash by the manufacturer as bad. When we format the FTL's on-flash data structures, we scan the SSD for such blocks and we mark them in-use in the array; this ensures that they will never be allocated and hence they will never be used. We also reserve a range of blocks to serve as replacement for blocks that go bad.

Blocks may also go bad during normal operation, as indicated by a failed erasure or program operation. When an erasure operation fails, we mark the failed block as *in-use* in the in-RAM data structure. Normally, this information will commit to flash later and this ensures that the block will never be used. If the SSD crashes before this information is committed to flash, the block's state will revert to *garbage* or *in-the-process-of-being-erased*. In the latter case, we modify the state to *garbage*. Either way, the SSD will attempt to erase the block again. If the erasure succeeds, the block will continue to be used; if it fails, it will again be marked as *in-use*.

Failures during programming operations are a bit more difficult to handle. When a programming operation occurs, the SSD selects a replacement block (which is still marked in the array as erased) and writes to the metadata area of its first page a data structure that indicates that it is now replacing the failed block, and the page at which the replacement begins. The SSD then starts the programming operation again, but now directs it to the replacement block. The page offset within the block remains the same.

When the SSD boots, it reads the first page of all the blocks that have been initially reserved as replacements and builds an in-RAM data structure that specifies the current replacement mappings. This data structure is never committed back to flash. On every write and every read, the SSD checks whether the page to be read or written is in a block that has been replaced; writes are directed to the replacement block, and reads are directed to either the replaced or the replacement, depending on their offset. When a replaced block needs to be erased, the SSD erases only the replacement block; the replaced block will remain marked *in-use* forever, but since there are no references to its pages anywhere, it will never be actually used. There is no need to maintain the replacement after such an erasure.

A.2 Checkpointing and Crash Recovery

Our checkpointing mechanism is general and it is decoupled from all the other modules. Our main design objective has been orthogonality of this mechanism rather than performance. It is not inefficient asymptotically, but it is not highly optimized either.

The design assumes that all the modules whose data structures need to be checkpointed use static memory allocation and that the location of their data structures is known to the checkpointing mechanism. Our prototype satisfies these assumptions.

The other modules modify their in-RAM data structures only in the context of *transactions*. Transactions cannot span flash operations; they must end before a module initiates a flash read, program, or erasure. We therefore assume that transactions always complete in RAM (although they might not be committed to flash before a crash). The beginning and ending of transactions are reported to the checkpointing mechanism, as well as redo and undo records for all the individual RAM modification operations that constitute a transaction. Our SSD is single threaded and event driven, so there is at most one active transaction at any given time. Thus, transactions have a total order, and they commit in order. The redo records of a transaction are not allowed to span more than half a flash page.

The checkpoint mechanism stores transaction records in an in-RAM buffer whose size is at least half a flash page (it can be larger to improve performance).

A module can request that a specific transaction be committed to flash. An example for such a transaction would be changing a block's state from *garbage* to *in the process of being erased*. The checkpointing mechanism responds by committing all the transactions up to this one (and perhaps a few later ones). The checkpointing mechanism also writes to flash when its buffer contains at least half a page of transaction records. The records written back to flash must be part of a completed transaction..

Transaction records and RAM checkpoints are stored in a linked list of flash blocks. Each page in these blocks contains half a page of RAM image of the FTL's static data structures, and up to half a page of *redo* records. The undo records are only stored in RAM for internal FTL use as explained below. The half-page RAM images are copied from RAM to flash cyclically and lazily; this copying is triggered by transactions that must commit, or an accumulation of more than half a page of redo records as described earlier.

To avoid writing page images that contain modifications from uncommitted transactions, we apply all the undo records of transactions whose records are still in RAM to the half-page image before we copy it to flash. These records are applied backwards in time, to ensure that the on-flash image is the one that existed just after the last committed transaction ended.

When the SSD boots after a crash, the checkpointing mechanism locates the end of the linked list of blocks reads backwards until it finds an image of each half-page of RAM. It then reads the pages in this list from oldest to newest, copying the half-page images to RAM and redoing the actions of committed transactions.