It's Not Where Your Data Is, It's How It Got There

Gala Yadgar, Roman Shor, Eitan Yaakobi, and Assaf Schuster
Computer Science Department, Technion
{*gala,yaakobi,assaf*}*@cs.technion.ac.il* *shroman3@mail.technion.ac.il*

Abstract

When hard disks (and tapes before them) dominated long term storage, sequentiality of placement and access was their main optimization goal. But in modern, flash based devices, which now dominate an increasing market share, data is updated 'out of place'. Their optimization must therefore focus on *data movement* rather than on static placement. An understanding of the processes that move data on a flash device is crucial for analyzing and managing it.

While sequentiality on hard drives is easy to visualize, as is done by various defragmentation tools, data movement on flash is inherently dynamic. With the lack of suitable visualization tools, researchers and developers must rely on aggregated statistics and histograms from which the actual movement is derived. The complexity of this task increases with the complexity of state-of-the-art FTL production and research optimizations.

Adding *visualization* to existing research and analysis tools will greatly improve our understanding of modern, complex flash-based systems. We developed SSDPlayer, a graphical tool for visualizing the various processes that cause data movement on SSDs. We use SSDPlayer to demonstrate how visualization can help us shed light on the complex phenomena that cause data movement and expose new opportunities for optimization.

1 Introduction

Data on flash devices moves to a different location whenever it is updated: the previous data location is marked as invalid, and the data is written again on a clean page. The *flash translation layer (FTL)* is responsible for mapping logical addresses to physical pages. The *garbage collection* process maintains a pool of clean blocks by occasionally erasing a block with mostly invalid pages and copying its valid pages to another available block. These internal writes, referred to as *write amplification*, are another cause for data movement throughout the device. The write amplification is usually estimated using a formula derived from an analysis of the greedy garbage collection [3].

Many FTL optimizations incur additional internal data movement. Examples include wear leveling [1], merging of log blocks [12], partition resizing [17], and parity updates [10]. Quantifying the write amplification is important for analyzing the effect of such optimizations on the performance and durability of the flash device. However, doing so is not always trivial and requires a deep understanding of the interacting causes of data movement within each device.

Currently available simulators [1, 11] output internal state and statistics in the form of lists, tables and histograms, from which deriving internal processes is cumbersome and requires a great deal of skill and imagination. Basic hardware evaluation boards[1] provide similar output, while advanced ones provide graph output of block level reliability tests [16]. SSD optimization tools provide fragmentation information[2], S.M.A.R.T statistics and block update frequency[3]. However, complicated flash processes cannot be derived from these aggregated statistics. Furthermore, these tools are intended for off-the-shelf SSDs, and cannot be used for research prototypes.

The increasing complexity of state-of-the-art flash management justifies the adoption of new research and analysis techniques. Just as graphs illustrate phenomena that are hard to identify in tables, and just as one picture is said to be worth a thousand words, we claim that *one clip is worth a thousand histograms*. To establish this claim, we developed *SSDPlayer*, an open source graphical tool for visualizing data layout and movement on flash devices. This tool will give us a better understanding of how our data gets from one place to another and why.

In the rest of this paper, we first introduce the basic features and structure of SSDPlayer. We then take a close look at several common data movement processes that were analyzed with standard mathematical methods. We use SSDPlayer to show how the analyzed phenomena can be easily identified by visualizing each of these processes and explain how visualization can shed light on similar processes in more complex systems. We will refer the reader to a few one-minute online clips generated with SSDPlayer for demonstration purposes[4].

2 SSDPlayer

The SSDPlayer display, depicted in Figure 1, is organized into chips, planes, blocks and pages, as specified by the user at startup. Colors and textures are used to represent

[1] http://www.openssd-project.org/
[2] http://www.auslogics.com/en/software/disk-defrag-pro/
[3] http://www.raxco.com/home/products/perfectdisk-pro
[4] http://www.cs.technion.ac.il/~gala/SSDPlayer/

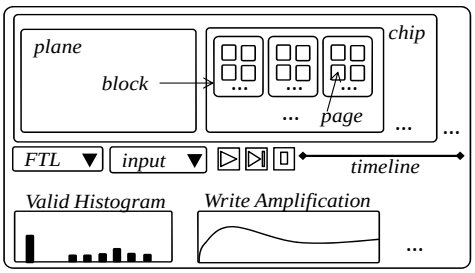

Figure 1: SSDPlayer display (simplified)

page and block properties, such as data 'temperature' or valid page count. A page's properties and state determine its fill color, texture, and frame color. A block's properties determine its background and frame colors. Note that the page and block properties need not necessarily match. Aggregated information such as write amplification is displayed in continuously updated histograms, illustrating how the device's state changes over time.

SSDPlayer is implemented in Java and is designed to provide the most general SSD functionality, in order to allow easy extensions and additions for a wide range of capabilities. The basic flash components – e.g., page, block, page mapping and garbage collection – are implemented as abstract classes that can be extended according to the desired FTL functionality. The simulation and visualization components are similarly flexible: the trace parser can be extended to process different trace formats. Alternatively, synthetic access distributions can be added by extending the workload generator. The basic histograms can be extended to display additional aggregated statistics.

SSDPlayer supports two modes of operation. In *simulation* mode, it simulates the chosen FTL on a raw I/O trace or on a synthetic workload, illustrating the SSD state at each step. This illustration is continuous, thus forming a "clip" of the data movements that take place during execution. This mode is useful for testing and analyzing various features without, or before, implementing them in a full scale simulator or hardware platform.

In *visualization* mode, SSDPlayer illustrates operations that were performed on an upstream simulator or device. The input in this mode is an output trace generated by a simulator, hardware evaluation platform, or a host level FTL, describing the basic operations that were performed on the flash device — writing a logical page to a physical location, changing block state, etc. This mode is useful for illustrating processes that occur in complex research and production systems, without porting their entire set of features into SSDPlayer.

SSDPlayer is intended for use as an open source project. Thanks to its flexible structure, a wide range of functionalities can be added to it in a straightforward manner. These include many recently suggested FTL optimizations, including wear leveling, page mapping, and garbage collection algorithms. Users can easily modify the page coloring scheme to visualize the concepts they are interested in and display continuously updated histograms required for their analysis. We describe several such scenarios in the following sections.

Our goal of keeping SSDPlayer as simple and easily extendible as possible lead to several design choices. Most of the complexity of full scale simulators is due to accurate performance modeling. Thus, we implemented SSDPlayer from scratch, focusing only on the way data moves, regardless of how much time it takes. However, it can be extended to provide performance analysis by adding delays during time consuming operations such as erasures and copies, or by collecting the relevant statistics and presenting them as a histogram or a final output file.

There is a tradeoff between the complexity and amount of details displayed, and how easily the visualized processes can be identified and interpreted. Thus, while there is no restriction on the complexity of the FTL schemes implemented within SSDPlayer, users should carefully choose which page and block attributes to display. For simplicity, we use a 'toy' device (2K pages) in our demonstrations. However, we used SSDPlayer to visualize devices with up to 25K pages on an HDTV screen by omitting fill texture and page numbers. Larger devices can be analyzed by visualizing a subset of the device's planes or chips, which is sufficient for a wide range of purposes.

3 Greedy Garbage Collection

The most commonly used formula for estimating the write amplification with greedy garbage collection as a function of page size and overprovisioning is that of Bux and Iliadis [3]. They derive the formula from a detailed analysis of the number of blocks with each *valid count* — number of valid pages. Their analysis shows that with a random uniform workload, the minimum value (*MinValid*) converges to a single value or to two consecutive values. To date, we are not aware of a similar derivation for purely non-uniform distributions such as Zipf. In this section, we use SSDPlayer to illustrate data movement in the uniform case, where it is well-understood. We then show how a visual illustration can shed some light on the non-uniform case, where data movement is complex and not fully understood.

The *Greedy* FTL in SSDPlayer implements greedy garbage collection within each plane, and a page allocation scheme that balances the number of valid pages between planes. All pages have the same color, but the page fill changes to a checkered pattern if it has been copied to a new block during garbage collection. Invalid pages are crossed out, but maintain their fill color and pattern until they are erased.

In the *Greedy-Uniform* demo, the basic manager is executed with a small SSD and a uniform random workload. This clip shows that shortly after the SSD's logical capacity is filled and garbage collection begins, *MinValid* sta-

bilizes at 10-11 pages. The portion of each block that is taken up by valid pages transferred at garbage collection is clearly visible thanks to their different pattern.

We use the same SSD and FTL with a Zipf workload. The *Greedy-Zipf* demo shows that *MinValid* converges much slower and at a higher value, of 15-16 pages. The reason is that cold pages that are rarely updated remain valid during consecutive garbage collection invocations. As a result, write amplification increases, leaving less space available in the erased blocks for invalid copies of hot pages, thus causing even more frequent garbage collection, and so on. This phenomenon is graphically visible as a dense grouping of *invalid* (*X*) marks on the plainly filled pages that represent user writes.

4 Hot/Cold Data Separation

Separating hot and cold data has been shown to reduce write amplification and, respectively, garbage collection costs and cell wear [4, 17]. Desnoyers [4] analyzes cases in which the hot and cold portions of the workloads are each accessed with different uniform distributions, showing that separating them to different partitions with greedy garbage collection results in the same write amplification as in the uniform case. Stoica and Ailamaki [17] analyze a workload with several *temperatures*. They show that several temperatures can be grouped into the same partition without increasing the write amplification, as long as the skew within each partition does not exceed a certain degree. The conclusions of both studies are based on a rigorous analysis of data movement processes. In this section, we use SSDPlayer to show how a graphical visualization can greatly clarify these processes and is certain to assist in analyzing more complicated scenarios.

The *HotCold* FTL separates pages into partitions according to their temperature. It is used with traces in which each input write request is tagged by a temperature tag. The user specifies the number of partitions, *P*, and the highest temperature of pages that belong to each partition. Each plane has *P* active blocks, on which pages of each partition are written. When an active block is full, a new clean block is allocated for this partition. Greedy garbage collection is used, determining partition sizes implicitly according to the number of writes with each temperature.

As a reference point, we first run the *HotCold* FTL with one partition and a Zipf workload where requests are tagged with ten different temperatures. The *HotCold-1* demo is essentially a replay of the demonstration in *Greedy-Zipf*. It shows how a simple addition of colors can facilitate our understanding of the process described in Section 3: before garbage collection starts, the red pages, which belong to the top five temperatures (and only 2% of the data), occupy roughly half of each block, representing their portion of accesses in the trace. As the garbage collection process advances, blue (cold) checkered (copied) pages occupy increasing portions of each block, most of them remaining valid until the next garbage collection on this block. Figure 2 shows snapshots of the first block in the device during this demo.

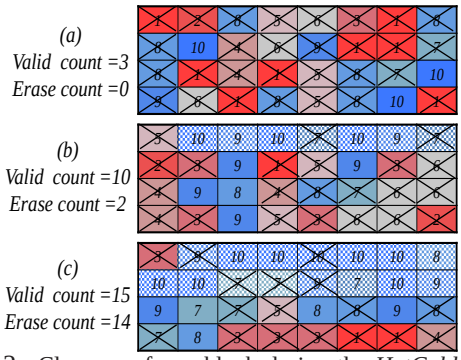

Figure 2: Close-up of one block during the *HotCold-1* demo with Zipf workload, tagged with 10 temperature ranges, where red (1) is the hottest and blue (10) is the coldest. The valid count is shown at the time when the block is chosen for the next erasure, where it is equal to *MinValid*. The *MinValid* pages that were copied to a clean block during previous garbage collections are filled with a checkered pattern. This demo shows their portion increasing until it stabilizes at roughly half the block size.

When we separate the data into two or three partitions, we observe a process similar to that in *HotCold-1*, because within each partition, pages are still accessed with a relatively high skew. However, this behavior changes when we define five partitions, one for every two temperatures. For this trace, this granularity is fine enough to reduce the skew in the cold partitions, so that garbage collection within each partition behaves as with a uniform workload. Indeed, in the *HotCold-5* demo, *MinValid* stabilizes at 10-11 pages like in *Greedy-Uniform*. This process, described in [4], is seen clearly in the demo. We believe much more complicated phenomena can be identified and analyzed as visualization becomes a standard research tool.

5 Reusable SSD

The recent *Reusable SSD* [19] reuses flash pages for additional (*second*) writes before they are erased. To perform a second write, the logical page written by the user is encoded with a special encoder that adds redundancy bits, producing an output that is twice the page size and can be written on a pair of physical pages that have already been programmed. The encoder guarantees that writing the new data will only require increasing the cell voltage level, thus complying with standard flash programming constraints.

The commonly used formula for write amplification cannot be used when additional writes are performed before the block is erased. The derivation in [3] does not extend trivially to this case, because the number of additional writes that can be performed depends on the way invalid pages or entire blocks are reused. In fact, since some redundancy must always be added to the logical data to enable second writes, the conventional definition of write

amplification does not accurately represent flash utilization in this context. Several models, with varying degrees of complexity, were suggested for analyzing the properties of second writes in various designs [13, 14, 18]. We use SSDPlayer to show how a graphical illustration can provide important insights for such complex designs.

The *Reusable* FTL implements second writes in SSDPlayer. Each block is first written normally by first writes. When it is chosen by the garbage collector it is either erased or *recycled* — allocated for second writes without erasure[5]. Upon receiving a write command, if a recycled block is available, a second write is performed on a pair of physical pages in the recycled block whose data has been invalidated.

Pages are colored according to the write level of their logical page. When a page is copied to a new block before erasure (such copies are always performed as first writes), it maintains the color of its *original* write level, but changes its texture to that of an internal write. Thus, the different colors represent the portion of the data written in first and second writes within both user and internal writes. In addition, we replaced the write amplification histogram with one showing *logical writes per erasure*. With N pages per block and first writes only, N logical writes per erasure are equivalent to a write amplification of 1. With second writes, $N\times1.5$ logical writes per erasure are the maximum value achievable when all pages are fully utilized for two writes, with no internal writes.

In the *Reusable* demo, we run the Reusable FTL on a small SSD with N=32 and a Zipf workload. It shows that most of the pages are utilized for two writes, but that many of the logical pages written as second writes (blue) are still valid when the block is erased and must be copied to a clean block (checkered). This means that pages written without prior erasure of the block end up occupying newly erased blocks when they are copied, reducing the benefit from second writes. Indeed, only 26 logical writes (out of $N\times1.5$=48 possible) are performed per erasure. Although this is more than the 17 writes per erasure achieved with first writes only[6], flash utilization can clearly improve. This understanding motivated the use of second writes in Reusable SSD for hot pages only.

The *HotColdReusable* FTL uses second writes only for hot data, which it identifies by the temperature tag in the trace. We run this FTL in the *HotCold-Reusable* demo, with the Zipf workload from the HotCold demos, where requests are tagged with ten different temperatures. Second writes are used for the top 5 temperatures. The demo shows that pages written in second writes are almost always invalid by the time their block is erased. As a result, the logical writes per erasure increase to 32, representing a significant benefit from second writes. As a reference, recall that the best partitioning of this trace according to temperature (in the HotCold-5 demo) resulted in *MinValid*=10, corresponding to 22 writes per erasure. The two versions of Reusable SSD demonstrate the power of visualization as a research tool for new techniques and system designs. Similar visual experiments can provide valuable insight for formalizing their utilization, and for designing optimal garbage collection schemes for such systems.

The full Reusable SSD design is much more complex. It performs second writes in parallel to blocks in different planes, identifies cold data without external tagging, and handles encoding failures and mapping constraints [19]. The implications of Reusable SSD for device lifetime and performance have been thoroughly evaluated by a detailed implementation in DiskSim [1]. We take advantage of this implementation to illustrate the full Reusable SSD design in SSDPlayer. We added a logging mechanism to the implementation in DiskSim, which logs all physical write commands, garbage collection procedures, and state changes to a trace file. In the online *ParallelReusable-*$*$ demos we use this trace file as input to SSDPlayer in visualization mode to visualize the complex data movement in the full Reusable SSD design with Zipf and real traces.

6 Other Data Movement Processes

We discuss here several popular flash optimization domains that we plan to make available in future versions of SSDPlayer. Data movement plays a major role in all of them, occurring within complex interacting processes. We describe how visualizing these processes will help to understand them and to optimize the systems in which they occur.

RAID. The effect of various redundancy schemes such as RAID5 and erasure coding on SSD performance and wear is a hot research topic [2, 5, 10]. The performance of these schemes is greatly affected by the data movements they incur. Parity updates are a major contributor to write amplification and accelerated wear, especially in update schemes that were originally designed for hard drives [2]. The location of parity blocks as well as the availability of previous, invalid data and parity blocks, greatly affect the durability of the system and its recovery costs.

We are currently extending SSDPlayer to include notions of parity and stripes, so that the distribution of parity and data throughout the device will be easily visible and stripes can be discerned. Stripes can be extended to vary in size or to include invalid data or parity pages. Parity can be extended to represent numerous erasure coding techniques [9]. These optimizations, already a subject for ongoing research, complicate data movement to the point where visualization is crucial for understanding it.

Caching. SSDs that are used as a caching tier employ an additional management layer, further increasing the complexity of data movement processes. Data may

[5]The detailed conditions for block recycling appear in [19].

[6]This value is derived from *MinValid*=15 in the GreedyZipf demo.

move as a result of varying the overprovisioned space or read and write cache sizes [15], or the movement of pages within the garbage collection process may depend on dynamic properties such as the logical queue they belong to or their dirty status. The complex interactions between these processes, easily illustrated within a tool like SSDPlayer, will be much better understood through visualization, where page colors and textures can represent popularity, dirty status, or prefetch hints, and block backgrounds and frames can be used to represent logical partitions and queues, and to distinguish between read and write caches.

Wear. Many FTL optimizations that target wear leveling incur additional data movements, such as migration of cold data into old blocks. At the page level, various optimizations distinguish between the LSB or MSB pages in MLC flash for garbage collection, page allocation and mapping, thus modifying the way data is moved within existing processes [7, 8]. Many such optimizations are triggered by the observed page or block bit error rates, which are in essence dynamic properties. A graphical illustration can help point out unexpected interactions between these highly correlated processes. To that end, SSDPlayer or similar tools can depict the physical layout of LSB and MSB pages on blocks, and use colors, backgrounds and frames to represent page or block level bit error rate, erase count, and so forth.

Content. Although all of the examples in this work referred to the page metadata or physical properties, visualization can also help analyze content based optimizations, such as compression or deduplication [6]. For example, colors and patterns can represent the compression ratio or number of duplicates of a page, to show how these affect or incur data movement on flash. Snapshots, versions and clones can be represented in a similar manner, to visualize the interaction between file systems or databases and their underlying storage.

7 Conclusions

The ever-increasing complexity of flash based systems and their management makes it more and more difficult to analyze related new methods and optimizations. We showed that a graphical illustration of data movement processes on flash can facilitate a much deeper understanding of their causes and effects. It can also expose unexplored phenomena and opportunities for optimization. We thus believe that visualization should be a standard mechanism in the tool box of every flash oriented research or development team. Furthermore, visualization can provide similar benefits in the analysis and optimization of any logical layout that incurs extensive data movement, such as shingled magnetic recording, log structured file systems, etc.

We supported our claims with SSDPlayer: a flexible, extendible tool for visualizing the various processes that cause data movement on SSDs. We continue to work on additional features and optimizations that will expand the scope of the player and improve user experience. The code and executable files of SSDPlayer are available online[7]. We encourage researchers and developers to use this tool for their analysis and to contribute to the online repository.

Acknowledgments

We thank Niva Bar-Shimon and Kai Li for their valuable suggestions for improving SSDPlayer and its appearance. We thank the anonymous reviewers and our shepherd, Daniel Ellard, for helping improve this paper.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX (ATC)*, 2008.

[2] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi. Differential RAID: Rethinking RAID for SSD reliability. *Trans. Storage*, 6(2):4:1–4:22, July 2010.

[3] W. Bux and I. Iliadis. Performance of greedy garbage collection in flash-based solid-state drives. *Perform. Eval.*, 67(11):1172–1186, Nov. 2010.

[4] P. Desnoyers. Analytic models of SSD write performance. *Trans. Storage*, 10(2):8:1–8:25, Mar. 2014.

[5] K. Greenan, D. D. E. Long, E. L. Miller, T. Schwarz, and A. Wildani. Building flexible, fault-tolerant flash-based storage systems. In *HotDep*, 2009.

[6] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *FAST*, 2011.

[7] X. Jimenez, D. Novo, and P. Ienne. Wear unleveling: Improving NAND flash lifetime by balancing page endurance. In *FAST*, 2014.

[8] X. Jimenez, D. Novo, and P. Ienne. Libra: Software controlled cell bit-density to balance wear in NAND flash. *ACM Trans. Embed. Comput. Syst. (TECS)*, 14(2), 2015.

[9] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *FAST*, 2012.

[10] J. Kim, J. Lee, J. Choi, D. Lee, and S. H. Noh. Improving SSD reliability with RAID via elastic striping and anywhere parity. In *DSN*, 2013.

[11] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar. FlashSim: A simulator for NAND flash-based solid-state drives. In *SIMUL*, 2009.

[12] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: Locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.*, 42(6):36–42, Oct. 2008.

[13] X. Luojie, B. M. Kurkoski, and E. Yaakobi. WOM codes reduce write amplification in NAND flash memory. In *GLOBECOM*, 2012.

[14] S. Odeh and Y. Cassuto. NAND flash architectures reducing write amplification through multi-write codes. In *MSST*, 2014.

[15] Y. Oh, J. Choi, D. Lee, and S. H. Noh. Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems. In *FAST*, 2012.

[16] Siglead Inc. *SigNAS-II: Siglead NAND Analyzer System*, 2.2 edition, September 2012.

[17] R. Stoica and A. Ailamaki. Improving flash write performance by using update frequency. *Proc. VLDB Endow.*, 6(9):733–744, July 2013.

[18] E. Yaakobi, A. Yucovich, G. Maor, and G. Yadgar. When do WOM codes improve the erasure factor in flash memories? In *ISIT*, 2015.

[19] G. Yadgar, E. Yaakobi, and A. Schuster. Write once, get 50% free: Saving SSD erase costs using WOM codes. In *FAST*, 2015.

[7] http://www.cs.technion.ac.il/~gala/SSDPlayer/