A Performance Model and Space Allocation Schemes for SSDs

USENIX FAST 2009 Submission Version

Abstract

The recently introduced Solid State Drives (or Disks) (SSDs) have started to replace hard drives in laptop computers. The server market is also looking for an opportune time to deploy SSDs because of the many merits that they possess. 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 simple performance model for the SSD. Based on this model and SSDs' inherent characteristic that reads are constant, while writes are not, we devise two file system space allocation schemes for SSDs. Through real implementations on Linux and using three SSD products available in the market, we show that substantial performance improvements can be achieved for a wide range of workloads. We show that for most of the workloads that we experiment with, the execution time is reduced to half to one-third of the original ext2 file system.

1 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, 2, 8]. With no mechanical parts, unlike the hard drive, SSDs are light, shock-resistant, noiseless, and consumes considerably less power.

Though interest regarding SSDs has started to rise, they have mostly been directed to the internals of the SSD [3, 12]. 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 the SSD that, in essence, is identical to that of the hard drive. The model introduces a notion of a focal factor that provides guidance in allocating space in the file system. Using this model and SSDs' inherent characteristic that reads are constant, while writes are not, we devise two file system space allocation schemes for SSDs. Through real implementations on Linux 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 space allocation schemes that we propose, for most of the workloads, the execution time is reduced to half to one-third of the original ext2 file system. In one case, execution time of roughly 1300 seconds when using the original allocation scheme is reduced to around 200 seconds.

The rest of the paper is organized as follows. In Section 2, we describe the basics characteristics of Flash memory storage and SSDs as well as the related works. In Section 3, we derive write cost models for the hard drive and the SSD. Based on the implications of the models, we devise two new space allocation schemes for SSDs in Section 4. In Sections 5 and 6 we present the experimental environment and discuss the results obtained from the experiments. Then, we conclude with a summary and directions for future work in Section 7.

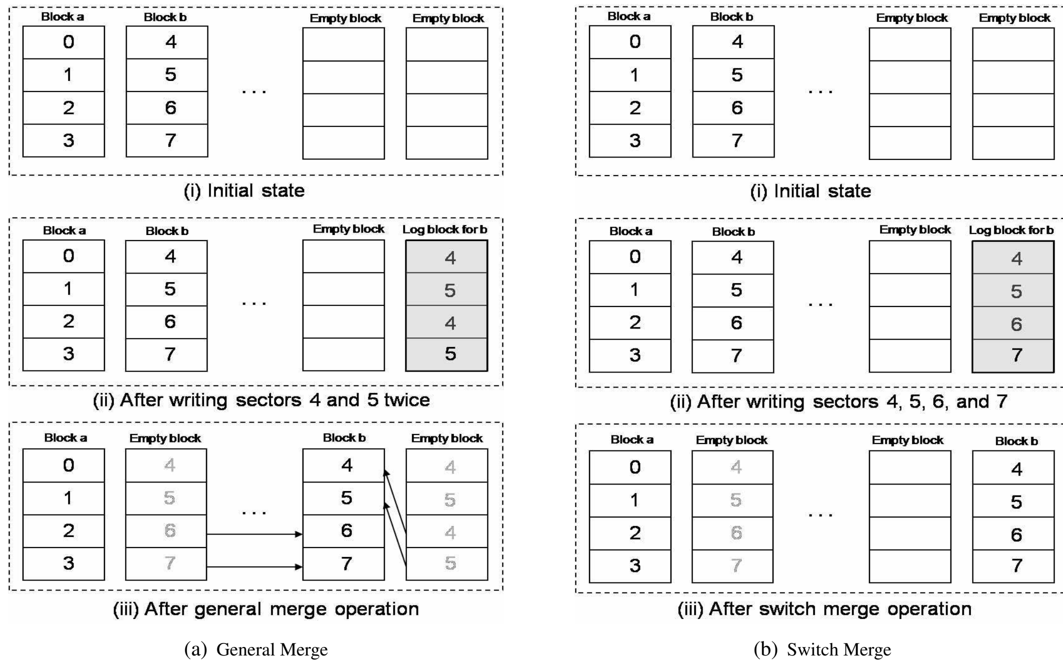
2 SSD Background and Related Works

In this section, we present background information as well as previous works related to Flash memory and SSDs that pertain to our study.

2.1 Flash memory and SSD basics

The storage medium of an SSD is Flash memory. Hence, it inherits most of the essential properties of Flash memory. Though both NOR type and NAND type Flash memory could be used, because of its low cost-per-bit, the NAND Flash memory is widely used in high capacity Flash memory storage such as SSDs and USB drives.

A NAND Flash memory chip has multiple blocks and each block has a set of pages. Typically, a block size is 16$\sim$256KB and a page size is 0.5$\sim$4KB. Data are read/written from/to in page units. It takes 20$\sim$25μsec to read a page and 200$\sim$300μsec to write a page within a Flash chip (excluding data transmission time) [25, 26]. Data once written to a page cannot be modified without erasing the block containing the page. An erase operation takes the longest time, specifically, around 1.5$\sim$2msec. To accommodate the asymmetric unit sizes



(a) General Merge (b) Switch Merge

Figure 1: Merge operations of an FTL

and times for read, write, and erase operations at the Flash memory chip level and provide a readable/writable sector disk interface, Flash memory storages employ a complex software module called the Flash Translation Layer (FTL) [9, 19].

To achieve high performance and large capacity, an SSD has numerous Flash memory chips and SRAM and SDRAM buffers controlled by a microprocessor [3]. The SRAM buffer stores data for the processor and the SDRAM buffer stores the Flash memory data that is requested. SSDs also employ FTLs to control and schedule all operations of these chips and buses. Multiple read/write requests stored in the SDRAM buffer can be served simultaneously on multiple buses and chips when possible. In most SSDs, multiple Flash memory chips comprise a single logical Flash memory chip. All same numbered blocks in all the chips form a logical block, and all same numbered pages in these blocks form a logical page. As data can be read/written from/to physical pages of a logical page in parallel, this logical union improves performance of SSDs as RAID does with hard drives. An important role of an FTL would be to exploit the multiple chips and buses so as to maximize the parallelism among these resources. As multiple chips share a bus, commands and data transfer to/from each chip on a bus occurs in an interleaved manner and, thus, parallel operations of multiple chips and buses are called chip-level and bus-level interleaving, respectively. Only through these interleaving operations can the performance of an SSD exceed that of a single Flash memory chip.

2.2 Mapping in FTL

Another important task of the FTL is to map sectors to Flash memory storage. The mapping technique of choice can have a profound effect on the behavior of Flash memory storage such as SSDs.

The two basic forms of mapping are page mapping and block mapping. Low capacity NOR Flash memory cards have used page mapping in their FTLs [9], while many high capacity NAND Flash memory storages such as USB drives and SSDs have used block mapping [13]. Hybrid approaches have also been suggested [17, 30]. Since our interest is in SSDs, hereafter, we first concentrate on block mapping.

A physical block in Flash memory holds a fixed number of sectors (N_s). In block mapping, these sectors are typically placed in pages within the block in an ordered manner. The map contains a logical block number and physical block number pair. 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. Then, since sectors are placed in order, finding the sector we want is straightforward. However, in real systems, writes make use of log blocks as we will explain shortly. Hence, to locate sectors, the log blocks are first searched before the physical block is checked.

Writing a sector is more complicated. Consider Figure 1(a)(i). To modify a sector within a data block b,

where b is a logical block number, the FTL writes the new sector data to a page of the log block. Thereafter, subsequent writes for sectors of data block b are directed to a page in this log block. Figure 1(a)(ii) shows the results after sectors 4 and 5 of the same logical block are modified twice.

Eventually, when the pages in the log block runs out or when, based on the block recycling policy, the FTL decides to retrieve the log block, a merge operation that consolidates the original data block and the log block happens. Specifically, the merge operation retrieves an empty block (generally by erasing a block with obsolete data) and copying all valid sectors to it from the old data block (sectors 6 and 7 of block b) and the log block (the most recent sectors 4 and 5 of the log block) as shown in Figure 1(a)(iii). After copying the sectors, the old empty block now becomes the new data block. The old data block and the log block become empty blocks that will be used as a log block or copy-target block in the future. The roles of the blocks are switched by updating the information in the map. This form of merging is referred to as the general merge.

Next, consider the case where writes occur in sequence. For instance, consider the initial state of Figure 1(b)(i) where sectors 0 through 7 have been stored. Now, a sequential write of sectors 4 through 7 are requested. These writes are written to a log block as before, as shown in Figure 1(b)(ii).

Now consider the merge operation. In this case, when a merge has to occur, we do not need to copy the four sectors to an empty block as was done for general merge. Instead, all that needs to be done is to update the information in the map to reflect that a switch has occurred. That is, the log block is now the data block and the old data block is now obsolete as depicted in Figure 1(b)(iii). This type of merge is called a switch merge. Switch merge eliminates copying of pages, hence is much more efficient.

Kim et al. introduced the log block scheme that was just discussed [13]. As FTL design is an important factor in SSD design, work on hybrid mapping, that is, schemes that combine block mapping and page mapping, has also been conducted. Observing that Kim et al.'s work has a weakness with random writes, Lee et al. propose a hybrid scheme called FAST (Full Associative Sector Translation) that uses sector mapping in log blocks, while block mapping is used for other data blocks [17]. Later, Yoon et al. further improve the log block scheme through heuristic based selection of block mapping and page mapping schemes for each logical block [30]. Instead of the merge operation, Lee et al. introduce the migration operation that is more efficient for repeated write patterns, and they show that write cost can be significantly reduced through cost-based selection of migration and merge operations [14]. Finally, contrary to the conventional approach, Birrel et al. propose a pure page mapping scheme based FTL for the SSD that is optimized for random write patterns, but would incur higher costs [6]. A real implementation of this design, however, is not known yet.

2.3 Other related works

There are some recent works related to Flash memory and SSDs that consider issues other than the FTL. Leventhal discusses the changing landscape of memory hierarchy in computer systems due to Flash memory storage [18]. Other than an alternate to conventional disks, making use of Flash memory in various components of a computer system such as a booting device and disk caching media have also been studied [11, 20]. Kim and Ahn show that block level LRU ordering increases the chance of switch merge in SSDs [12]. Agrawal et al. gives a taxonomy of the many design choices that are available to SSD designers and analyzes how these choices would affect performance through simulations [3]. They conclude that SSD lifetime and performance is highly workload sensitive. Finally, the enterprise database community has been an ardent advocate of adopting SSDs to boost performance and to save energy. Lee and Moon show that Flash memory aware design of algorithms can improve database application performance on Flash memory storages [15]. Also, case studies by Lee et al. show that by simply replacing hard drives with SSDs for the transaction log, rollback segments, and temporary table spaces, performance of a database application increases up to an order of magnitude in some cases [16].

3 Write Cost Models

In this section, we derive write cost models for file systems when the underlying storage is a hard drive and an SSD. (As read cost is constant for SSDs, we do not consider the read cost model.) The file systems that we consider are those that take the Overwrite approach [29]. In deriving the model we assume that repeated references are filtered in buffer caches or disk caches and write requests are issued altogether by something like the `sync` operation from the file system. We first derive the cost model for the hard drive, and then, based on this model derive a model for the SSD.

3.1 Hard drive write cost model

To derive a new cost model, we start off from the simple performance model of what Wang et al. refer to as the Overwrite approach [29]. This is the traditional approach where new data is overwritten on top of old copies.

A simple performance model for writing a sector using Overwrite 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 [29]. In reality, however, writes to disks do not occur individually. While each application may do so, requests to the disk are issued through a `sync` operation from the file system. Thus, they will be requested in groups. Assuming n write requests are requested together, we can generalize T_{1sect} as follows. First is the worst case scenario where every write request goes to a different cylinder from the previous one. In this case, the performance cost model will be nT_{1sect}, the worst performance that can be achieved. In the best case scenario the write cost to write all n sectors to the same cylinder 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 subrequests without compromising the model. Hence, we do not consider this situation any further.) In fact, maximum bandwidth would be achieved when $n = C$. Even so, most file systems making use of hard drives cannot utilize this full bandwidth even though efforts are made to localize relevant files and metadata to the same cylinder group.

In reality, the n sector write requests that are made together are dispersed among the many cylinder groups. Let us denote R as the average number of sectors that are requested for writing to disk at each time, that is, at each `sync`. As a file system, in general, cannot always write the R requests to the same cylinder, we introduce the focal factor, f, where $\frac{1}{R} \leq f \leq 1$, which refers to the proportion of write requests that are directed to a particular cylinder. Then, we can generalize the time for one sector write at each particular cylinder as follows:

$$T_{magdisk} = \frac{T_{pos} + fR\frac{S}{B}}{fR} = \frac{T_{pos}}{fR} + \frac{S}{B} \qquad (1)$$

where $R \leq C$ and $\frac{1}{R} \leq f \leq 1$.

In file systems that take the Overwrite approach, f will generally be much smaller than 1 because relevant files and metadata will not always be able to be placed within the same cylinder group.

3.2 SSD write cost model

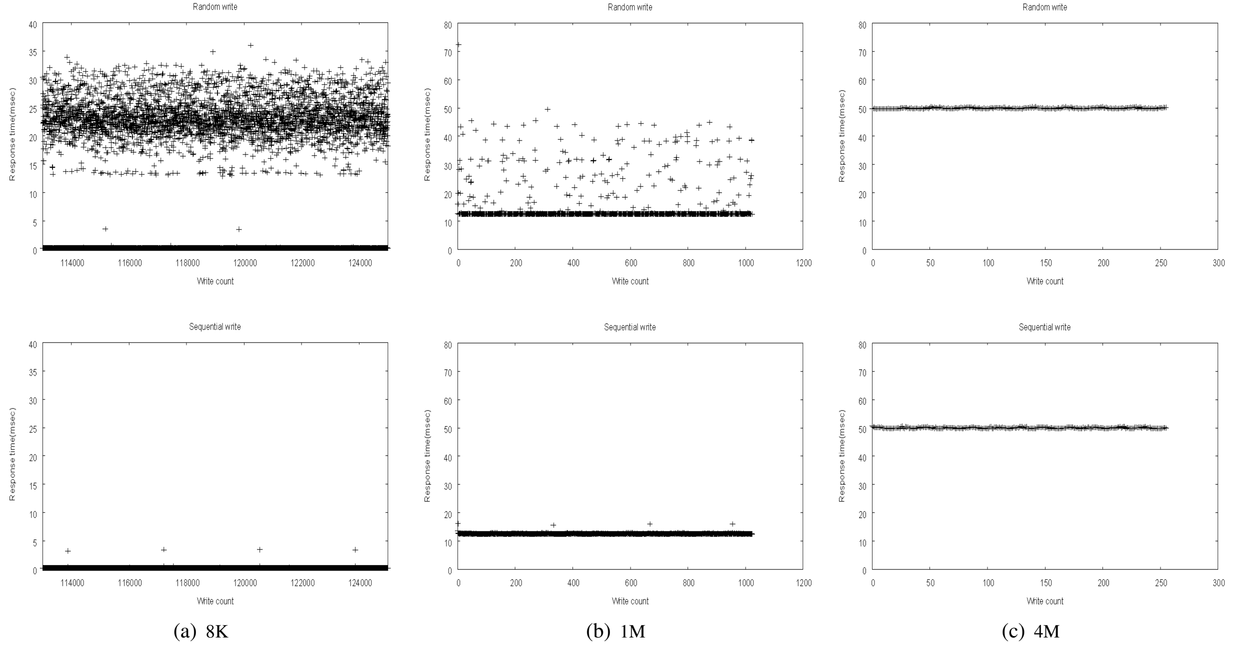
In this subsection, we derive the write cost model when the file system takes the Overwrite approach using an SSD as its underlying storage.

Observe the write performance shown in Figure 2. This figure shows the response time for each block write request of the corresponding block size when requesting a total of 1GB to one of the SSDs that we describe later. (For the 8K block size, we show only part of the results as the excessive number of data points for the whole graph makes the graph difficult to comprehend. However, the trend presented here repeats itself throughout the write sequence.) 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. The request sequence of the lower figures is sequential, that is, blocks are requested in sequence starting from 0, and the upper figures are for random request sequences. Observe from the lower graphs of Figure 2(a) and (b) that a thick band along the x-axis forms meaning that the majority of requests are serviced in that time frame. (Ignore Figure 2(c) for now as we will come to this figure later in Section 3.3.) For block size of 8KB, the band forms around the 200μsec range (which we could not show clearly due to the scale and resolution) and for the 1MB block size it forms near the 12 millisecond range. This increase is due to the size of the request and the limited bandwidth available. Then, there are numerous "spikes", that is, response times that are out-of-band. These spikes tend to be as low as a few milliseconds to as high as a few tens of milliseconds. Contrasting the upper and lower figures, we see that for the sequential writes, these spikes are few. However, when the 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 [3]. However, based on our understanding of Flash memory and FTL software described in Section 2, 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 the simple switch merge, while the larger spikes being for those of general merges with numerous copy operations.

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 make a similar argument as was described in the previous subsection. Using the same notations, that is, B, the write bandwidth of an SSD and S, the sector size in bytes, we can argue that the worst case write performance model to write a single sector in SSD is $T_{1sect} = T_{merge} + \frac{S}{B}$.

A similar argument can be made for the best case with



(a) 8K (b) 1M (c) 4M

Figure 2: Appearances of merge spikes mainly due to erase operations for particular block sizes (random writes: upper figures; sequential writes: lower figures)

SSDs. Since a merge operation is inevitable, the best performance will happen when the n sectors are written with overhead for only a single merge operation. Hence, the write cost to write all n sectors 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 and something determined from the hardware. In an SSD, there is no concept of a cylinder. (Though this notion is represented when partitioning the SSD, this does not, in any way, match to any physical characteristics as in the hard drive.) 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.

So, given the performance model above, we can generalized as we did with the hard drive. Again, let us denote R as the average number of sectors that are requested for writing at each disk write, that is, at each `sync`. As a file system may not be able to always write the R requests to the same logical block, we again introduce f, the focal factor, where $\frac{1}{R} \leq f \leq 1$, which refers to the proportion of write requests that are directed to a particular logical block. Then, just like the hard drive, we can generalize the write cost function for a single sector write to a particular logical block in an SSD as follows:

$$T_{SSD} = \frac{T_{merge} + fR\frac{S}{B}}{fR} = \frac{T_{merge}}{fR} + \frac{S}{B} \qquad (2)$$

where $R \leq L$ and $\frac{1}{R} \leq f \leq 1$.

Though the model is simple, it suffices to guide us in developing a new space allocation scheme for SSDs.

3.3 Implications of the model

The write cost model for SSDs that we just derived gives us insights for optimizing file systems and disk scheduling algorithms for SSDs. At first glance, write behavior of an SSD seems to be the same as a magnetic disk. For example, the models imply that sequential writes are preferable to random writes in both the hard drive and the SSD. This might imply that file systems that make use of an LFS-style write, which collects modified data in memory chunks and then writes to disk altogether, might also be an interesting topic that should be investigated as SSDs become popular. Unfortunately, this is beyond the scope of this paper as we limit our focus to file systems using the Overwrite approach.

The thing to note here is that based on the write cost model derived, there are features in SSDs that may be exploited to improve performance. Specifically, first, note that given a logical block (which is discussed in more detail below), controlling f 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. Hence, devising a scheme to increase f under the Overwrite approach is feasible.

The second 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 "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 2. As discussed previously, Figure 2(a) and (b) shows spikes in response time due to merge operations within the SSD. Now, note Figure 2(c) when the write block size is 4MB. In this figure we do not observe any large intermittent spikes. 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. Hence, when at all possible, all writes should be made at this size. We refer to a block of this size to be the logical block. In the next section, we discuss the empirical aspect of a logical block.

4 Space Allocation for SSD

This section describes the space allocation schemes that we develop. In order for these schemes to work, we need to make concrete the notion of a logical block. To do this we start out with more of logical blocks.

4.1 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 evidenced in Figure 2(c).

Many design decisions affect the performance of SSDs [3, 12]. 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, busses, 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_SYNC option. Then, starting from a small write block size that is a power of 2, say 4KB, we do the following. First, sequentially write 1GB into the device file obtaining its throughput. Then, 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 a rather ad hoc value chosen such that all log blocks managed by the FTL are safely consumed. In current implementations of SSDs, 1GBs is a fairly safe size.) The throughput is obtained here as well. The throughputs for the sequential and random writes are compared. If the values are close enough, then 4KB 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.

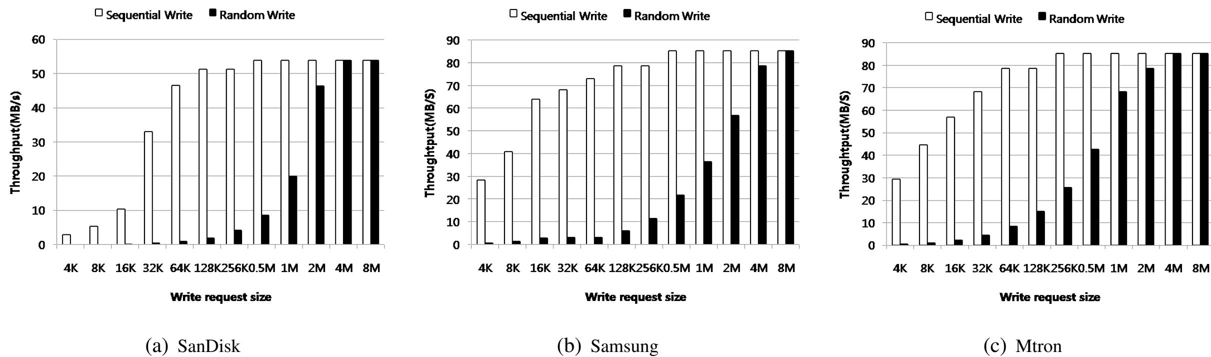
Figure 3 shows results after going through the process just described starting from a write size of 4KBs for three types of SSDs that we use to validate our study. (We describe the SSDs in detail later.) Note how the random write throughput converges towards the sequential write throughput eventually becoming the same at some point. Each of the figures in Figure 2 represent a point in Figure 3(c), that is, the Mtron SSD. With write performance for sequential and random requests converging at the 4MB block size as shown in Figure 3(c), we observe that merge overhead minimizes as was observed in Figure 2(c).

4.2 Design of space allocation schemes

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, in this subsection, we propose two new space allocation schemes for file systems that employ SSDs based on the performance model presented in the previous section. Note from Equation 2 that in order to reduce the write cost the focal factor f should be increased as the other parameters, including R, are all fixed values.

The key idea of the schemes that we propose is summarized in Figure 4. There are five files A, B, C, D, and E and space is allocated as shown with each subscript representing a block or metadata of that file. Representing files as sets, we have file $A = \{A_1, A_2, A_3\}$, $B = \{B_1, B_2\}$ and so on. We then have a new set of 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.

Figure 4(a) depicts the traditional scheme of allocating space for new blocks. Each new block is allocated to the cylinder group where the metadata and the rest of the file it belongs to resides. The key idea in the scheme that we propose is to allocate all the new blocks to the logical block that has the most free blocks as depicted in Figure 4(b). That is, blocks $A_4, A_5, C_5, E_2, E_3, E_4, E_5$



(a) SanDisk (b) Samsung (c) Mtron

Figure 3: Finding the logical block size

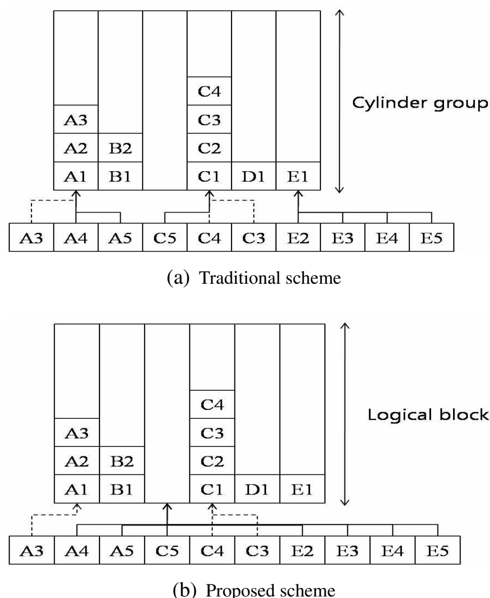
are all allocated to the same logical block. By so doing, we increase the focal factor, f. This does not incur any penalty for reads as reads in SSDs are constant, unlike hard drives. Note that writes to existing blocks are done in overwrite fashion in both schemes.

Specifically, the first allocation scheme is the Greedy-Space scheme. 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 the focal factor f 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.

A limitation of the Greedy-Space scheme, however, is that it incurs bookkeeping overhead. For single SSDs of today the overhead is only a few hundred KBs (as we show later in Section 5). However, capacity of SSDs is growing at a rapid pace, and if we consider RAID-like systems with high capacity, then this overhead may not be negligible. Also, the file system needs to continuously keep a sorted list of blocks based on the amount of free space available, which incurs computational overhead.

Our second space allocation scheme, which we call the Clock-Space scheme is a scheme that is more amenable for real-life deployment. Instead of keeping track of every logical block in the system, here, each logical block is checked one at a time. The name of the scheme comes from its similarity to the the Clock page replacement algorithm where the clock hand moves from one page to the next in clockwise order. As the write requests arrive and new space has to be allocated, the scheme starts out by moving the clock hand to the next logical block. At this logical block, it checks to see if the number of occupied blocks (multiple sectors) is below some threshold value. If not, then the clock hand moves on until the condition is satisfied. Once the logical block has been selected, the current write requests and write requests following thereafter are allocated to this logical block until it fills up.



(a) Traditional scheme

(b) Proposed scheme

Figure 4: Example of space allocation schemes

Let us now describe how the threshold value, which is set dynamically depending on the load of the system, is determined. For this, we borrow from a

model proposed in regards to LFS. Specifically, to select a segment for cleaning the notion of average segment occupancy (ASO) and the best segment occupancy (BSO) are introduced and their relationship, ASO $=$ $(1-\text{BSO})/ln(1/\text{BSO})$ is derived [21]. This relationship essentially says that given that segments are occupied on average by ASO on the whole, the segments that are least occupied are those that occupy BSO space, and these should be selected for cleaning. Through simulations, Wang et al. later verify that this relationship is quite accurate [29].

We take this result and apply it to our threshold value. As we can easily find the average load of logical blocks on the whole system (that is, ASO), the threshold value is set to the BSO value. That is, by selecting logical blocks with load lower than the BSO value, we are selecting blocks with light loads.

5 Experimental Environments

In this section, we describe the environment in which all our experiments were conducted. Then, we describe the implementation details of the Greedy-Space and Clock-Space schemes within the environment.

5.1 The platform

Table 1: SSDs used in the experiments

Manufacturer & Part #	Capacity	Interface
SanDisk SDS5C-032G-102500	32G	SATA
Samsung MCCOE64G5MPP-0VA	64G	SATA
Mtron MSD-SATA3025-032	32G	SATA

The environment in which the experiments are conducted is the Linux operating system version 2.6.23.1 running on an Intel Core 2 Duo 2.20Ghz CPU with 2GB of memory. The SSDs that we consider are listed in Table 2.

Each of these products use proprietary hardware and software designs in their products, though a flavor of the hardware schemas and FTL related design issues may be guessed upon [3, 22, 30]. We emphasize again that it is difficult, if not impossible, to know the exact details of the hardware and software design in real products, let alone how the software and hardware interact.

5.2 Implementation of the schemes

The implementation of the Greedy-Space scheme is straightforward. The Linux file system space is divided into block groups (which is Linux jargon for cylinder groups). These block groups are divided into logical blocks determined by the empirical method described in Section 4. The logical block size (LBS) for the SanDisk, Samsung, and Mtron SSDs turn out to be 4MB, 8MB, and 4MB, respectively. Hence, depending on the SSD used, each block group is divided into some specific number of logical blocks.

Table 2: Memory overhead for Greedy-Space scheme

SSD	No. of Entries (Capacity/LBS)	Total Overhead (Entry size: 36B)
SanDisk	8K (32G/4M)	$36\times8\text{K} = 288\text{KB}$
Samsung	8K (64G/8M)	$36\times8\text{K} = 288\text{KB}$
Mtron	8K (32G/4M)	$36\times8\text{K} = 288\text{KB}$

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. We also keep a hash table to efficiently find a logical block. 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.

Hence, the overhead for maintaining this information for the Greedy-Space scheme for each of the SSDs can be calculated as shown in Table 2. Given the capacity of the disk and the logical block size (LBS), the number of entries that needs to be managed can be calculated by dividing the capacity by the LBS. As overhead for each entry is 36 bytes, the total overhead is obtained by multiplying 36 to the number of entries. The hash table managed in block group units adds another 10KB of overhead (which we did not include in Table 2).

The implementation of the Clock-Space scheme is more platform specific. As mentioned previously, Linux manages file system space in block groups, which in our experiments are composed of 4KB blocks. To manage the 4KB blocks in block groups, ext2 keeps a bit map of the usage of these blocks, and this bit map also fits in one 4KB block. Hence, when the clock hand of the Clock-Space scheme moves from one logical block to the next, it is much more efficient to read the whole block group bit map altogether than to read a particular number of bits within the bit map block. Once the block group bit map is read, we take the number of bits that comprise the logical block and calculate their load. For example, for a SanDisk SSD, since a block is 4KB in ext2 and the logical block is 4MB, 1024 bits comprise the logical block. Of these 1024 bits, let us say that we counted 256 reset bits (that is, 256 unused blocks), then the load of this logical block is 0.25. This value is compared to the threshold value.

In the previous section we described how the ASO and BSO values are used to determine the threshold. However, in a real system, floating point calculations should be avoided for optimized performance. Hence,

even though the exact BSO numbers are available [24], we choose to make use of gross approximations of these numbers instead when deciding whether to pass or use a particular logical block. The numbers that we use are shown in Table 3. In particular, for ASO values between the given range we use the ceiling threshold value. For example, if the ASO value is 0.45, then the threshold value is set to 0.2.

Table 3: Threshold values used in Clock-Space implementation (ASO: Average Segment Occupancy, BSO: Best Segment Occupancy)

ASO	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
BSO	0.1	0.1	0.1	0.1	0.2	0.3	0.45	0.6	0.8

6 Experimental Results

In this section, we present the results of the experiments performed on the platform described in the previous section. The subsections are divided according to the workload that we used for the experiments.

6.1 Postmark Benchmark: Type I

Table 4: Parameters of Postmark benchmark: Type I

Workload	No. of files (Subdirectories)	File size	Transactions
FSL	20,000 (200)	0.5KB−10KB	200,000
CVFS-Large	5,000 (50)	512B−328072B	20,000

The Postmark benchmark, which models workloads of large Internet servers such as systems used for electronic mail, netnews, and web based services, has been widely used to evaluate disk based file system performance [10]. The Postmark benchmark creates a specified number of files and directories of a particular size range. After creating the files, the benchmark executes a specified number of transactions where each transaction is a pair of file operations that either creates or deletes a file and then appends to an existing file in 512B units or reads another existing file in its entirety.

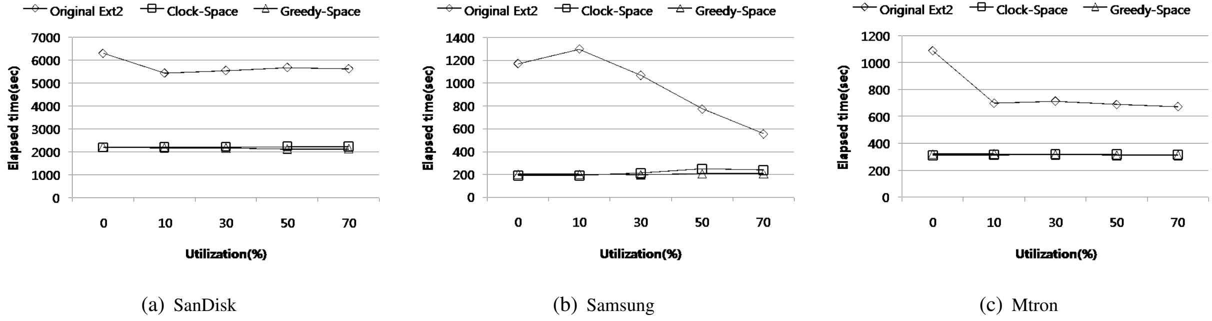
Instead of generating our own set of Postmark benchmarks based on parameters of our choice, we choose to make use of two Postmark benchmarks used in previous studies. One is used in this subsection and the other in the next subsection. In all our Postmark experiments, we use Postmark version 1.51.

For the first set of Postmark benchmarks, which we denote as Type I, we set the Postmark parameters to those used by Traeger et al. in their study on the quality of benchmarks used in performance evaluation studies [28]. They make use of three benchmarks that were used in previous studies, two of which are identical to the ones used previously [4, 27] and one that is a modification of the one used by Soules et al. [27]. Here, we take two of these benchmarks, FSL and CVFS-Large; 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 parameters of the FSL and CVFS-Large benchmarks are summarized in Table 4. Also, the I/O unit is set to 4KB, and all other parameters not stated here are set to default values.

The results are presented in Figures 5 and 6, 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.) The utilization is initialized by first completely filling up the file system with files whose size range between 1KB and 16MB, then randomly selecting and deleting these files 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 files are deleted. We make the following observations from these results.

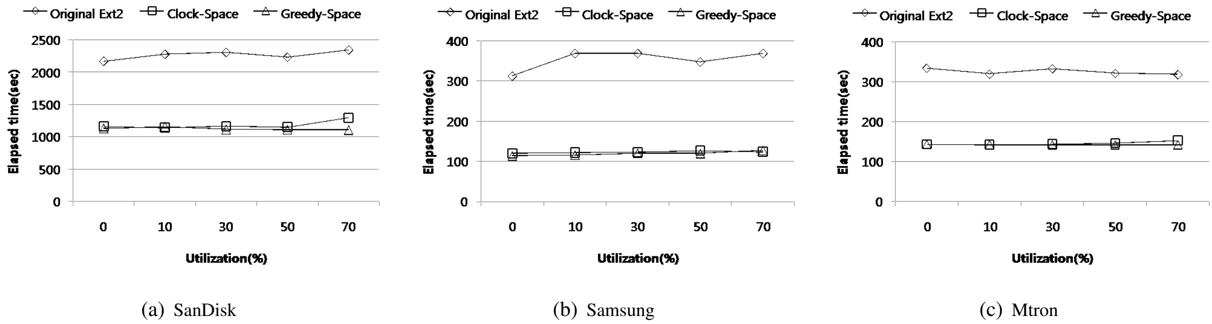
Even though the three SSDs show varying performance results, we observe that both the Greedy-Space and Clock-Space schemes perform substantially better than the original ext2. In most cases, the elapse time is reduced to around one-third of the original scheme, and in the best case, the original elapsed time of roughly 1300 seconds is reduced to around 200 seconds. Greedy-Space and Clock-Space perform better than the original scheme irrespective of utilization. Furthermore, the performance of the two proposed schemes remain quite stable for all utilizations, while the original scheme can be somewhat sensitive as exemplified by the results shown in Figure 5(b) and (c). Between the Greedy-Space and Clock-Space schemes, the Greedy-Space scheme generally performs slightly better.

A somewhat strange observation can be found in Figure 5(b) and (c) where performance actually improves as utilization increases. This, at first glance, is contrary to conventional belief. 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 because of the way the file system is aged. Hence, the SSD is not directly influenced by the file system utilization, but rather, it is strongly influenced by the focal factor. In these results, as utilization gets higher from the ext2 standpoint, it may be inadvertently allocating sectors close together because there is less room to maneuver, thereby increasing f, resulting in improved performance.



(a) SanDisk (b) Samsung (c) Mtron

Figure 5: Postmark benchmark: Type I (FSL)



(a) SanDisk (b) Samsung (c) Mtron

Figure 6: Postmark benchmark: Type I (CVFS-Large)

6.2 Postmark Benchmark: Type II

Table 5: Parameters of Postmark benchmark: Type II

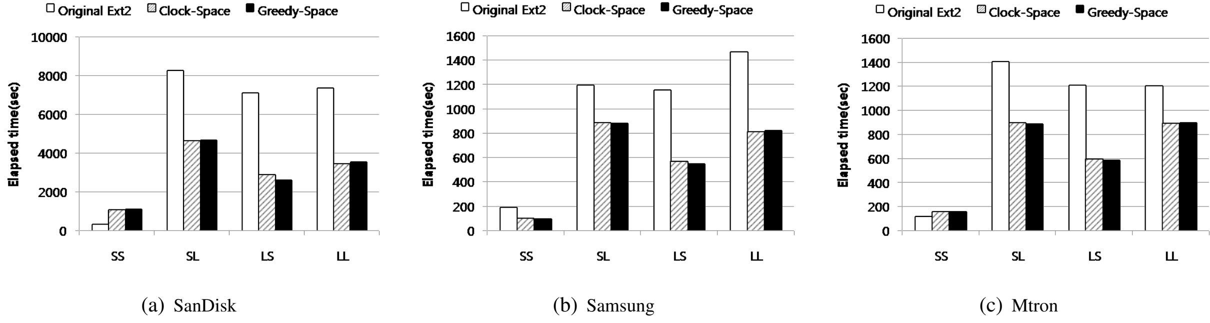
Workload	File size	Number of files	Transactions
SS	9-15KB	10,000	100,000
SL	9-15KB	200,000	100,000
LS	0.1-3MB	1,000	20,000
LL	0.1-3MB	4,250	20,000

In their study on the effectiveness of optimizations at various layers of the I/O path, Riska et al. use 4 sets of Postmark benchmarks to represent a wide variety of workloads [23]. This is done by varying the number of files and the working set size; and this is the second set of Postmark benchmarks that we use to evaluate the proposed schemes. The number of files and the file sizes used as parameters to generate the workload are summarized in Table 5 with all other parameters set to the default values except for I/O size that is set to 4KB.

The two letters used for the name of the workload represent the size of files and the working set size, which is controlled by the number of files. That is, SS represents the Postmark benchmark that is set to generate small files and a small working set, while SL refers to one that generates small files but a large working set, and so on. All the experiments were done with utilization set to 0%, that is, just after format.

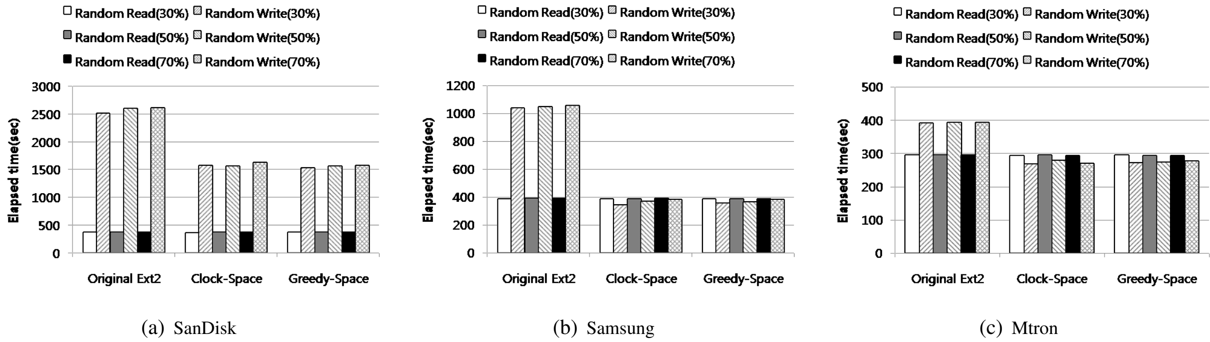
Figure 7 shows the results where the x-axis is the workload type and the y-axis is the elapsed time in seconds. Similarly to the Postmark Type I benchmarks the Greedy-Space and Clock-Space schemes perform substantially better the the original scheme.

There are two exceptions in the results, however. Specifically, with the SS workload for SanDisk and Mtron, the original scheme performs better than the Greedy-Space and Clock-Space schemes. It is difficult to pinpoint exactly the reason behind this disparity. However, we conjecture that the main reason for this is due to the effect of locality, which becomes more prominent for a smaller working set. As the working set is small (and with only a small number of these files), for the original scheme, most writes are concentrated to the front part of the block group. That is, block group management data and file/directory data are likely being serviced within a single logical block. For the proposed schemes, where as block group management data is allocated to the logical block at the head of the block group, file and directory data would be dispersed to a separate logical block increasing overhead. Now for the Samsung SSD, since the logical block size is 8MB, it is doing a better job of containing the bookkeeping information and data blocks in the same logical block, hence the better performance.



(a) SanDisk (b) Samsung (c) Mtron

Figure 7: Postmark benchmark: Type II



(a) SanDisk (b) Samsung (c) Mtron

Figure 8: Flexible I/O benchmark

6.3 Flexible I/O Benchmark

The third benchmark is the Flexible I/O (FIO) benchmark. FIO is another I/O benchmark that has previously been used for performance studies, though it has not been used as prevalently as the Postmark benchmark [5, 7]. This benchmark was originally made as a benchmark and a stress verification tool. It provides many parameters that can be set to test various forms of I/O. Table 6 shows our parameters of choice with all other parameters aside from these set to default values.

Table 6: Parameters of Flexible I/O Benchmark

Number of jobs	10
File size	700MB
Number of files	1000
I/O depth	1
I/O engine	sync
Buffered	unbuffered
I/O type	randread/randwrite
R/W unit	4KB

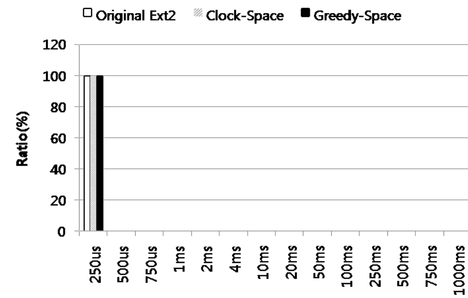
We choose this benchmark to verify aspects of performance that are not reported by the Postmark benchmark. Of the many numbers that are reported by FIO, we present two graphs. The first is shown in Figure 8, which show the total elapsed time for reads and writes for the three SSDs when the experiments are started with utilizations of 30%, 50%, and 70%. The numbers reported are the averages of the 10 jobs of the benchmark. The second graph, Figure 9, shows the distribution of the response times for the read and write operations observed while executing the benchmark on the Mtron SSD. The x-axis shows the elapsed time to execute the operation and the y-axis represents the rate of requests that complete in between the previous x value and this x value. For example, in Figure 9(b), we see that the ratio is approximately 60% for 750μsec in the Greedy-Space scheme. This is to say that approximately 60% of all writes were greater than 500μsec (which is the previous x value) and less than or equal to 750μsec.

From Figure 8, we make the same observation that write performance is substantially better with the two schemes that we propose than the original ext2 scheme. More importantly, we observe that the assumption that we made about read operations, that is, that reads will be the same irrespective of the placement policy, holds. This is further supported by Figure 9. From Figure 9(a) we observe that all reads take less than 250μsec irrespective of the allocation scheme. Note that 250μsec was the minimum value reported by FIO (which is rather coarse) and that this is the elapsed time observed by FIO, not the time spent in the Flash chip. Though these numbers together with the read results Figure 8 show that for all

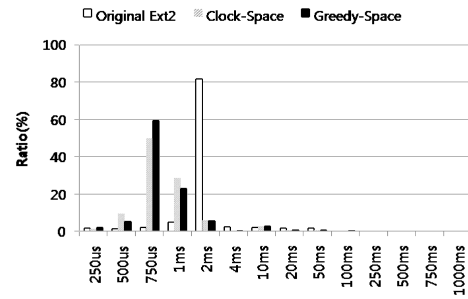
SSDs and allocation schemes, read time is within a reasonably predictable range.

From Figure 9(b), for write operations, we observe that the majority of writes for the original scheme is happening at the 1∼2msec range. From this, we can deduce that many general merge operations that incur multiple copy operations and possibly an erase operation is happening. This is in contrast to the numbers for the Greedy-Space and Clock-Space schemes where the majority of the writes are below the 1msec range with more of the Greedy-Space scheme writes having smaller values. As an erase operation within a chip generally takes more than a millisecond, we can safely deduce that for these write operations a merge that incurs an erase is not happening. From these results, we can conclude that by employing the proposed schemes the number of merge operations incurring erase operations is being drastically reduced.

Finally, for the Samsung and Mtron results of Figure 8, we see that write time is taking less than the read time. This is because of the 4KB request size. For reads of this size, all reads are being done synchronously disallowing any form of interleaving that is essential in improving throughput. Though writes are synchronous for this benchmark as well, it seems that the SSDs are buffering the requests and then making multiple writes to chips through interleaving. This is evidenced by the fact that many of the writes have higher response times than reads as shown in Figure 9. This means that though individually the writes are taking longer, since many of them are being serviced together through interleaving, the elapsed time does not add up, but rather overlap. Hence, the overall elapsed time is considerably less. As the Greedy-Space and Clock-Space schemes increase the focal factor, there is also more chances for interleaving, hence more of the writes respond better than the original scheme.



(a) Read response time



(b) Write response time

Figure 9: FIO benchmark read/write response time distribution

7 Conclusion

We have considered the use of Solid State Drives (Disks) (SSDs) that have recently been introduced into the market. With no mechanical parts, these devices have been commended for possessing interesting features such as being lightweight, shock-resistant, noiseless, and consuming low power. However, there have been little interest in making efficient use of SSDs in terms of performance. In this paper, we attempted to model the characteristics of SSD write performance from the file system standpoint. In so doing, a simple performance model was derived 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 and Clock-Space space allocation schemes.

Through real implementations in the Linux ext2 file system and using three SSD products available in the market, we performed a wide range of experiments with the Postmark and Flexible I/O benchmarks. For all the benchmarks, we observed substantial performance improvements when employing the two allocation schemes that we proposed compared to the original scheme in the ext2 file system. For most of the workloads, the execution time is reduced to half to one-third of the original ext2 file system. In one case, execution time of roughly 1300 seconds when using the original allocation scheme is reduced to around 200 seconds.

There are still much to be done. First, we need to look more closely at the logical block size. We mentioned that ideally the logical block size should bring about the optimal performance. Whether this is true in practice, or whether an optimal size really exists at all needs to be investigated. Second, observations of our results indicate that locality may play a factor in performance even in SSDs. Whether this is true or not, and if so, whether this could be incorporated to our schemes is a question that warrants investigation. Finally, 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.

References

[1] IBM Blade Server Product Specification, 2008. http://www-03.ibm.com/systems/bladecenter/.

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

[3] 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*, 2008.

[4] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '04)*, pages 129–143, 2004.

[5] J. Axboe. Fio - flexible io tester. http://freshmeat.net/projects/fio/.

[6] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A Design for High-Performance Flash Disks. *SIGOPS Operating Systems Review*, 41(2):88–93, 2007.

[7] N. Gupta. IO Containment. In *Proceedings of the Linux Symposium*, pages 151–161, Ottawa, Canada, 2008.

[8] W. Hutsell. Using SSDs to Boost Legacy RAID and Database Performance. http://www.storagesearch.com/, 2004.

[9] Intel Co. *Understanding the Flash Translation Layer (FTL) Specification*, 1998.

[10] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997.

[11] T. Kgil, D. Roberts, and T. Mudge. Improving NAND Flash Based Disk Caches. In *Proceedings of the 35th ACM International Symposium on Computer Architecture (ISCA '08)*, pages 327–338, 2008.

[12] 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)*, pages 1–14, 2008.

[13] 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*, 48(2):366–375, 2002.

[14] J. Lee, S. Kim, H. Kwon, C. Hyun, S. Ahn, J. Choi, D. Lee, and S. H. Noh. Block Recycling Schemes and their Cost-based Optimization in Nand Flash Memory based Storage System. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT '07)*, pages 174–182, 2007.

[15] S.-W. Lee and B. Moon. Design of Flash-Based DBMS: An in-page Logging Approach. In *Proceedings of the 2007 ACM International Conference on Management of Data (SIGMOD '07)*, pages 55–66, 2007.

[16] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A Case for Flash Memory SSD in Enterprise Database Applications. In *Proceedings of the 2008 ACM International Conference on Management of Data (SIGMOD '08)*, pages 1075–1086, 2008.

[17] 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*, 6(3):18, 2007.

[18] A. Leventhal. Flash Storage Memory. *Communications of the ACM*, 51(7):47–51, 2008.

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

[20] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. Intel®Turbo Memory: Nonvolatile Disk Caches in the Storage Hierarchy of Mainstream Computer Systems. *ACM Transactions on Storage*, 4(2):1–24, 2008.

[21] J. Menon. A Performance Comparison of RAID-5 and Log-Structured Arrays. In *Proceedings of the IEEE Symposium on High-Performance Distributed Computing*, pages 167–178, 1995.

[22] Mtron Co. *Solid State Drive MSD-SATA3025 Product Specification*, 2007.

[23] A. Riska, J. Larkby-Lahet, and E. Riedel. Evaluating Block-level Optimization through the IO Path. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 1–14, 2007.

[24] J. T. Robinson. Analysis of Steady-state Segment Storage Utilizations in a Log-structured File System with Least-utilized Segment Cleaning. *SIGOPS Operating Systems Review*, 30(4):29–32, 1996.

[25] Samsung Electronics Co. *512M x 8Bit / 256M x 16Bit NAND Flash Memory (K9K4GXXX0M) Data Sheets*, 2003.

[26] Samsung Electronics Co. *1G x 8Bit / 2G x 8Bit NAND Flash Memory (K9L8G08U0M) Data Sheets*, 2005.

[27] C. A. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata Efficienty in Versioning

File Systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '03)*, pages 43–58, 2003.

[28] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright. A Nine Year Study of File System and Storage Benchmarking. *ACM Transactions on Storage*, 4(2):1–56, 2008.

[29] 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)*, pages 145–158, 2004.

[30] J. H. Yoon, E. H. Nam, Y. J. Seong, H. Kim, B. S. Kim, S. L. Min, and Y. Cho. Chameleon: A High Performance Flash/FRAM Hybrid Solid State Disk Architecture. *IEEE Computer Architecture Letters*, 7(1):17–20, 2008.