Experience from Two Years of Visualizing Flash with SSDPlayer

Gala Yadgar and Roman Shor
Computer Science Department, Technion
Email: {gala,shroman}@cs.technion.ac.il

Abstract—**Data visualization is a thriving field of computer science, with widespread impact on diverse scientific disciplines, from medicine and meteorology to visual data mining. Advances in large scale storage systems, as well as low level storage technology, played a significant role in accelerating the applicability and adoption of modern visualization techniques. Ironically, "the cobbler's children have no shoes": researchers who wish to analyze storage systems and devices are usually limited to a variety of static histograms and basic displays.**

The dynamic nature of *data movement* on flash has motivated the introduction of SSDPlayer, a graphical tool for visualizing the various processes that cause data movement on SSDs. In 2015, we used the initial version of SSDPlayer to demonstrate how visualization can assist researchers and developers in their understanding of modern, complex flash-based systems. While we continued to use SSDPlayer for analysis purposes, we found it extremely useful for education and presentation purposes as well. In this paper, we describe our experience from two years of using, sharing, and extending SSDPlayer, and how similar techniques can further advance storage systems research and education.

I. INTRODUCTION

Data visualization refers to applying meaningful geometric or visual encoding to otherwise "non-visual" data, and is the focus of numerous academic studies and commercial products [1]–[6]. Early visualization techniques, such as statistical maps, scatter plots, and histograms, still form the basis of fundamental research and presentation tools. However, technological advances and new techniques allow researchers in many disciplines to use increasingly powerful tools to visualize large scale and complex data. Special attention is given to the ability to dynamically control the way data is displayed, by interactive zooming, filtering, distortion and aggregation.

Ironically, the analysis of the systems and architectures that facilitate these advances is still limited to basic visualization in the form of many types of graphs and histograms. In this context, analyzing the state of the data within storage systems and devices is especially difficult: the illustration of data sequentiality on hard drives using various defragmentation tools is just about the only well-known example. Unfortunately, this static representation does not capture some of the major phenomena in modern storage systems.

Storage systems are designed to dynamically adjust to changing workload characteristics and system conditions. Thus, data may migrate between storage nodes for load balancing, or between storage hierarchies according to its popularity. Background deduplication may eliminate logical copies of data, while changing availability requirements may trigger the creation of replicas or other forms of redundancy. Visualization is a natural technique for understanding the effects and implications of these processes. In this paper, we focus on flash based storage as one example of the complexity of modern storage architectures.

Data on flash devices moves to a different location whenever it is updated: the data is written again on a clean page, and the previous data location is marked as invalid. 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 after copying its valid pages to another available block. These internal writes are another cause for data movement throughout the device.

Many FTL optimizations incur additional internal data movement. Examples include wear leveling [7], merging of log blocks [8], partition resizing [9], and parity updates [10]. Quantifying these additional writes 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 [7], [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 [12] provide similar output, while advanced ones provide graph output of block level reliability tests [13]. SSD optimization tools provide fragmentation information [14], S.M.A.R.T statistics and block update frequency [15]. However, complicated flash processes cannot be understood 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 motivated us to introduce data visualization principles to storage systems research and analysis. We developed *SSDPlayer*, an open source graphical tool for visualizing data layout and movement on flash devices, and presented its initial version in the 7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '15) [16]. The feedback we received from the workshop attendees, as well as from our colleagues, inspired us to extend the interactive features in SSDPlayer and the complexity of the devices it can display. It also encouraged our use of SSDPlayer for educational
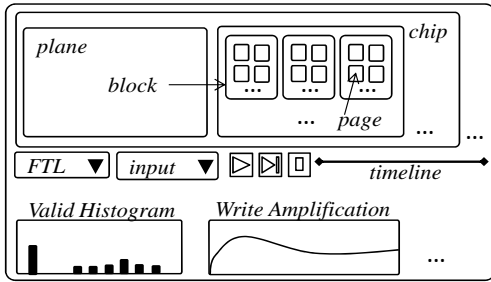
Fig. 1. SSDPlayer display (simplified)

purposes in undergraduate courses and projects, and for the presentation of new ideas and designs in academic conferences and industrial collaboration meetings. In this paper, we describe our experience from two years of visualization, along with our insights into how it can advance storage system research.

The rest of this paper is organized as follows. We introduce the basic features and structure of SSDPlayer in Section II. We then take a close look at several use cases that demonstrate the different aspects of visualization in storage system analysis. Section III describes our experience from using SSDPlayer for educational purposes. In Section IV, we describe our experience from presenting complex ideas and designs. Section V describes some of the feedback we received from colleagues and conference participants and how this feedback was incorporated into the new version of SSDPlayer. We describe some of our research experience with SSDPlayer in Section VI, and provide additional notes to users and developers in Section VII. We discuss related work on visualization in Section VIII, and conclude in Section IX. Throughout this paper, we refer the reader to one-minute clips that were generated with SSDPlayer for demonstration purposes and are available on the SSDPlayer website[1].

II. SSDPlayer

SSDPlayer is 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 graphical parameters to visualize the concepts they are interested in and display the details and statistics relevant to their analysis. We describe several such scenarios and additions in the following sections.

We implemented SSDPlayer in Java in order to maximize portability and minimize platform-specific dependencies. It 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

[1]http://ssdplayer.cswp.cs.technion.ac.il/

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.

Our goal of keeping SSDPlayer as simple and easily extendible as possible led to several design choices. Most of the complexity of full scale simulators is due to accurate performance modeling that takes into account numerous device-specific parameters. Thus, we implemented SSDPlayer from scratch, focusing on write-only workloads, and 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.

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. We demonstrate the benefit of this mode in Section IV.

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 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.

There is a tradeoff between the complexity and number 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 must carefully choose the size of the visualized device and which page and block attributes to display.

For demonstration purposes, we normally use a 'toy' device of 2K pages. A device of up to 12K pages can be viewed in full detail on a regular HDTV screen. SSDPlayer handles larger devices of more than 250K pages by adjusting the level of detail presented. A subset of the device's planes or chips

can be viewed in full detail while the simulation continues to update the state of the entire device. Alternatively, the entire device can be viewed by omitting fill texture and by aggregating the presentation of an entire block's pages into one smaller rectangle. SSDPlayer allows to dynamically zoom-in and zoom-out between several levels of detail and different aggregation criteria. We describe this option in greater detail in Section V.

III. AN EDUCATION USE CASE: WRITE AMPLIFICATION

We originally used SSDPlayer to illustrate data movement with uniform workloads, where it is well-understood, and to show how a visual illustration can shed some light on the non-uniform case, where data movement is complex and not fully understood. In the process of generating the respective demos, we identified their potential for illustrating even the basic concepts of garbage collection and write amplification for students who are encountering them for the first time. In this section, we describe these concepts and our experience in using SSDPlayer to illustrate them in the context of under-graduate courses and projects.

SSD basic design concepts. Updates in SSDs are performed out-of-place: the previous data location is marked as invalid, and the data is written again on a clean page. To accommodate these writes, some physical storage capacity is not included in the device's exported logical capacity. Thus, the device's *overprovisioning* is defined as $\frac{T-U}{U}$, where T and U represent the number of physical and logical blocks, respectively [17]. The FTL is responsible for mapping logical addresses to physical pages.

The garbage collection process is invoked whenever the number of clean blocks drops below a certain threshold. Garbage collection is typically performed *greedily*, picking the block with the minimum *valid count* (the lowest number of valid pages) as the victim for *cleaning*. The valid pages are *moved*—read and copied to another available block, and then the block is erased. These additional internal writes, referred to as *write amplification*, delay the cleaning process, and require, eventually, additional erasures.

The most accurate formula for estimating the write ampli-fication with greedy garbage collection as a function of page size and overprovisioning is that of Bux and Iliadis [18]. They derive the formula from a detailed analysis of the number of blocks with each valid count, and show that with a random uniform workload, the minimum value (*MinValid*) converges to a single value or to two consecutive values. Desnoyers [17] performs a similar analysis which results in a formula which is less accurate but can be calculated more easily.

Illustration with SSDPlayer. 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.
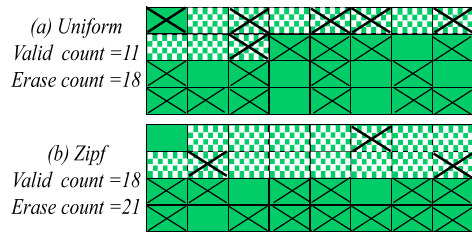

Fig. 2. Close-up of one block at the end of the *Greedy-Uniform* (a) and the *Greedy-Zipf* (b) demos. The pages that were copied to a clean block during previous garbage collections are filled with a checkered pattern. With the Zipf workload, their number is higher, which also results in a higher valid count and more erasures.

In the *Greedy-Uniform* demo, this basic FTL 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* stabilizes 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 more slowly 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. Figure 2 illustrates this for one block.

The *Greedy-Sequential* demo is designed to show the best-case scenario of garbage collection and write amplification. With a completely sequential workload, the demo shows that *MinValid* is always zero and the write amplification is always one, and that garbage collection does not generate any internal writes. This is a trivial result in terms of workload and performance analysis, but is not completely obvious to students who encounter these concepts for the first time.

We now use these demos regularly in an undergraduate course on storage systems, where their contribution is twofold. First, explaining data movement and its effects while referring to an "animated" example that can be paused or rewinded is much easier for an instructor than appealing to the students' imagination or constructing a complex series of individual slides. Second, as students assimilate the new concepts much faster, they often raise issues that are beyond basic design principles, such as wear leveling and possible optimizations of greedy garbage collection. Thus, visualization helps us use our limited teaching hours more effectively, and to generate discussion and interest in advanced related topics.

Hands-on experience with SSDPlayer. For students who are interested in storage system design, we leverage the flexible design of SSDPlayer to serve as a basis for undergrad-uate projects. In these projects, students typically read some

background on SSD design challenges [7], [10], and then delve into the details of implementing additional FTLs or features within SSDPlayer. These projects have been very successful, and their products have been merged into the new version of the tool.

For example, one pair of students added the possibility to define breakpoint rules within the player, stopping the simulation whenever a predetermined condition holds. In the course of this project, they had to identify interesting events that could be triggers for analysis or debugging purposes, such as the first invocation of garbage collection in a certain plane or chip, or the first time the valid count or write amplification reach a certain value. Other projects' goals, such as displaying an info screen when the simulation is paused, or dynamically switching between different zoom levels, required students to identify meaningful aggregation metrics and possible inconsistencies between different FTLs. The RAID functionality, discussed in Section VI, was also implemented as part of an undergraduate project. This project required a deep understanding of the different RAID levels, the challenges in parity updates, and how they differ in SSDs and in hard drives.

SSDPlayer has become an integral part of our educational tool box, where visualization is the fundamental contributor to its success. Entry-level students benefit from our ability to clearly illustrate basic design concepts. Students who choose to specialize in the subject benefit from a simplified framework that can be easily extended. More importantly, SSDPlayer *shows* them what is going on inside the device. This helps them understand the consequences of their design choices, and use their time effectively. We believe that visualization can help in a similar manner when teaching subjects such as cache replacement, paging, dynamic memory allocation, and large cluster management, where data continuously moves from one place to another.

IV. A PRESENTATION USE CASE: REPROGRAMMING

We originally used SSDPlayer to demonstrate the advantage of visualization in the analysis of data movement in complex FTL designs, such as *Reusable SSD* [19], which reuses flash pages for additional (*second*) writes before they are erased. The demo videos we generated turned out to be a valuable tool in presenting this and subsequent research results, both to an academic audience in conferences and to practitioners within industry collaborations. In this section, we explain the basic challenges in reusing flash pages, and show how we used SSDPlayer to visualize our approach.

Flash page reuse. Flash pages are composed of floating-gate cells, whose voltage levels represent different bit values. Single-level flash cells (SLC) can store a single bit value, 1 (initially) or 0. Multi-level flash cells (MLC) support four voltage levels, mapped to four two-bit values: 11 (in the initial state), 01, 00 or 10. In MLC flash, the MSB (*high*) and the LSB (*low*) bits represented by the cell are each mapped to a different flash page. Thus, MLC flash blocks are composed of high and low pages, respectively. Flash is a write-once

medium—after its cells are *programmed* to increase their voltage level, they must be erased prior to writing again. This constraint motivates the use of out-of-place updates in SSDs, which incur additional internal writes and erasures.

Reusable SSD reduces the number of erasures by performing additional writes on a block before it is 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. This condition is sufficient to allow reuse of SLC flash pages. We thus refer to this scenario as "ideal" page reuse.

Additional limitations apply to the reuse of MLC flash pages, as a result of specific optimizations applied during MLC page programming. Page reuse is still possible, but cannot utilize all the block's pages for two writes [20]. One possible pattern for page reuse is the low-low-high (LLH) reprogramming scheme [21], in which blocks are programmed in two rounds. In the first round, only the low pages are programmed as first writes. The second round takes place after most of these pages have been invalidated, and consists of programming the unused (high) pages for the first time, and reprogramming the invalidated low pages as second writes.

The commonly used formula for write amplification cannot be used when additional writes are performed before the block is erased. The derivation in [18] and [17] 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 [22]–[24]. We use SSDPlayer to show how a graphical illustration can provide important insights into such complex designs.

Presentation with SSDPlayer. The *Reusable* FTL implements ideal second writes in SSDPlayer. Each block is first written normally by first writes. When it is chosen as victim for garbage collection, it is either erased or *recycled* — allocated for second writes without erasure[2]. 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

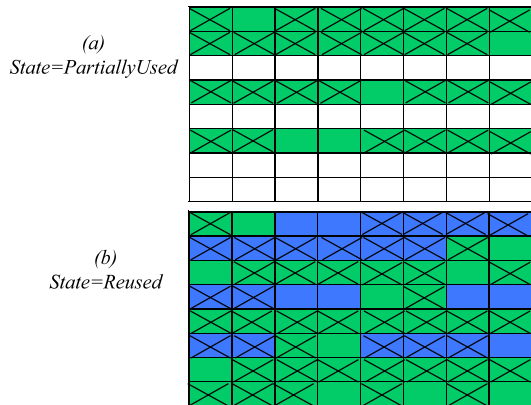[2]The detailed conditions for block recycling are specified by the Reusable SSD design [19].

Fig. 3. Reuse process of one block in the *LLH-FTL-MSR* demo. When the block is allocated for future reuse, its state is changed to PartiallyUsed (a) and only its low pages are programmed (green). When most of these pages are invalid, the block is allocated for reuse. Its state changes to Reused (b), its invalid low pages are reprogrammed (blue), and its high pages are programmed for the first time. The position of low and high pages in this block represents their layout in the OpenSSD hardware.

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, $N \times 1.5$ 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 $N \times 1.5$=48 possible) are performed per erasure. Although this is more than the 17 writes per erasure achieved with first writes only[3], flash utilization can clearly improve. We used this demo in a graduate course on coding theory, to illustrate the design challenges of performing additional writes and to motivate a theoretical model for analyzing and optimizing garbage collection in this context.

Visualization of Reusable SSD. 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 [7].

We took advantage of this implementation to illustrate the full Reusable SSD design in SSDPlayer. We added a

---

[3]This value is derived from *MinValid*=15 in the Greedy-Zipf demo.

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-Zipf* and *ParallelReusable-MSR* 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 workloads [25], [26], respectively. We used this demo for presenting Reusable SSD at conferences, where it was especially useful for illustrating how all our design choices were combined within a complete FTL implementation.

Visualization of LLH-FTL. *LLH-FTL (Low-Low-High Reprogramming FTL)* is a full FTL design that emerged from our detailed research on MLC flash page reuse [20]. To accommodate second writes, LLH-FTL reserves some of its blocks in a *partially-used* state where only their low pages are used. A partially-used block can be reused, in which case the FTL will reprogram all or some of the low pages and all the high pages. The number of partially-used blocks is controlled by a set of conditions that balance reuse potential and the availability of overprovisioned space. To dynamically adjust their number, the FTL can forego recycling of a partially-used block, and instead program the high pages and leave the low pages untouched until the block is erased.

Our research consisted of a full implementation of LLH-FTL on the OpenSSD Jasmine board [12] for evaluation. We also used an adaptation of this FTL implementation as an emulator for evaluating the effect of additional parameters that could not be modified on the hardware platform. We added a logging mechanism to this emulator that produced a similar output as the log of DiskSim described above, and we used this output to generate the *LLH-FTL-MSR* demo. The lifecycle of reused blocks is clearly illustrated: the high pages remain white while they are partially used. The low, used (green) pages are then reused and turn blue, while the clean (white) pages are used for the first time. Figure 3 zooms in on one block during this process.

We used this demo to illustrate LLH-FTL at the conference where it was first presented. In subsequent, longer talks, we played both this demo and that of Reusable SSD, to emphasize the difference between ideal and practical page reuse in real systems. Thus, our presentation consisted of a visualization of the same workload (the prn_0 volume from the MSR Cambridge collection [25], [26]) handled by two different FTL designs implemented on two different platforms. This visualization complemented our theoretical analysis and evaluation results, by illustrating the applicability as well as the limitations of our research results.

SSDPlayer has proved a powerful tool for presenting complex ideas and designs to expert audiences in advanced courses, academic conferences, and collaborations with the industry. In this context, too, explaining the details of a complex design is much easier when a visualization of its full implementation is playing in the background. This allows us to use our presentation time effectively and engage our audience, who, in turn, can easily follow the details of our
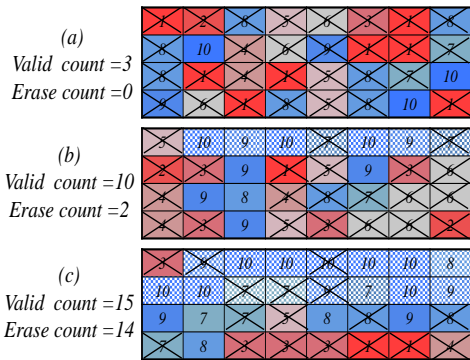
Fig. 4. Close-up of one block during the *HotCold-1* demo with a Zipf workload, tagged with 10 temperature ranges, where red (1) is the hottest and blue (10) is the coldest. The valid count is shown at the time when the block is chosen for the next erasure, where it is equal to *MinValid*. The *MinValid* pages that were copied to a clean block during previous garbage collections (checkered pattern) are from the coldest temperature ranges. This demo shows their portion increasing until it stabilizes at roughly half the block size.
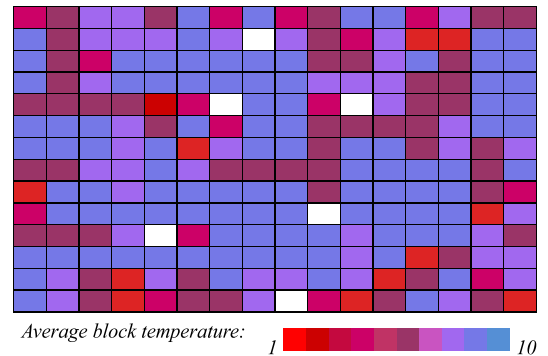

Fig. 5. Zoom-out view of a 32K-page device in the *Large-HotCold-5* demo. Aggregate information is displayed by coloring each block according to the average temperature of its pages.

design. These benefits of visualization can also be gained by developers and distributers in their interaction with existing and potential customers.

V. A FEEDBACK USE CASE: HANDLING LARGE DEVICES

We originally used colors to represent page access frequency in SSDPlayer to show how simple visual aids can help clarify not only how data moves, but also why it moves. When presenting our demos, we received valuable feedback and advice on how to extend this concept to additional attributes, realistic device sizes, and additional architectures and domains. In this section, we demonstrate the benefit from using colors in analyzing workloads and SSD performance, and how the feedback from the community helped us improve SSDPlayer in this context.

Hot and cold data separation. Separating hot and cold data has been shown to reduce write amplification and, respectively, garbage collection costs and cell wear [9], [27]. Desnoyers [27] analyzes cases in which the hot and cold portions of the workload 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 [9] 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.

The *HotCold* FTL implemented in SSDPlayer 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[4]. The user specifies the number of partitions, P, and the highest temperature of pages that belong to each

[4]SSDPlayer does not currently implement online temperature classification. This functionality can be added by extending the HotCold FTL.

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.

Visualization with SSDPlayer. 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 the *Greedy-Zipf* demo (Figure 4 shows snapshots of the first block in the device during this demo). It shows how a simple addition of colors can facilitate our understanding of the process described in Section III: 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.

When we separate the data into two or three partitions, we observe a process similar to that in the *HotCold-1* demo, 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 the *Greedy-Uniform* demo. This process, described by Desnoyers [27], is seen clearly in the demo.

Visualizing large scale devices. The design of SSDPlayer had to address the tradeoff between the level of detail presented and the size (measured in number of pages) of the device that can be clearly visualized. In the initial version of SSDPlayer, users could turn off the display of logical page numbers and per-block counters, which allowed them to view devices of up to 20K pages with reasonable clarity. However, this option had to be specified at startup, and consisted mainly of minimizing the pages without modifying their displayed

attributes.

A valuable piece of advice we received when presenting SSDPlayer was to handle large devices by "zooming out", aggregating the information per block instead of just reducing page size. Thus, instead of blocks being presented as a collection of pages, they can be presented as solid objects, whose color represents the aggregate value of one of the pages' attributes. In the current SSDPlayer version, users can pause the simulation, adjust the level of detail presented on the screen, and specify the attributes they wish to view. Some of these attributes are common to all FTLs, such as the valid count or block *age*—the number of times it has been erased. Others are available only for specific FTLs, such as the average temperature of pages in the HotCold FTL, or average write level in the Reusable FTL. This addition makes it possible to view devices with over 250K pages, and observe phenomena that could not be easily discerned in smaller devices or partial visualization of large devices.

The online *Large-HotCold-5* demo shows how zooming out helps analyze a device with 32K pages. The input is a Zipf workload with the same parameters used in the *HotCold-5* demo (α=1), where pages from every two temperatures are stored in a separate partition. The first zoom level shows entire pages, as in the previous demos, illustrating the different speeds in which pages of different temperatures are invalidated. The second zoom level omits the page numbers and block counters from the display, providing a detailed view of a larger portion of the device. In the last two zoom levels, which differ in the size of the blocks presented, pages are omitted altogether and blocks are colored according to an aggregate metric.

In this demo, blocks are colored according to the average temperature of their pages so that the simulation continues to show how the allocation of blocks to partitions converges. Figure 5 shows a snapshot of the SSDPlayer display at this zoom level. At the end of the demo, we switch the color scheme to represent block age, in order to show the case for wear leveling—blocks that are allocated to the hot partition are erased repeatedly, while blocks storing cold pages are rarely erased, and stay "young".

We note that this phenomenon was not as obvious with the same distribution on a small device. When the number of logical pages is small, the "long tail" of the Zipf distribution is not long enough—the cold pages are accessed less frequently than the hot pages, but frequently enough to generate some data movement in the cold partitions. Thus, the uneven wear is less pronounced. Uneven wear can still be demonstrated with a small device, but doing so requires a more extreme access distribution as input.

The intuitive nature of visualization is a key factor in receiving valuable feedback from the community. Our experience is that when researchers are first presented with SSDPlayer, ideas on extending or applying it to their own area of interest immediately come to mind. Notable suggestions we have received include applying our visualization technique to large pools of RAM [28], content defined storage [29],

heterogenous storage hierarchies of RAM and flash, cache organization [30], [31], log structured file systems, shingled magnetic recording [32], and the interaction between file systems or databases and their underlying storage. Some of these extensions are part of our future work. We are even aware of an ongoing project, inspired by SSDPlayer, of visualizing satisfiability of clauses and derived conditions during long executions of SAT solvers [33]. We believe that complicated phenomena can be identified and analyzed in many domains within computer science and specifically in systems research, as visualization becomes a standard research tool.

VI. A RESEARCH USE CASE: RAID PARITY OVERHEAD

The increase in SSD capacity, with the shift from SLC to MLC and TLC flash, comes at the cost of reduced reliability. An increasingly common approach to compensate for the reduced reliability is to organize data in RAID stripes, either within an array of SSDs [34]–[36] or within the chips of a single SSD [10], [37]. However, the frequent parity updates required in these architectures increase the write amplification and device wear. Thus, understanding the data movement processes caused by these additional updates is crucial for evaluating the overall contribution of RAID to SSD reliability. In this section, we describe the basic challenges of RAID in SSDs, and how they are visualized within SSDPlayer. We then describe our insights from visualizing several common scenarios.

RAID in SSDs. In traditional RAID architectures, designed for an array of hard drives, interleaved parities help minimize the overheads of reading old parity values and writing the updated values. However, flash based architectures must also consider the *parity update overhead*—the additional flash writes caused by parity updates. The parity overhead is defined as $\frac{P}{P+D}$, where P and D are the number of parity and data pages written, respectively [10]. The parity update overhead depends on the size of the write requests (larger requests require less parity updates), and on the amount of parity pages that are copied to new blocks during garbage collection. Commercial RAID architectures that are applied to arrays of SSDs minimize the parity update overhead by only writing entire stripes to flash [35], [36].

RAID can also be employed within a single SSD, where data is striped across separate chips and protected by one or more parity pages in each stripe. In these architectures, the RAID functionality is embedded into the FTL, which is responsible for updating the parity whenever data is written. Several optimizations have been suggested for minimizing the parity update overhead in these architectures. Examples include write buffering of parity pages [38], adapting stripe size to device age [39], and "elastically" mapping data and parity to stripes of flexible size [10]. The storage and update overhead of parity pages motivated these optimizations. We are further interested in how parity updates affect the efficiency of the garbage collection process, and how their overhead is affected by it.
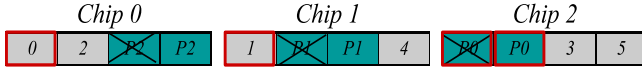
Fig. 6. Toy example of RAID-5-FTL and a device of three chips and four pages per block. Logical pages 0-5 (gray) were written in consecutive requests, each generating a parity (turquoise) update. Stripe 0 is highlighted by a red page frame, showing the logical pages that belong to it, and both the valid and invalid copies of the parity page.

Visualizing RAID in SSDPlayer. RAID is implemented within SSDPlayer in three different FTLs, corresponding to the most commonly used RAID architectures. *RAID-1-FTL* implements mirroring: chips are divided into pairs and data in one chip in the pair is replicated on the other. *RAID-5-FTL* implements one parity for every stripe. The parity position is interleaved, so that different chips store the parity of different stripes. *RAID-6-FTL* implements two parity pages per stripe. When a logical page is written, the RAID FTLs are responsible for updating the parity (or parities) of the stripe this page belongs to. If a write request includes several pages in the same stripe, the parity of this stripe is updated only once. Parity and data pages have different colors, which makes it easy to distinguish between them when they are first written, and when they are copied during garbage collection. Extra reads of old parity pages are not visualized by SSDPlayer, which is designed to visualize data movement. Similarly, the encoding scheme which determines the content of the parity pages is orthogonal to this analysis and is not implemented.

The parity update overhead is displayed in a continuous histogram next to the one showing the write amplification, making it easy to compare the two measures. Another important feature is *stripe highlighting*: when the simulation is paused, users may specify a stripe they wish to follow closely. Once the simulation is resumed, all the data and parity pages belonging to this stripe will be highlighted with a colored page frame. Users may specify whether they want to highlight only the valid pages in the stripe, or to include invalid copies of the data and/or parity pages as well. Several stripes can be highlighted simultaneously with different frame colors. Figure 6 shows an example of a highlighted stripe.

Parity update overhead and garbage collection. The *RAID-5-Parity* demo shows a device with eight chips and a total of 10K pages configured as RAID-5. We run a Zipf workload in which the size of all write requests is one page. We expect every write to generate a parity update, resulting in a parity update overhead of $\frac{1}{2}$. We first note the uneven distribution of parity updates due to the high update frequency of pages in the first stripe. This phenomenon was discussed in the context of elastic striping [10]. Highlighting this stripe shows that it is not only responsible for a significant portion of the updates, but its invalid pages also occupy a significant part of the device's overprovisioned space.

As the simulation progresses, the parity update overhead drops from $\frac{1}{2}$ to $\frac{1}{3}$. The reason for this drop is that the data pages are inherently colder than the parity pages that protect

them. Thus, they are more likely to be valid and copied during garbage collection. As a result, parity pages are responsible for a smaller portion of internal writes than data pages, and their update overhead decreases as the write amplification increases. The space occupied by the data and parity of the hottest stripes decreases with each garbage collection invocation. We also see, at the end of the simulation, that the space occupied by valid and invalid parity pages consists of roughly 25% of the device's capacity. This is twice the storage overhead expected in a RAID-5 architecture with eight nodes, which is 12.5%.

This demo illustrates a phenomenon similar to the one shown in the *HotCold-1* demo, where the cold pages are repeatedly copied during garbage collection, generating excessive internal writes. Our extension of the RAID-5 FTL was a natural next step following the identification of the same process in two different architectures. The *RAID-5+* FTL minimizes the effect of parity updates on write amplification by writing data and parity pages in two separate partitions. This can be viewed as a special case of separating hot and cold data, where the FTL is aware of the "hotness" of the parity pages. In the *RAID-5-SeparateParity* demo, we run the same trace on the same device with the RAID-5+ FTL. This demo shows that the space occupied by parity pages converges to 14%, which is only slightly higher than the expected 12.5%. As a result, the parity update overhead remains almost $\frac{1}{2}$, but the write amplification is lower (1.75 instead of 1.9).

The RAID-5+ FTL is not intended to be a full FTL design. The benefit from separating data and parity pages depends on the size of requests, the skew in the data itself, and on additional optimizations such as write buffering. Nevertheless, our experience of using it within SSDPlayer, on a variety of device sizes and workload distributions, resulted in valuable insight into the interaction between parity update overhead and write amplification. This insight was a significant step forward in our research, and is yet another example of how visualization can contribute to our understanding of complex processes within storage systems and the interactions between them.

VII. NOTES FOR USERS AND DEVELOPERS

SSDPlayer supports three levels of user involvement. The first consists of passively viewing the online demos, which cover a range of representative phenomena of data movement on flash devices. The second level is that of the *power user*, which makes use of basic as well as advanced features included in the SSDPlayer distribution. The third level is development, in which users add new features or FTLs according to their own use cases.

SSDPlayer power users. The SSDPlayer downloads page[5] provides access to the latest version as an executable Java application, which is distributed with the traces we used for generating the online demos, and a sample configuration file. The SSDPlayer Users' Guide [40] provides detailed information on the input type and format, configuration parameters, available FTLs, and additional features.

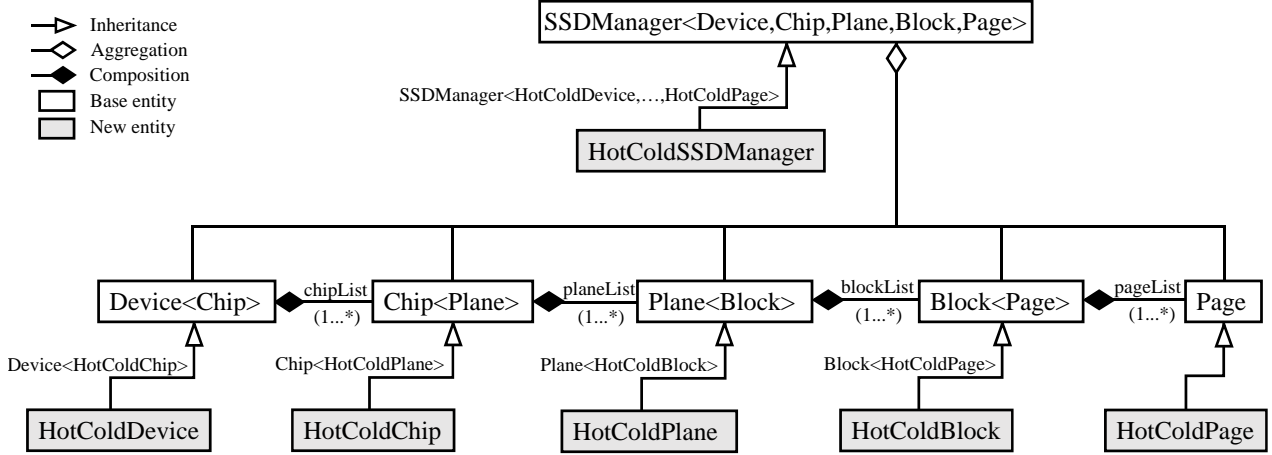[5]http://ssdplayer.cswp.cs.technion.ac.il/downloads/

Fig. 7. Partial class diagram, depicting the relationship between the generic abstract base entities (no fill) and those added for the HotCold FTL (solid fill).

The SSDPlayer distribution allows users to explore a wide range of use cases without writing a single line of code. By editing the configuration file, users may configure the physical layout of their device (i.e., number of pages per block), adjust its visual settings (i.e., color of hot and cold pages), and provide FTL-specific parameters (i.e., number of partitions).

After starting SSDPlayer, users first choose which FTL to run. They then have the option of specifying an existing I/O trace as input, or generating synthetic input with one of the existing workload generators. During the simulation, users can adjust the zoom level dynamically, specify breakpoints that conditionally pause the simulation, and view and save the detailed device state at any point in the simulation.

SSDPlayer developers. The SSDPlayer Programmer's Guide [41] provides a detailed description of the core classes in SSDPlayer's code base. This open-source project consists of approximately 14K lines of code in approximately 200 files. We distinguish between two scales of programming tasks. Adding a new FTL or feature requires adding or modifying several class-files, and is considered a large-scale task. However, extending existing features typically consists of modifying one or two files, and is considered a small-scale programming task. We demonstrate the different scales below by outlining our addition of the *HotCold* FTL. We refer the interested reader to the Programmer's Guide for a complete documentation of the relevant classes and methods.

Extending base entities. The *HotCold* FTL was one of the first FTLs added to SSDPlayer, and thus it directly extends the abstract base classes, as depicted in Figure 7. *HotColdPage* extends page base entity. It includes a temperature property and sets the background color accordingly by overriding the appropriate method. *HotColdBlock* extends the block entity, and includes a *HotColdPartition* property which defines the partition the block belongs to. We had to override the methods for getting the block status and frame color to display according to its partition.

Garbage collection in all of SSDPlayer's current FTLs is performed within a single plane. The *HotCold* FTL employs greedy garbage collection within each partition, which is implemented within the *HotColdPlane*. The methods in this class are responsible for allocating an active block within each partition, and for maintaining the separation of data according to temperature. This is done by ensuring that valid pages from the victim block are moved to an active block in the same partition. There is no specific functionality that had to be implemented within the chip entity. However, since the base entities are abstract, we had to create a *HotColdChip* which aggregates the appropriate planes. *HotColdDevice* extends the device class by adding several FTL-specific properties for collecting statistics for display.

FTLs are implemented as *managers*, and *HotColdSSD-Manager* extends the basic manager class. The manager is responsible for loading the FTL-specific parameters from the configuration file and creating the device according to those parameters. *HotColdSSDManager* also provides a special trace parser, *HotColdTraceParser*, which extends the basic parser by handling input lines with temperature tags.

Extending existing features. In addition to the basic FTL components described above, we opted to extend two existing features as part of the *HotCold* FTL. We first add *HotColdWriteAmplificationGetter*, which implements the basic statistics interface. It computes the write amplification within each partition by accumulating page writes and moves at the device level. We defined a histogram where the write amplification for each partition is plotted in a different color. We then added *HotColdWriteAmplificationGetter* to *HotColdSSD-Manager*'s list of statistics for display. Next, we extended the breakpoint base class and defined a new breakpoint type. The *HotColdWriteAmplification* breakpoint allows users to specify a write amplification value, W, and a partition, p, so that the simulation pauses when the write amplification in p reaches W.

The entire *HotCold* FTL implementation consists of approximately 900 lines of code, of which 150 handle the additional statistics and breakpoint types. Clearly, the programming effort required to add a new FTL or feature depends on its com-

plexity. However, straightforward additions to SSDPlayer's functionality are usually limited to several well defined parts of its code base.

VIII. RELATED WORK

Scientific visualization has been defined as "the transformation of complex, multidimensional data into informative graphical displays to see the unseen by leveraging what is known through visual methods." [6]. Traditional scientific visualization tools include graphical representations of numerical data, such as the scatterplot, the histogram, the boxplot, and the contour map. Currently, scientific visualization is considered a field within computer graphics. Research in this field addresses challenges such as efficient use of advanced hardware, human-computer interaction, scalable platforms, abstraction models, or protocol standardization [4], [42]. Some notable examples of current tools include map animation for earth system research [5], medical visualization applications of augmented reality [1], and visualization of three-dimensional nucleic acid structures [43]. To the best of our knowledge, SSDPlayer is the first tool designed for visualizing data movement processes in general, and specifically in SSDs.

Information visualization, or *data visualization*, "extends traditional scientific visualization of physical phenomena to diverse types of information (e.g., text, video, sound, or photos) from large heterogenous data sources" [44]. It focuses on representation of "non-visual data" by attaching meaningful geometric or visual encoding [45]. Michael Friendly [2] surveys the history of data visualization, drawing a line from early geometric diagrams and maps of the 14th century to large-scale statistical and graphics software engineering of the 21st century, through notable examples from the 19th century: Dr John Snow's dot map that helped identify the waterborne cause of cholera during its outbreak in London (1855), Florence Nightingale's polar area charts (or 'rose diagrams'), which motivated the improvement of sanitary conditions in battlefield treatment (1857), and The Statistical Atlas of the Ninth US Census (1874).

Current research in information visualization addresses the representation of very large data sets, such as network graphs, connections between text documents, and real-time streaming data, focusing on dynamic and interactive visualization [2], [44], [45]. The interactive aspect is considered crucial for *visual data exploration*, or *visual data mining*, and includes interactive linking, filtering, zooming, and projection [3]. SSDPlayer facilitates the pursuit of insight into data movement processes via visualization, in a dynamic and interactive manner, which is the goal of modern-era information visualization.

An important related challenge is to adapt the visual display of digital content to the needs of users with various types of visual impairment. Low vision and color vision deficiencies ("color blindness") make it difficult for users to access online learning material [46], websites [47]–[49] and scientific literature [50]. Initiatives such as the Web Accessibility Initiative (WAI) [51] offer resources and guidelines for developers, while others provide products such as magnifiers and voice readers [52]. Most of SSDPlayer's features can be made accessible by choosing the page and font sizes, as well as color scheme that best suits each user's needs.

Many surveys evaluated the effectiveness of visualization in computer science and mathematical education [53]–[55]. A graphical representation can help explain the basic concepts in these fields, which are inherently abstract. Indeed, these surveys indicate that visualization makes teaching more enjoyable, improves student motivation, participation, and learning, and provides a basis for classroom discussion and interaction with colleagues. At the same time, several obstacles hinder the wide adoption of visualization. These mainly consist of the overheads of identifying effective software, learning new tools, searching or generating good examples, and adapting them to the course content [54].

SSDPlayer presents several advantages in this context. It is distributed with a set of FTLs that represent the major approaches in SSD design, which can be easily used with the built-in workload generator or sample traces. The online demos provide a set of initial examples for both instructors and students. Finally, it facilitates varying degrees of "active" learning: none at all, when viewing online demos, moderate, when choosing simulation parameters and analyzing the illustrated outcome, or high, when implementing new features.

IX. CONCLUSIONS

The ever-increasing complexity of modern storage systems and their management makes it more and more difficult to analyze underlying processes as well as related new methods and optimizations. However, while the scope and functionality of data visualization techniques advance, storage system analysis continues to rely on traditional basic visualization tools. Our experience with SSDPlayer demonstrates how visualization can contribute to our understanding of data movement processes on flash. Our experience also indicates that similar benefits can be obtained by applying data visualization principles to almost any other storage system component as well as to entire systems as a whole.

In addition to the obvious benefit for storage system analysis and research, our experience revealed additional valuable benefit of visualizing storage devices. We were able to improve the quality of our teaching of basic and advanced concepts by playing short demos in the classroom, and by defining extensions to the tool as undergraduate project tasks. We also improved our presentation of research results that consist of complex ideas and FTL designs by demonstrating them within SSDPlayer at conferences and meetings with the industry. Finally, SSDPlayer was the trigger for valuable discussions with colleagues at these events, where we received feedback and ideas from the community on how to improve the tool and to apply it to additional systems and research domains. Our experience confirms that the well-established benefits of data visualization can and should be adopted to storage system research and design.

REFERENCES

[1] R. T. Azuma, "A survey of augmented reality," *Presence: Teleoperators and Virtual Environments*, vol. 6, no. 4, pp. 355–385, 1997.

[2] M. Friendly, *A Brief History of Data Visualization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 15–56.

[3] D. A. Keim, "Information visualization and visual data mining," *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 1, pp. 1–8, Jan 2002.

[4] R. S. Laramee, H. Carr, M. Chen, H. Hauser, L. Linsen, K. Mueller, V. Natarajan, H. Obermaier, R. Peikert, and E. Zhang, *Future Challenges and Unsolved Problems in Multi-field Visualization*. London: Springer London, 2014, pp. 205–211.

[5] D. DiBiase, A. M. MacEachren, J. B. Krygier, and C. Reeves, "Animation and the role of map design in scientific visualization," *Cartography and Geographic Information Systems*, vol. 19, no. 4, pp. 201–214, 1992.

[6] D. A. Griffith, *Spatial Autocorrelation and Spatial Filtering: Gaining Understanding Through Theory and Scientific Visualization*. Springer Science & Business Media, 2013.

[7] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *USENIX Annual Technical Conference (ATC)*, 2008.

[8] 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.*, vol. 42, no. 6, pp. 36–42, Oct. 2008.

[9] R. Stoica and A. Ailamaki, "Improving flash write performance by using update frequency," *Proc. VLDB Endow.*, vol. 6, no. 9, pp. 733–744, Jul. 2013.

[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 *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks DSN*, 2013.

[11] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "FlashSim: A simulator for NAND flash-based solid-state drives," in *1st International Conference on Advances in System Simulation SIMUL*, 2009.

[12] http://www.openssd-project.org/.

[13] *SigNAS-II: Siglead NAND Analyzer System*, 2nd ed., Siglead Inc., September 2012.

[14] http://www.auslogics.com/en/software/disk-defrag-pro/.

[15] http://www.raxco.com/home/products/perfectdisk-pro/.

[16] G. Yadgar, R. Shor, E. Yaakobi, and A. Schuster, "It's not where your data is, it's how it got there," in *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2015.

[17] P. Desnoyers, "What systems researchers need to know about NAND flash," in *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[18] W. Bux and I. Iliadis, "Performance of greedy garbage collection in flash-based solid-state drives," *Perform. Eval.*, vol. 67, no. 11, pp. 1172–1186, Nov. 2010.

[19] G. Yadgar, E. Yaakobi, and A. Schuster, "Write once, get 50% free: Saving SSD erase costs using WOM codes," in *13th USENIX Conference on File and Storage Technologies FAST*, 2015.

[20] F. Margaglia, G. Yadgar, E. Yaakobi, Y. Li, A. Schuster, and A. Brinkmann, "The devil is in the details: Implementing flash page reuse with WOM codes," in *14th Usenix Conference on File and Storage Technologies (FAST)*, 2016.

[21] F. Margaglia and A. Brinkmann, "Improving MLC flash performance and endurance with extended P/E cycles," in *IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

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

[23] S. Odeh and Y. Cassuto, "NAND flash architectures reducing write amplification through multi-write codes," in *30th International Conference on Massive Storage Systems and Technology (MSST)*, 2014.

[24] E. Yaakobi, A. Yucovich, G. Maor, and G. Yadgar, "When do WOM codes improve the erasure factor in flash memories?" in *IEEE International Symposium on Information Theory ISIT*, 2015.

[25] "SNIA IOTTA," http://iotta.snia.org/traces/388, SNIA, 2014, retrieved: 2014.

[26] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *Trans. Storage*, vol. 4, no. 3, pp. 10:1–10:23, Nov. 2008.

[27] P. Desnoyers, "Analytic models of SSD write performance," *Trans. Storage*, vol. 10, no. 2, pp. 8:1–8:25, Mar. 2014.

[28] P. Reinecke, G. Barnett, P. Goldsack, and B. Monahan, "GAS: Guess, abstract, and speculate," Hewlett Packard Labs, Technical Report HPE-2017-05, January 2017.

[29] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing NAND flash-based SSDs," in *9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[30] G. Yadgar, M. Factor, and A. Schuster, "Cooperative caching with return on investment," in *29th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2013.

[31] G. Yadgar, M. Factor, K. Li, and A. Schuster, "Management of Multilevel, Multiclient Cache Hierarchies with Application Hints," *ACM TOCS*, vol. 29, pp. 5:1–5:51, 2011.

[32] A. Aghayev and P. Desnoyers, "Skylight—a window on shingled disk operation," in *13th USENIX Conference on File and Storage Technologies FAST*, 2015.

[33] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *38th Annual Design Automation Conference (DAC)*, 2001.

[34] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi, "Differential RAID: Rethinking RAID for SSD reliability," *Trans. Storage*, vol. 6, no. 2, pp. 4:1–4:22, Jul. 2010.

[35] J. Colgrove, J. D. Davis, J. Hayes, E. L. Miller, C. Sandvig, R. Sears, A. Tamches, N. Vachharajani, and F. Wang, "Purity: Building fast, highly-available enterprise flash storage from commodity components," in *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

[36] "Introduction to the EMC XtremIO storage array (ver. 4.0)," EMC, White Paper H11752.7, April 2015.

[37] K. Greenan, D. D. E. Long, E. L. Miller, T. Schwarz, and A. Wildani, "Building flexible, fault-tolerant flash-based storage systems," in *5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[38] S. Im and D. Shin, "Flash-aware RAID techniques for dependable and high-performance flash memory SSD," *IEEE Transactions on Computers*, vol. 60, no. 1, pp. 80–92, Jan 2011.

[39] S. Lee, B. Lee, K. Koh, and H. Bahn, "A lifespan-aware reliability scheme for RAID-based flash storage," in *ACM Symposium on Applied Computing (SAC)*, 2011.

[40] G. Yadgar, R. Shor, E. Yaakobi, and A. Schuster, "SSDPlayer visualization platform version 1.2.1 users guide," May 2017.

[41] R. Shor, G. Yadgar, O. Mauda, D. Hadar, and R. Matza, "SSDPlayer visualization platform programmers guide for version 1.2.1," May 2017.

[42] C. Johnson, "Top scientific visualization research problems," *IEEE Computer Graphics and Applications*, vol. 24, no. 4, pp. 13–17, July 2004.

[43] X. Lu and W. K. Olson, "3DNA: a software package for the analysis, rebuilding and visualization of three-dimensional nucleic acid structures," *Nucleic Acids Research*, vol. 31, no. 17, p. 5108, 2003.

[44] J. A. Wise, J. J. Thomas, K. Pennock, D. Lantrip, M. Pottier, A. Schur, and V. Crow, "Visualizing the non-visual: Spatial analysis and interaction with information from text documents," in *IEEE Symposium on Information Visualization (INFOVIS)*, 1995.

[45] C. Chen, "Information visualization," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 4, pp. 387–403, 2010.

[46] K. L. Crow, "Four types of disabilities: Their impact on online learning," *TechTrends*, vol. 52, no. 1, pp. 51–55, 2008.

[47] J. Carter and M. Markel, "Web accessibility for people with disabilities: an introduction for Web developers," *IEEE Transactions on Professional Communication*, vol. 44, no. 4, pp. 225–233, Dec 2001.

[48] H. Takagi, C. Asakawa, K. Fukuda, and J. Maeda, "Accessibility designer: Visualizing usability for the blind," *SIGACCESS Access. Comput.*, no. 77-78, pp. 177–184, Sep. 2003.

[49] L. Jefferson and R. Harvey, "Accommodating color blind computer users," in *8th International ACM SIGACCESS Conference on Computers and Accessibility (Assets)*, 2006, pp. 40–47.

[50] B. Wong, "Points of view: Color blindness," *Nature Methods*, vol. 8, no. 6, p. 441, May 2011.

[51] https://www.w3.org/WAI/.

[52] http://www.abledata.com/.

[53] N. Presmeg, *Handbook of Research on the Psychology of Mathematics Education: Past, Present and Future*. Sense Publishers, 2006, ch. Research on visualization in learning and teaching mathematics, pp. 205–235.

[54] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. A. Velázquez-Iturbide, "Exploring the role of visualization and engagement in computer science education," in *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR)*, 2002.

[55] T. Naps, S. Cooper, B. Koldehofe, C. Leska, G. Rößling, W. Dann, A. Korhonen, L. Malmi, J. Rantakokko, R. J. Ross, J. Anderson, R. Fleischer, M. Kuittinen, and M. McNally, "Evaluating the educational impact of visualization," *SIGCSE Bull.*, vol. 35, no. 4, pp. 124–136, Jun. 2003.