A Performance Model and File System Space Allocation Scheme for SSDs

Choulseung Hyun
School of Computer Science
University of Seoul
cshyun@uos.ac.kr

Yongseok Oh
School of Computer Science
University of Seoul
ysoh@uos.ac.kr

Eunsam Kim
School of Information & Computer Engineering
Hongik University
eskim@hongik.ac.kr

Jongmoo Choi
School of Computer Science & Engineering
Dankook University
choijm@dku.edu

Donghee Lee
School of Computer Science
University of Seoul
dhl_express@uos.ac.kr

Sam H. Noh
School of Information & Computer Engineering
Hongik University
samhnoh@hongik.ac.kr

***Abstract*—Solid State Drives (SSDs) are now becoming a part of main stream computers. Even though disk scheduling algorithms and file systems of today have been optimized to exploit the characteristics of hard drives, relatively little attention has been paid to model and exploit the characteristics of SSDs. In this paper, we consider the use of SSDs from the file system standpoint. To do so, we derive a performance model for the SSDs. Based on this model, we devise a file system space allocation scheme, which we call *Greedy-Space*, for block or hybrid mapping SSDs. From the Postmark benchmark results, we observe substantial performance improvements when employing the Greedy-Space scheme in ext3 and Reiser file systems running on three SSDs available in the market.**

I. Introduction

The recently introduced Solid State Drives (or Disks) (SSDs) are slowly, but surely catching the interest of consumers. They are starting to replace hard drives in laptop computers and are serious contenders in server computers as well due to its many favorable characteristics inherited from Flash memory that are the building blocks of SSDs [1]–[3]. With no mechanical parts, unlike the hard drive, SSDs are light, shock-resistant, noiseless, low-power consuming, and they show quite different performance characteristics [4], [5]. Though interest regarding SSDs has started to rise, they have mostly been directed to the internals of the SSD [6], [7]. Our interest rests on how the file system can make better use of SSDs. Even though disk scheduling algorithms and file systems of today have been optimized to exploit the characteristics of hard drives, relatively little attention has been paid to model and exploit the characteristics of SSDs from the file system standpoint.

In this paper, we derive a simple performance model for block or hybrid mapping SSDs that, in essence, is identical to that of the hard drive. Using this model, we devise a file system space allocation scheme for SSDs called Greedy-Space. Through real implementations on the ext3 and Reiser file systems and using three SSD products available in the market, we show that substantial performance improvements can be achieved for a wide range of workloads. By employing the Greedy-Space space allocation scheme, for most of the workloads, the execution time is substantially reduced with the reduction rate varying depending on the workload and environment.

The rest of the paper is organized as follows. In Section II, we describe the basic characteristics of Flash memory storage and SSDs as well as other work related to this topic. In Section III, we derive write cost models for the hard drive and the SSD. Based on the implications of the model, we devise the new Greedy-Space allocation scheme for SSDs in Section IV. In Section V we present the experimental environment as well as the results obtained from the experiments. Then, we conclude with a summary and directions for future work in Section VI.

II. SSD Background and Related Work

A NAND Flash memory chip that is the storage medium of an SSD has multiple blocks and each block has a set of pages. Data once written to a page cannot be modified without erasing the block containing the page. To accommodate the asymmetric unit sizes and times for read, write, and erase operations at the Flash memory chip level and to provide a readable/writable sector disk interface, Flash memory storages employ a complex software module called the Flash Translation Layer (FTL) [8], [9].

To achieve high performance and large capacity, an SSD has numerous Flash memory chips, SRAM, and SDRAM that are connected through buses [6]. SSDs employ FTLs to control and schedule all operations of these chips and buses. In most SSDs, multiple Flash memory chips comprise a single logical Flash memory chip. All same numbered blocks in all the chips form a clustered block, and all same numbered pages in these blocks form a clustered page. As data can be read/written from/to physical pages of a clustered page in parallel, performance of an SSD exceeds that of a single Flash memory chip.

978-1-4244-7153-9/10/$26.00 ©2010 IEEE

Aside from controlling the chips and buses, another important task of the FTL is to map sectors to Flash memory storage. The two basic forms of mapping in SSDs are page mapping [8] and block mapping [10]. In block mapping, a physical block in Flash memory holds a fixed number of sectors (N_s). To read a sector, the FTL calculates the logical block number by dividing the sector number by N_s. Then it looks up the map to convert the logical block number to a physical block number. If sectors are placed within the block in an ordered manner, finding the sector is straightforward.

Writing a sector is more complicated. To modify a sector within a data block b, the FTL writes the new sector data to a clean page of an over-provisioned block, hereafter called a *log* block. Then, subsequent writes for sectors of data block b are directed to a page of the log block. Later, the FTL consolidates the original data block and the log block through a *merge* operation, which erases another empty log block and copies all valid sectors to it from the data block and the log block. After copying the sectors, the empty log block now becomes the new data block and the old data and log blocks become empty log blocks. Note also that there are situations, such as when the log block is written to sequentially, where the merge operation may require only a small number of copies.

Contrary to block mapping FTL, page mapping FTL relocates each modified sector (or multiple sectors in a page) separately to any available page in an over-provisioned block. Its operation is similar to the log-structured approach [11] in that write requests are appended to an empty over-provisioned block. Like LFS (Log-structured File System) [11], it needs to recycle used blocks to make empty blocks for further write requests. This recycling mechanism is similar to *segment cleaning* in LFS except that blocks are reclaimed instead of segments [12]. Numerous hybrid mapping schemes that combine the advantages of block mapping and page mapping have also been proposed [13]–[16].

There are other works related to SSD design and characterization. Chen et al. analyze the performance characteristics of state-of-the-art SSDs through wide range of experiments providing insight to system designers [4]. Agrawal et al. gives a taxonomy of the many design choices that are available to SSD designers and, through simulations, analyses how these choices would affect performance [6]. Kim et al. propose a methodology for extracting essential parameters from SSDs [17]. In a Linux based study, Kim et al. considers disk schedulers for block or hybrid mapping SSDs [18].

A point to make here is that as each SSD manufacturer develops their own proprietary FTL and does not provide much information regarding their design, one cannot definitely conclude what type of scheme is being used for each SSD. However, if one has a solid understanding of the internal workings of the FTL, one may be able to fairly safely guess the general design. Based on our understanding and experience, the SSDs that we have experimented with in this paper employ either block or hybrid mapping schemes.

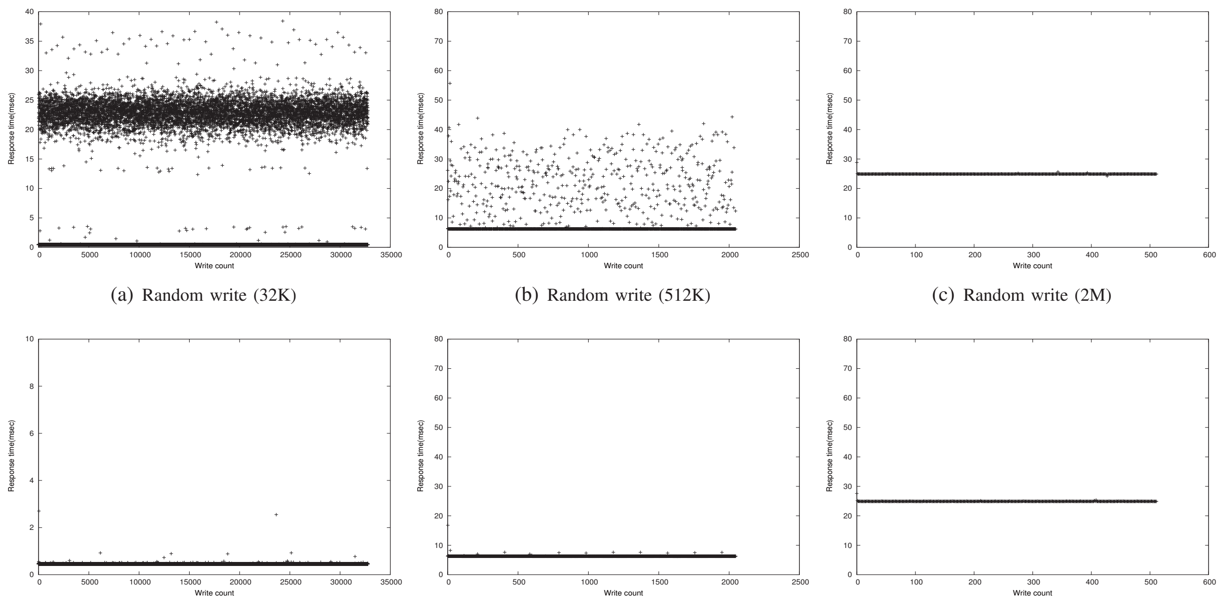
III. Write Cost Models

In this section, we derive a write cost model for file systems when the underlying storage is an SSD that employ the block mapping technique. (As read cost is constant for SSDs, we do not consider the read cost model.) To derive a cost model for the SSD, we start off from the simple performance model of what Wang et al. refer to as the Overwrite approach [19].

A simple performance model for writing a sector in a hard drive is given as $T_{1sect} = T_{pos} + \frac{S}{B}$ where T_{pos} is the sum of the average seek time and the average rotational latency, B is the write bandwidth of the disk, and S is the sector size in bytes [19]. In reality, however, writes to disks are requested in groups through a `sync` operation from the file system. Assuming n write requests are requested together, we can generalize T_{1sect} in two ways. First is the worst case scenario where every write request goes to a different cylinder from the previous one and the performance cost model will be nT_{1sect}. The best case scenario is writing all n sectors to the same cylinder and the write cost can be modeled as $T_{nsect} = T_{pos} + n\frac{S}{B}$ where $1 \leq n \leq C$, where C is the number of sectors in a cylinder. (If n is larger than C, the request can be regarded as two independent sub-requests without compromising the model.)

Now, we derive the write cost model for an SSD. Observe the write performance shown in Figure 1. This figure shows the response time for each write request of the corresponding data size when requesting a total of 1GB to one of the SSDs that we describe later. These values were those obtained by devising an application to synchronously write raw data directly to the SSD. The y-axis in the figures is the response time in milliseconds, while the x-axis is the request sequence. Observe from all the graphs of Figure 1 that a relatively thick band forms along the x-axis meaning that the majority of requests are serviced in that time frame. Above the band, there are numerous "spikes" (shown as dots), that is, response times that are out-of-band. In the lower figures for the sequential writes, we see that these spikes are few. However, in the upper figures when writes are random, we see a much higher number of spikes of a wider range. This observation is quite similar to one that would be observed in a hard drive. With random writes, positioning delay will be more variant than for sequential writes.

It is difficult to derive the exact reasons for each of these spikes as there are many factors that influence the design of an SSD [6]. However, based on our understanding of Flash memory and FTL software described in Section II, we know that write operations incur merge operations to make available free blocks. Hence, we can conjecture with high confidence that the key factor that induces such spikes is closely related to the merge operation. Depending on the merge operation, different responses can occur. Thus, with much simplification, we consider the spikes to be equivalent to the merge cost; the shorter spikes being those incurred by merges with small number of copy operations, while the larger spikes being for those of merges with numerous copy operations.



(a) Random write (32K) (b) Random write (512K) (c) Random write (2M)

(d) Sequential write (32K) (e) Sequential write (512K) (f) Sequential write (2M)

Fig. 1. Appearances of spikes mainly due to merge operations for particular request sizes

The key observation here is that there is an analogy between the merge times (T_{merge}) of an SSD and the positioning time T_{pos} of a hard drive. Based on this, we can argue that the worst case write performance model to write a single sector in an SSD is $T_{1sect} = T_{merge} + \frac{S}{B}$ where B is the write bandwidth of an SSD and S is the sector size in bytes.

A similar argument can be made for the best case with SSDs which happens when the n sectors are written with an overhead of a single merge operation. Hence, the write cost to write all n sectors with an overhead of a single merge can be modeled as $T_{nsect} = T_{merge} + n\frac{S}{B}$ where $1 \leq n \leq L$, where L is the number of sectors in a "cylinder" of an SSD.

Now the question that arises here is that of L. What is L? In a hard drive, this was simply the cylinder group size, which could be easily visualized as a hardware concept. In an SSD, there is no concept of a cylinder but there is something similar to it (as we show later), and we will refer to this cylinder counterpart in the hard drive as the *logical block* in SSD. And for now, let us just say that for every SSD this L exists. We will later show how this L can be obtained.

The write cost model for SSDs that we just derived gives us insights for optimizing file systems and disk scheduling algorithms for SSDs. The model implies write behavior of an SSD is similar to that of a hard drive, thus implying that sequential writes are preferable to random writes in both drives. However, this does not imply that hard drives and SSDs are the same. In fact, there is a key difference between the two drives. That is, reads are more or less constant for SSDs, while this is not true for hard drives. As a result, given a logical block, increasing n is more amenable in SSDs as read cost, which we can safely assume to be constant irrelevant to location, is no longer a factor to be considered. Hence, blocks (that is, multiple sectors from the file system viewpoint) may be placed anywhere instead of at particular cylinders as is done for hard drives.

Another thing to note is that of logical blocks. Though we described a logical block of an SSD to be analogous to a cylinder of a hard drive, a logical block does not posses a concrete notion of a physical "cylinder" as in hard drives. Hence, the size of a logical block is a unit that may be freely determined by the file system. The question is, then, how to choose this logical block size.

For this let us return to Figure 1. As discussed previously, Figure 1 shows spikes in response time due to merge operations within the SSD. However, note that for Figures 1(c) and (f), which is when the write request size is 2MB, large intermittent spikes are not observed. This is true for both the sequential and random requests. This tells us that, ideally, there is an "optimal" size that maximizes the utility of the resources that the SSD has, possibly leading to maximized performance. We refer to a block of this size to be the logical block and discuss the empirical aspect of a logical block in the next section.

IV. Space Allocation for SSD

This section describes the space allocation scheme that we develop. In order for this scheme to work, we first need to make concrete the notion of a logical block [17], [18].

A. *The logical block*

Ideally, a logical block of an SSD is the write unit where write cost is optimized. That is to say, if every write could be made in logical block units, then the minimum number of merge operations for writing the given blocks are incurred as evident in Figure 1(c) and (f).

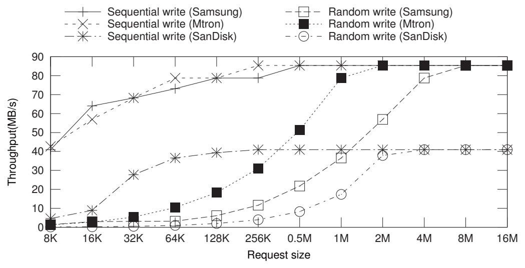
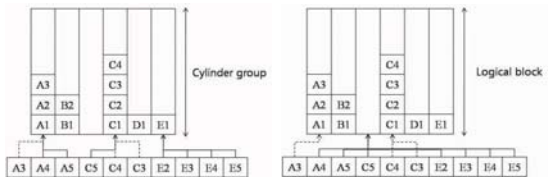


Fig. 2. Finding the logical block size



(a) Traditional scheme (b) Proposed scheme

Fig. 3. Comparison of space allocation schemes

Many design decisions affect the performance of SSDs [6], [7]. To optimize performance FTL designers take great efforts to design FTL software to make full use of all available hardware resources and features such as the planes, chips, buses, and buffers through interleaving and parallelism. Finding the logical block size through product specifications is virtually impossible due to the complicated interactions between hardware and software, and more so as this information is proprietary and undisclosed.

The logical block size, though, can be obtained through a simple set of experiments as follows. First, we open the device file that maps with the SSD with the O_DIRECT option to avoid the disk cache effect. Then, starting from a small write block size that is a power of 2 (and smaller than the Flash memory block size), say 8KB, we do the following. First, sequentially write 1GB into the device file obtaining its throughput. Then, we do the same thing again, only this time writes are done randomly making sure there is no overlap in the requests so that all 1GB is written to. (1GB is an ad hoc but fairly safe size in current implementations of SSDs such that all log blocks managed by the FTL are safely consumed.) The throughputs for the sequential and random writes are compared. If the values are close enough, then 8KB is the logical block size, and we are done. (In our case, we consider the two numbers to be close enough if the whole numbers of the reported numbers are the same.) Otherwise, we double the write size and start the process over. This is done until we find a write size where the sequential and random write throughputs meet the terminating condition.

Figure 2 shows results after going through the process just described for the three SSDs in Table I. Note how the random write throughput converges towards the sequential write throughput eventually becoming the same at some point. These results represent typical characteristics of SSDs that employ block mapping FTL. Sequential write performance is much better than random write. However, at a specific request size, merge overhead minimizes resulting in random and sequential write performance becoming almost the same.

B. Design of the Greedy-Space allocation scheme

Now that the notion of a logical block has been clarified, we can now view the file system space as a collective sequence of logical blocks. Given this viewpoint, we propose a new space allocation scheme for file systems that employ SSDs based on the performance model presented in the previous section. Note from the write cost model of an SSD that, to reduce the write cost, the number of sector writes per merge, n, should be maximized as the other parameters are all fixed values.

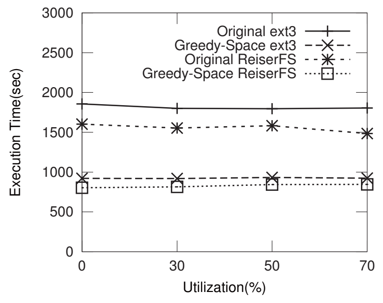
The key idea of the scheme that we propose is summarized in Figure 3. There are five files A, B, C, D, and E with their blocks being represented with subscripts. We then have new write requests $A_3, A_4, A_5, C_5, C_4, C_3, E_2, E_3, E_4, E_5$ arriving at `sync`. Of these, A_3, C_4, C_3 are to existing blocks. Then, in traditional file systems such as ext3, a new block would be allocated to the cylinder group where the metadata and the rest of the its file resides (Figure 3(a)). In contrast, the key idea of our scheme is to allocate all the new blocks to the logical block that has the most free blocks as depicted in Figure 3(b). That is, blocks $A_4, A_5, C_5, E_2, E_3, E_4, E_5$ are all allocated to the same logical block so as to maximize n. This does not incur any penalty for reads as reads in SSDs are near constant, unlike hard drives.

Let us now discuss the Greedy-Space scheme in detail. The name comes from the fact that the scheme takes a greedy approach and allocates the logical block with the most free space when space is needed. It simply keeps track of how much free space is available at each logical block. When applications make new write requests, the file system selects the logical block with the most free space for space allocation. Once a logical block has been selected, then for subsequent new writes, space is allocated from the same logical block until all free space is consumed. By sending new write request sequences to the same logical block, the Greedy-Space scheme maximizes n, the number of sector writes per merge, on the next `sync`.

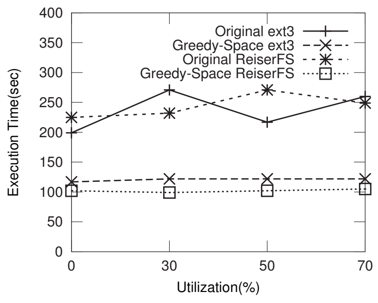
Unlike the hard drive environment this approach does not need to consider the geometrical adjacency of logical blocks, that is, the selected logical blocks need not be consecutive as there is no performance penalty for jumping between logical blocks as there would be in a hard drive when moving between cylinders. The only requirement is that the write requests are grouped within the logical block boundary.

TABLE I
MEMORY OVERHEAD FOR GREEDY-SPACE SCHEME

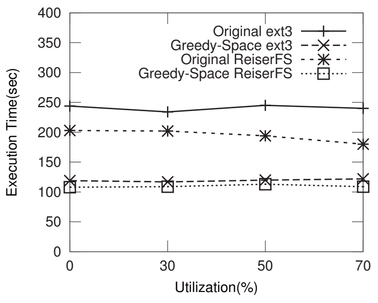
SSD	No. of Entries (Capacity/LBS)	Total Overhead (Entry size: 36B)
SanDisk (SDU5B-032G-102501)	8K (32G/4M)	36×8K = 288KB
Samsung (MCCOE64G5MPP-0VA)	8K (64G/8M)	36×8K = 288KB
Mtron (MSP-SATA7035-064)	32K (64G/2M)	36×32K = 1152KB



(a) SanDisk

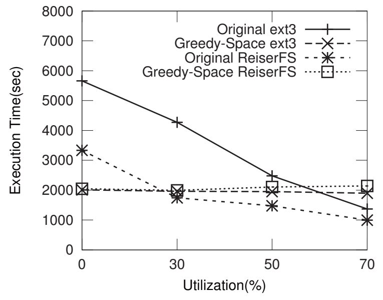


(b) Samsung

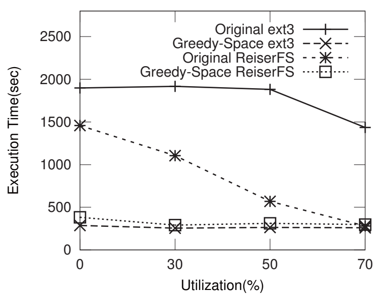


(c) Mtron

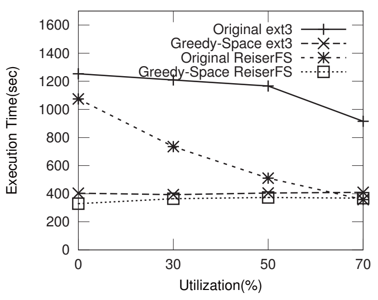
Fig. 4. Postmark benchmark: CVFS-Large



(a) SanDisk



(b) Samsung



(c) Mtron

Fig. 5. Postmark benchmark: FSL

V. Implementation and Experimental Results

The Greedy-Space allocation scheme is implemented into two file systems, namely, the ext3 and Reiser file systems that come with Linux version 2.6.27. Hence, there are two sets of ext3 and ReiserFS, that is, the original implementation and the implementation that incorporates the Greedy-Space allocation scheme. For all cases, journalling is set to the ordered mode. For ReiserFS, we mount the file system with the notail option set.

Greedy-Space maintains all the logical blocks in ordered fashion using the Red-Black tree data structure provided in Linux, so that the block with the most free space may be efficiently found. Associated with each logical block is a 36B data structure consisting of information such as the logical group number, the offset with the logical block, and pointers for the data structure. Overhead for maintaining this information for each of the SSDs used in our experiments is calculated and shown in Table I. Experiments were conducted in a system with an Intel Core 2 Duo 2.20Ghz CPU with 2GB of memory.

We use the Postmark benchmark for evaluating our scheme and set the Postmark parameters to those used by Traeger et al. in their study [20]. The I/O unit is set to 4KB and all other parameters not stated here are set to default values. Here, we take two of these benchmarks, CVFS-Large and FSL; though we have experimented with the third benchmark CVFS, we omit this because it is too small and does not provide any unique insight.

The results are presented in Figures 4 and 5, where the y-axis is the elapsed time (in seconds) to execute the benchmark, while the x-axis is the utilization of the disk observed from the file system standpoint. (Note that the y-axis scales are all different.) Utilization is initialized by randomly filling up the file system with files whose size ranges between 1KB and 16MB until the desired utilization is met. This was done to randomly spread out the valid blocks. For all experiments of the same utilization, the same set of files in the same sequences are used to fill up the file system.

Let us now discuss the results starting off with those for the CVFS-Large benchmark depicted in Figure 4. Here we see that, in general, the original ReiserFS does somewhat better than the ext3. The same file systems with the Greedy-Space scheme deployed performs much better than the original completing in roughly half the time.

Now consider the results for the FSL benchmark depicted in Figure 5. Here we notice that the results are different from those of the CVFS-Large benchmark. For this benchmark, we observe a similar trend, that is, as utilization increases the performance of the original file system improves. We even observe that in some cases the original file systems perform better than the Greedy-Space versions. This is more so for ReiserFS, and especially for the SanDisk SSD.

The fact that performance actually improves as utilization increases, at first glance, is contrary to conventional knowledge. However, we must note that utilization here is that of the file system and not the SSD. In SSDs, the FTL has its own view of what blocks are valid and what are not. When a sector is deleted in the file system, this knowledge is not reflected into the SSD until that specific sector gets overwritten. Hence, utilization from the SSD standpoint would be nearly 100% for every utilization data point once the blocks in the SSD fills up (as would be in our case as numerous experiments filling up the SSDs were conducted on these SSDs). Hence, the SSD is not directly influenced by file system utilization.

This, however, does not explain why the original file systems, especially ReiserFS, is doing so much better as utilization increases. The reason for this, we conjecture, is mainly due to the workload characteristics of FSL. FSL is a benchmark that accesses small-sized files and that is metadata operation intensive. Such a benchmark makes the environment ripe for locality-based optimization, which is a strength of the original ext3 and ReiserFS, especially the latter. As the SSD is initially filled up with files irrelevant to the benchmark to meet the utilization setup, this effect is exacerbated by the fact that all operations of the benchmark are concentrated to those files that the benchmark generates after the initial fill up of the SSD. In contrast, the Greedy-Space scheme does not consider any form of locality, hence the stable, flat performance. Incorporating locality to the Greedy-Space scheme does appear to be another promising direction of research. However, in this study, we do not consider this and leave it for the future.

VI. Conclusion

SSDs have recently been introduced into the market but little interest has been paid to making efficient use of SSDs in terms of file system performance. In this paper, we derived a write performance model of SSDs that gave insight to how space allocation should be done. Based on this observation and making use of a characteristic unique to SSDs, we presented the Greedy-Space space allocation scheme. From the Postmark benchmark results, we observed substantial performance improvements when employing the Greedy-Space allocation scheme in the ext3 and Reiser file systems running on three SSDs available in the market.

There is still much to be done. Our study has been limited to SSDs that we conjecture to employ block or hybrid mapping FTL schemes. In the future, our model and approach will have to be extended to accommodate SSDs with page mapping FTLs. Also, our model indicates that an LFS-style file system could be beneficial. Whether that should be the LFS scheme itself or something totally new tailored to SSDs is an interesting question that should also be pursued.

Acknowledgment

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. R0A-2007-000-20071-0) and by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (No. 2009-0085883).

References

[1] IBM Blade Server Product Specification, 2008, ftp://ftp.software.ibm.com/common/ssi/pm/rg/n/blo03014usen/BLO03014USEN.PDF.

[2] "Intel Introduces Solid-State Drives for Notebook and Desktop Computers," Intel News Release, 2008, http://www.intel.com/pressroom/archive/releases/20080908comp.htm.

[3] W. Hutsell, "Using SSDs to Boost Legacy RAID and Database Performance," http://www.storagesearch.com/texasmem-art3.html, 2004.

[4] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives," in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '09)*, 2009, pp. 181–192.

[5] A. Rajimwale, V. Prabhakaran, and J. D. Davis, "Block Management in Solid-State Devices," in *Proceedings of the 2009 USENIX Annual Technical Conference (USENIX '09)*, 2009, pp. 1–14.

[6] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX '08)*, 2008, pp. 57–70.

[7] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, 2008, pp. 1–14.

[8] *Understanding the Flash Translation Layer (FTL) Specification*, Intel Co., 1998, http://developers.intel.com.

[9] *Flash-Memory Translation Layer for NAND Flash (NFTL)*, M-Systems.

[10] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A Space-efficient Flash Translation Layer for CompactFlash Systems," *IEEE Transactions on Consumer Electronics*, vol. 48, no. 2, pp. 366–375, 2002.

[11] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, 1992.

[12] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write Amplification Analysis in Flash-based Solid State Drives," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference (SYSTOR '09)*, 2009, pp. 1–9.

[13] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A Superblock-based Flash Translation Layer for NAND Flash Memory," in *Proceedings of the 6th ACM/IEEE International Conference on Embedded Software*, 2006, pp. 161–170.

[14] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A Log Buffer-based Flash Translation Layer using Fully-associative Sector Translation," *ACM Transactions on Embedded Computing Systems*, vol. 6, no. 3, p. 18, 2007.

[15] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim, "A Re-configurable FTL (Flash Translation Layer) Architecture for NAND Flash-based Applications," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 4, 2008.

[16] Z. Liu, L. Yue, P. Wei, P. Jin, and X. Xiang, "An Adaptive Block-Set based Management for Large-Scale Flash Memory," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, 2009, pp. 1621–1625.

[17] J.-H. Kim, D. Jung, J.-S. Kim, and J. Huh, "A Methodology for Extracting Performance Parameters in Solid State Disks (SSDs)," in *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (IEEE MASCOTS '09)*, 2009, pp. 1–10.

[18] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk Schedulers for Solid State Drivers," in *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded Software (EMSOFT '09)*, 2009, pp. 295–304.

[19] W. Wang, Y. Zhao, and R. Bunt, "HyLog: A High Performance Approach to Managing Disk Layout," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04)*, 2004, pp. 145–158.

[20] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright, "A Nine Year Study of File System and Storage Benchmarking," *ACM Transactions on Storage*, vol. 4, no. 2, pp. 1–56, 2008.