Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems

Yongseok Oh[1], Jongmoo Choi[2], Donghee Lee[1], and Sam H. Noh[3]

[1]*University of Seoul, Seoul, Korea, {ysoh, dhl_express}@uos.ac.kr*
[2]*Dankook University, Gyeonggi-do, Korea, choijm@dankook.ac.kr*
[3]*Hongik University, Seoul, Korea, http://next.hongik.ac.kr*

Abstract

Hybrid storage solutions use NAND flash memory based Solid State Drives (SSDs) as non-volatile cache and traditional Hard Disk Drives (HDDs) as lower level storage. Unlike a typical cache, internally, the flash memory cache is divided into cache space and over-provisioned space, used for garbage collection. We show that balancing the two spaces appropriately helps improve the performance of hybrid storage systems. We show that contrary to expectations, the cache need not be filled with data to the fullest, but may be better served by reserving space for garbage collection. For this balancing act, we present a dynamic scheme that further divides the cache space into read and write caches and manages the three spaces according to the workload characteristics for optimal performance. Experimental results show that our dynamic scheme improves performance of hybrid storage solutions up to the off-line optimal performance of a fixed partitioning scheme. Furthermore, as our scheme makes efficient use of the flash memory cache, it reduces the number of erase operations thereby extending the lifetime of SSDs.

1 Introduction

Conventional Hard Disk Drives (HDDs) and state-of-the-art Solid State Drives (SSDs) each has strengths and limitations in terms of latency, cost, and lifetime. To alleviate limitations and combine their advantages, hybrid storage solutions that combine HDDs and SSDs are now available for purchase. For example, a hybrid disk that comprises the conventional magnetic disk with NAND flash memory cache is commercially available [30]. We consider hybrid storage that uses NAND flash memory based SSDs as a non-volatile cache and traditional HDDs as lower level storage. Specifically, we tackle the issue of managing the flash memory cache in hybrid storage.

The ultimate goal of hybrid storage solutions is providing SSD-like performance at HDD-like price, and achieving this goal requires near-optimal management of the flash memory cache. Unlike a typical cache, the flash memory cache is unique in that SSDs require *over-provisioned space (OPS)* in addition to the space for normal data. To make a clear distinction between OPS and space for normal data, we refer to the space in flash memory cache used to keep normal data as the *caching space*.

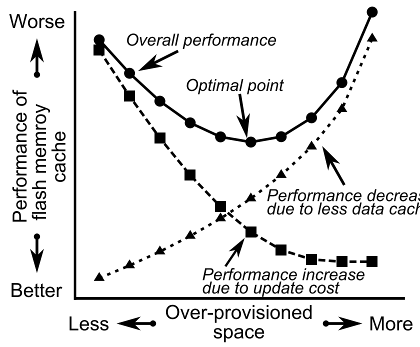


Figure 1: Balancing data in cache and update cost for optimal performance

The OPS is used for garbage collection operations performed during data updates. It is well accepted that given a fixed capacity SSD, increasing the OPS size brings about two consequences [11, 15, 26]. First, it reduces the caching space resulting in a smaller data cache. Less data caching results in decreased overall flash memory cache performance. Note Figure 1 (not to scale) where the x-axis represents the OPS size and the y-axis represents the performance of the flash memory cache. The dotted line with triangle marks shows that as the OPS size increases, caching space decreases and performance degrades.

In contrast, with a larger OPS, the update cost of data in the cache decreases and, consequently, performance of the flash memory cache improves. This is represented as the square marked dotted line in Figure 1. Note that as the two dotted lines cross, there exists a point where performance of the flash memory cache is optimal. The goal of this paper is to find this optimal point and use it in managing the flash memory cache.

To reiterate, the main contribution of this paper is in presenting a dynamic scheme that finds the workload dependent optimal OPS size of a given flash memory cache such that the performance of the hybrid storage system is optimized. Specifically, we propose cost models that are used to determine the optimal caching space and OPS sizes for a given workload. In our solution, the caching space is further divided into read and write caches, and we use cost models to dynamically adjust the sizes of the three spaces, that is, the read cache, write cache, and the OPS according to the workload for optimal hybrid storage performance. These cost models form the basis of the Optimal Partitioning Flash Cache Layer (OP-FCL) flash memory cache management scheme that we propose.

Experiments performed on a DiskSim-based hybrid storage system using various realistic server workloads show that OP-FCL performs comparatively to the offline optimal fixed partitioning scheme. The results indicate that caching as much data as possible is not the best solution, but caching an appropriate amount to balance the cache hit rate and the garbage collection cost is most appropriate. That is, caching less data in the flash memory cache can bring about better performance as the gains from reduced overhead for data update compensates for losses from keeping less data in cache. Furthermore, our results indicate that as our scheme makes efficient use of the flash memory cache, OP-FCL can significantly reduce the number of erase operations in flash memory. For our experiments, this results in the lifetime of SSDs being extended by as much as three times compared to conventional uses of SSDs.

The rest of the paper is organized as follows. In the next section, we discuss previous studies that are relevant to our work with an emphasis on the design of hybrid storage systems. In Section 3, we start off with a brief review of the HDD cost model. Then, we move on and describe cost models for NAND flash memory storage. Then, in Section 4, we derive cost models for hybrid storage and discuss the existence of optimal caching space and OPS division. We explain the implementation issues in Section 5 and then, present the experimental results in Section 6. Finally, we conclude with a summary and directions for future work.

2 Related Work

Numerous hybrid storage solutions that integrate HDDs and SSDs have been suggested [8, 11, 14, 29]. Kgil et al. propose splitting the flash memory cache into separate read and write regions taking into consideration the fact that read and write costs are different in flash memory [11]. Chen et al. propose Hystor that integrates low-cost HDDs and high-speed SSDs [4]. To make better use of SSDs, Hystor identifies critical data, such as metadata, keeping them in SSDs. Also, it uses SSDs as a write-back buffer to achieve better write performance. Pritchett and Thottethodi observe that reference patterns are highly skewed and propose a highly-selective caching scheme for SSD cache [26]. These studies try to reduce expensive data allocation and write operations in flash memory storage as writes are much more expensive than reads. They are similar to ours in that flash memory storage is being used as a cache in hybrid storage solutions and that some of them split the flash memory cache into separate regions. However, our work is unique in that it takes into account the trade-off between caching benefit and data update cost as determined by the OPS size.

The use of the flash memory cache with other objectives in mind have been suggested. As SSDs have lower energy consumption than HDDs, Lee et al. propose an SSD-based cache to save energy of RAID systems [18]. In this study, an SSD is used to keep recently referenced data as well as for write buffering. Similarly, to save energy, Chen et al. suggest a flash memory based cache for caching and prefetching data of HDDs [3]. Saxena et al. use flash memory as a paging device for the virtual memory subsystem [28] and Debnath et al. use it as a metadata store for their de-duplication system [5]. Combining SSDs and HDDs in the opposite direction has also been proposed. A serious concern of flash memory storage is its relatively short lifetime and, to extend SSD lifetime, Soundararajan et al. suggest a hybrid storage system called Griffin, which uses HDDs as a write cache [32]. Specifically, they use a log-structured HDD cache, periodically destaging data to SSDs so as to reduce write requests and, consequently, to increase the lifetime of SSDs.

There have been studies that concentrate on finding cost-effective ways to employ SSDs in systems. To satisfy high-performance requirements at a reasonable cost budget, Narayanan et al. look into whether replacing disk based storage with SSDs may be cost effective; they conclude that replacing disks with SSDs is not yet so [22]. Kim et al. suggest a hybrid system called HybridStore that combines both SSDs and HDDs [15]. The goal of this study is in finding the most cost-effective configuration of SSDs and HDDs.

Besides studies on flash memory caches, there are many buffer cache management schemes that use the idea of splitting caching space. Kim et al. present a buffer management scheme called Unified Buffer Management (UBM) that detects sequential and looping references and stores those blocks in separate regions in the buffer cache [13]. Park et al. propose CRAW-C (Clock for Read And Write considering Compressed file system) that allocates three memory areas for read, write, and compressed pages, respectively [24]. Shim et al. suggest an adaptive partitioning scheme for the DRAM buffer in SSDs. This scheme divides the DRAM buffer into the caching and mapping spaces, dynamically adjusting their sizes according to the workload characteristics [31]. This study is different from ours in that the notion of OPS is necessary for flash memory updates, while for DRAM, it is not.

3 Flash Memory Cache Cost Model

In this section, we present the cost models for SSDs and HDDs [35]. HDD reading and writing are characterized by seek time and rotational delay. Assume that C_{D_RPOS} and C_{D_WPOS} are sums of the average seek time and the average rotational delay for HDD reads and writes, respectively. Let us also assume that P is the data size in bytes and B is the bandwidth of the disk. Then, the data read and write cost of a HDD is derived as $C_{DR} = C_{D\_RPOS} + \frac{P}{B}$ and $C_{DW} = C_{D\_WPOS} + \frac{P}{B}$, respectively. (Detailed derivations are referred to Wang [35].)

Before moving on to the cost model of flash memory based SSDs, we give a short review of NAND flash memory and the workings of SSDs. NAND flash memory, which is the storage medium of SSDs, consists of a number of blocks and each block consists of a number of pages. Reads are done in page units and take constant time. Writes are also done in page units, but data can be written to a page only after the block containing the page becomes clean, that is, after it is erased. This is called the erase-before-write property. Due to this property, data update is usually done by relocating new data to a clean page of an already erased block and most flash memory storage devices employ a sophisticated software layer called the Flash Translation Layer (FTL) that relocates modified data to new locations. The FTL also provides the same HDD interface to SSD users. Various FTLs such as page mapping FTL [7, 34], block mapping FTL [12], and many hybrid mapping FTLs [10, 17, 19, 23] have been proposed. Among them, the page mapping FTL is used in many high-end commercial SSDs that are used in hybrid storage solutions. Hence, in this paper, we focus on the page mapping FTL. However, the methodology that follows may be used with block and hybrid mapping FTLs as well. The key difference would be in deriving garbage collection and page write cost models appropriate for these FTLs.

As previously mentioned, the FTL relocates modified

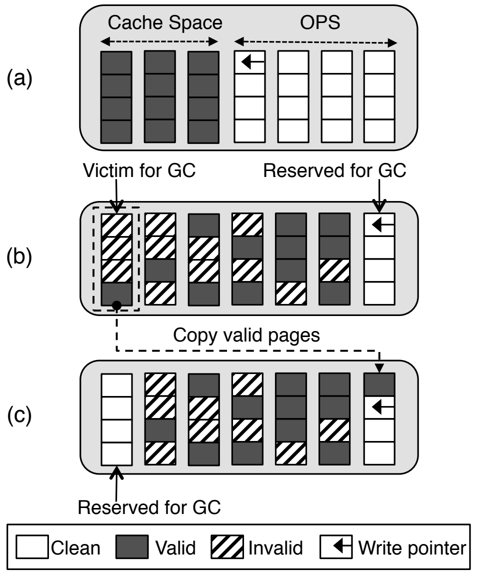


Figure 2: Garbage collection in flash memory storage

data to a clean page, and pages with old data become invalid. The FTL recycles blocks with invalid pages by performing garbage collection (GC) operations. For data updates and subsequent GCs, the FTL must always preserve some number of empty blocks. As data updates consume empty blocks, the FTL must produce more empty blocks by performing GCs that collect valid pages scattered in used blocks to an empty block, marking the used blocks as new empty blocks. The worst case and average GC costs are determined by the ratio of the initial OPS to the total storage space. It has been shown that the worst case and average GC costs become lower as more over-provisioned blocks are reserved [9].

If we assume that the FTL selects the block with the minimum number of valid pages for a GC operation, then the worst case GC occurs when all valid (or invalid) pages are evenly distributed to all flash memory blocks except for an empty block that is preserved for GC operations. For now, let us assume that u is the worst case utilization determined from the initial number of over-provisioned blocks and data blocks. Then, in Fig. 2(a), where there are 3 data blocks containing cached data and 4 initial over-provisioned blocks, the worst case u is calculated as $3/(3+4-1)$. (We subtract 1 because the FTL must preserve one empty block for GC as marked by the arrow in Fig. 2(b).) From u, the maximum number of valid pages in the block selected for GC can be derived as $\lceil u \cdot N_P \rceil$, where N_P is the number of pages in a block.

Then, the worst case GC cost for a given utilization u can be calculated from the following equation, where N_P is the number of pages in a block, C_E is the erase cost (time) of a flash memory block, and C_{CP} is the page copy cost (time). (We assume that the copyback operation is being used. For flash memory chips that do not support copyback, C_{CP} may be expanded to a sequence of read, C_{PR} , and write, C_{PROG}, operations.)

$$C_{GC}(u) = \lceil u \cdot N_P \rceil \cdot C_{CP} + C_E \tag{1}$$

That is, as seen in Fig. 2(b) and (c), a GC operation erases an empty block with cost C_E and copies all valid pages from the block selected for GC to the erased empty block with cost $\lceil u \cdot N_P \rceil \cdot C_{CP}$. Then, the garbage-collected block becomes an empty block that may be used for the next GC. The remaining clean pages in the previously empty block are used for subsequent write requests. If all those clean pages are consumed, then another GC operation will be performed.

After GC, in the worst case, there are $\lfloor (1-u) \cdot N_P \rfloor$ clean pages in what was previously an empty block (for example, the right-most block in Fig. 2(c)) and write requests of that number can be served in the block. Let us assume that C_{PROG} is the page program time (cost) of flash memory. (Note that "page program" and "page write" are used interchangeably in the paper.) By dividing GC cost and adding it to each write request, we can derive, $C_{PW}(u)$, the page write cost for worst case utilization u as follows.

$$C_{PW}(u) = \frac{C_{GC}(u)}{\lfloor (1-u) \cdot N_P \rfloor} + C_{PROG} \tag{2}$$

Equation 2 is the worst case page update cost of flash memory storage assuming valid data (or invalid data) are evenly distributed among all the blocks. Typically, however, the number of valid pages in a block will vary. For example, the block marked "Victim for GC" in Fig. 2(b) has a smaller number of valid pages than the other blocks. Therefore, in cases where the FTL selects a block with a small number of valid pages for the GC operation, then utilization of the garbage-collected block, u', would be lower than the worst case utilization, u. Previous LFS and flash memory studies derived and used the following relation between u' and u [17, 20, 35].

$$u = \frac{u'-1}{ln\, u'}$$

Let $U(u)$ be the function that translates u to u'. (In our implementation, we use a table that translates u to u'.) Then the average page update cost can be derived by applying $U(u)$ for u in Equation 1 and 2 leading to Equation 3 and 4.

$$C_{GC}(u) = U(u) \cdot N_P \cdot C_{CP} + C_E \tag{3}$$

$$C_{PW}(u) = \frac{C_{GC}(u)}{(1-U(u)) \cdot N_P} + C_{PROG} \tag{4}$$

4 Hybrid Storage Cost Model

In the previous section, the garbage collection and page update cost of flash memory storage was derived. In this section, we derive the cost models for hybrid storage systems, which consist of a flash memory cache and a HDD. Specifically, the cost models determine the optimal size of the caching space and OPS minimizing the overall data access cost of the hybrid storage system. In our derivation of the cost models, we first derive the read cache cost model and then, derive the read/write cache cost model used to determine the read cache size, write cache size and OPS size. Our models assume that the cache management layer can measure the hit and miss rates of read/write caches as well as the number of I/O requests. These values can be easily measured in real environments.

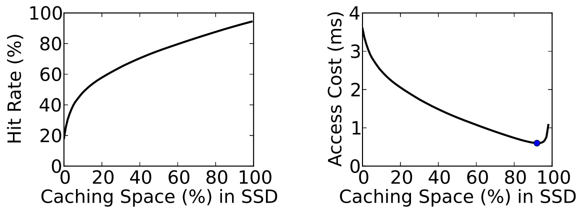
4.1 Read cache cost model

On a read request the storage examines whether the requested data is in the flash memory cache. If it is, the storage reads it and transfers it to the host system. If it is not in the cache, the system reads it from the HDD, stores it in the flash memory cache and transfers it to the host system. If the flash memory cache is already full with data (as will be the case in steady state), it must invalidate the least valuable data in the cache to make room for the new data. We use the LRU (Least Recently Used) replacement policy to select the least valuable data. In the case of read caching, the selected data need only be invalidated, which can be done essentially for free. (We discuss the issue of accommodating other replacement policies in Section 5.)

Let us assume that $H_R(u)$ is the cache read hit rate for a given cache size, which is determined by the worst case utilization u, as we will see later. With rate $H_R(u)$, the system reads the requested data from the cache with cost C_{PR}, the page read operation cost (time) of flash memory, and transfers it to the host system. With rate $1 - H_R(u)$, the system reads data from disk with cost C_{DR} and, after invalidating the least valuable data selected by the cache replacement policy, stores it in the flash memory cache with cost $C_{PW}(u)$, which is the cost of writing new data to cache including the possible garbage collection cost. Then, C_{HR}, the read cost of the hybrid storage system with a read cache, is as follows.

$$\begin{aligned} C_{HR}(u) = H_R(u) \cdot C_{PR} + \\ (1 - H_R(u)) \cdot (C_{DR} + C_{PW}(u)) \end{aligned} \tag{5}$$

Let us now take the flash memory cache size into consideration. For a given flash memory cache size, S_F, the read cache size, S_R and the OPS size S_{OPS} can be



(a) Read hit rate (b) Read cost

Figure 3: (a) Read hit rate curve generated using the numpy.random.zipf Python function (Zipfian distribution with $\alpha = 1.2$ and range $= 120\%$) and (b) the hybrid storage read cost graph for this particular hit rate curve, with optimal point at 92%.

approximated from u such that $S_{OPS} \approx (1-u) \cdot S_F$ and $S_R \approx u \cdot S_F$. These sizes are approximated values as they do not take into account the empty block reserved for GC. (Recall the empty block in Fig. 2.) Though calculating the exact size is possible by considering the empty block, we choose to use these approximations as these are simpler, and their influence is negligible relative to the overall performance estimation.

Let us now take an example. Assume that we have a hit rate curve $H_R(u)$ for read requests as shown in Fig. 3(a), where the x-axis is the cache size and the y-axis is the hit rate. Then, with Equation 5, we can redraw the hit rate curve with u on the x-axis, and consequently, the access cost graph of the hybrid storage system becomes Fig. 3(b). The graph shows that the overall access cost becomes lower as u increases until u reaches 92%, where the access cost becomes minimal. Beyond this point, the access cost suddenly increases, because even though the caching benefit is still high the data update cost soars as the OPS shrinks. Once we find u with minimum cost, the read cache size and OPS size can be found from $S_{OPS} \approx (1-u) \cdot S_F$ and $S_R \approx u \cdot S_F$.

4.2 Read and write cache cost model

Previous studies have shown that due to their difference in costs, separating read and write requests in flash memory storage has a significant effect on performance [11]. Hence, we now incorporate write cost to the model by dividing the flash caching space into two areas, namely a write cache and a read cache. The read cache, whose cost model was derived in the previous subsection, contains data that has recently been read but never written back while the write cache keeps data that has recently been written, but not yet destaged. Therefore, data in the write cache are dirty and they must be written to the HDD when evicted from the cache. When a write is requested to data in the read cache, we regard it as a write miss. In this case, we invalidate the data in the read cache and write the new data in the write cache. We consider the case of reading data in the write cache later.

In the following cost model derivation, we assume write-back policy for the write cache. This choice is more efficient than the write-through policy without any loss in consistency as the flash cache is also non-volatile. If the write-through policy must be used, our model needs to be modified to reflect the additional write to HDD that is incurred for each write to the flash cache. This will result in a far less efficient hybrid storage system.

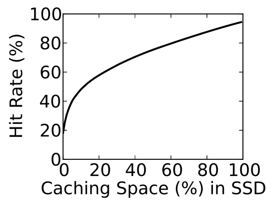
There can be three types of requests to the flash write cache. The first is a write hit request, which is a write request to existing data in the write cache. In this case, the old data becomes invalidated and the new data is written to the write cache with cost $C_{PW}(u)$. The second is a write miss request, which is a write request to data that does not exist in the write cache. In this case, the cache replacement policy selects victim data that should be read from the write cache and destaged to the HDD with cost $C_{PR} + C_{DW}$ to make room for the newly request data. (Note we are assuming the system is in steady state.) After evicting the data, the hybrid storage system writes the new data to the write cache with cost $C_{PW}(u)$. The last type of request is a read hit request, which is a read request to existing (and possibly dirty) data in the write cache. This happens when a read request is to data that is already in the write cache. In this case, the request can be satisfied with cost C_{PR}, that is, the flash memory page read cost. Note that there is no read miss request to the write cache because read requests to data not in cache are handled by the read cache.

Now we introduce a parameter r, which is the read cache size ratio within the caching space, where $0 \leq r \leq 1$. Naturally, $1-r$ is the ratio of the write cache size. If r is 1, all caching space is used as a read cache and, if it is 0, all caching space is used as a write cache. Let S_C denote the total size of the caching space. Then, we can calculate the read cache size, S_R, and write cache size, S_W, from S_C such that $S_R = S_C \cdot r$ and $S_W = S_C \cdot (1-r)$. Note that S_C is calculated from u such that $S_C \approx u \cdot S_F$. Then, S_R and S_W are determined by u and r.

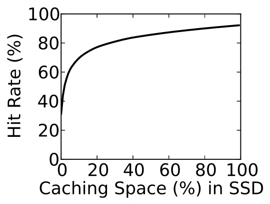
Let us assume that the cache management layer can measure the read hit rates of the read cache and draw $H_R(u,r)$, the read cache hit rate curve, which now has two parameters u and r. (We will show that the hit rate curve can be obtained by using ghost buffers in the next section.) Then, the read cost of the hybrid storage system is now modified as follows.

$$C_{HR}(u,r) = (1 - H_R(u,r)) \cdot (C_{DR} + C_{PW}(u)) \\ + H_R(u,r) \cdot C_{PR}$$

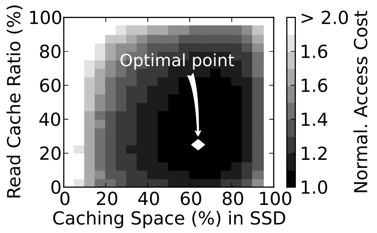
Let us also assume that we can measure the write hit, the write miss, and the read hit rates of the write cache



(a) Read hit rate



(b) Write hit rate



(c) Expected access cost

Figure 4: (a) Read and (b) write hit rate curves generated using the numpy.random.zipf Python function ((a) Zipfian distribution with $\alpha = 1.2$ and range = 120%, (b) Zipfian distribution with $\alpha = 1.4$ and range = 220%) and (c) the hybrid storage access cost graph for these hit rate curves.

and draw the hit rate curves. For the moment, let us regard the read hit in the write cache as being part of the write hit. Assume that $H_W(u,r)$ is the write cache hit rate for a given write cache size, and it has two parameters that determine the cache size. Then, with rate $H_W(u,r)$, a write request finds its data in the write cache, and the cost of this action is $H_W(u,r) \cdot C_{PW}(u)$. Otherwise, with rate of $1 - H_W(u,r)$, the write request does not find data in the write cache. Servicing this request requires reading and evicting existing data and writing new data to the write cache. Hence, the cost is $(1 - H_W(u,r)) \cdot (C_{PR} + C_{DW} + C_{PW}(u))$. In summary, the write cost of the hybrid storage system can be given as follows.

$$\begin{aligned} C_{HW}(u,r) = (1 - H_W(u,r)) & \\ \cdot (C_{PR} + C_{DW} + C_{PW}(u)) & \\ + H_W(u,r) \cdot C_{PW}(u) & \end{aligned}$$

Now let us consider the read hit case within the write cache. Although it is possible to maintain separate read hit and write hit curves for the write cache, this makes the cost model more complex without much benefits, especially in terms of implementation. Therefore, we devise a simple approximation method for incorporating the read hit case in the write cache. Assume that h' is the read hit rate in the write cache. (Then, naturally, $1 - h'$ is the write hit rate in the write cache.) Then, with rate h', the read hit is satisfied with cost C_{PR} and with rate $1 - h'$, the write hit is satisfied with cost $C_{PW}(u)$. Now we can calculate the average cost for both read hit and write hit such that $C_{WH} = (1 - h') \cdot C_{PW}(u) + h' \cdot C_{PR}$. By assuming $H_W(u,r)$ is the hit rate including both read and write hits, the write cost of the hybrid storage system now can be given as follows.

$$\begin{aligned} C_{HW}(u,r) = (1 - H_W(u,r)) & \\ \cdot (C_{PR} + C_{DW} + C_{PW}(u)) & \\ + H_W(u,r) \cdot C_{WH} & \end{aligned}$$

Now, let IO_R and IO_W, respectively, be the rate served in the read and write caches among all requests. For example, of a total of 100 requests, if 70 requests are served in the read cache and 30 requests are served in the write cache, then IO_R is 0.7 and IO_W is 0.3. Then we can derive, $C_{HY}(u,r)$, the overall access cost of the hybrid storage system that has separate read and write caches and OPS as follows.

$$\begin{aligned} C_{HY}(u,r) = C_{HR}(u,r) \cdot IO_R + & \\ C_{HW}(u,r) \cdot IO_W & \qquad (6) \end{aligned}$$

Let us take an example. Assume that, at a certain time, the hybrid storage system finds IO_R, IO_W, h' to be 0.2, 0.8, and 0.2, respectively, and the read and write hit rate curves are estimated as shown in Fig. 4(a) and (b). In the graph, both read and write hit rates increase as caches become larger but slowly saturate beyond some point. As the read and write cache sizes are determined by u and r, we can obtain the read and write cache hit rates for given u and r values from the hit rate curves. Then, with the cost model of Equation 6, we can draw the overall access cost graph of the system as in Fig. 4(c). In the graph, the x-axis is u and the y-axis is r. These two parameters determine the read and write cache sizes as well as the OPS size. In Fig. 4(c), darker shades reflect lower access cost and we pinpoint the lowest access cost with the diamond mark pointed to by the arrow.

Specifically, the minimum overall access cost of the hybrid storage system is when u is 0.64 and r is 0.25 for this particular configuration. For a 4GB flash memory cache, this translates to the read cache size of 0.64GB, the write cache size of 1.92GB, and an OPS size of 1.44GB.

5 Implementation Issues of Flash Cache Layer

In this section, we describe some implementation issues related to our flash memory cache management scheme, which we refer to as OP-FCL (Optimal Partitioning of Flash Cache Layer). Fig. 5(a) shows the overall structure of the hybrid storage system that we envision. The storage system has a HDD serving as main storage and an SSD, which we also refer to as the flash cache layer (FCL), that is used as a non-volatile cache keeping recently read/written data as previous studies have done [4, 11, 15]. As is common on SSDs, it has DRAM for buffering I/O data and storing data structures used by the SSD. The space at the flash cache layer is divided into three regions: the read cache area, the write cache area, and the over-provisioned space (OPS) as shown in Fig. 5(b). OP-FCL measures the read and write cache hit and miss rates and the I/O rates. Then, it periodically calculates the optimal size of these cache spaces and progressively adjusts their sizes during the next period.

To accurately simulate the operations and measure the costs of the hybrid storage system, we use DiskSim [2] to emulate the HDD and DiskSim's MSR SSD extension [1] to emulate the SSD. Specifically, the simulator mimics the behaviour of Maxtor's Atlas 10K IV disk whose average read and write latency is $4.4ms$ and $4.9ms$, respectively, with transfer speed of $72MB/s$. Also, the SSD simulator emulates SLC NAND flash memory chip operations, and it takes $25us$ to read a page, $200us$ to write a page, $1.5ms$ to erase a block, and $100us$ to transfer data to/from a page of flash memory through the bus. The page and block unit size is 4KB and 256KB, respectively, and the flash cache layer manages data in 4KB units.

In the simulator, we modified the SSD management modules and implemented additional features as well as the OP-FCL. OP-FCL consists of several components, namely, the *Page Replacer*, *Sequential I/O Detector*, *Workload Tracker*, *Partition Resizer*, and *Mapping Manager*.

The Page Replacer has two LRU lists, one each for the read and write caches, and maintains LRU ordering of data in the caches. In steady state when the cache is full, the LRU data is evicted from the cache to accommodate newly arriving data. For the read cache, cache eviction simply means that the data is invalidated, while for write cache, it means that data must be destaged, incurring a flash cache layer read and a disk write operation. In the actual implementation, the Page Replacer destages several dirty data altogether to minimize seek distance by applying the elevator disk scheduling algorithm. However, we do not consider group destaging in our cost model as it has only minimal overall impact. This is because the number of data destaged as a group is relatively small compared to the total number of data in the write cache.

Previous studies have taken notice of the impact of sequential references on cache pollution and thus, have tried to detect and treat them separately [13]. The Sequential I/O Detector monitors the reference pattern and detects sequential references. In our current implementation, consecutive I/O requests greater than $128KB$ are regarded as sequential references, and those requests bypass the flash cache layer and are sent directly to disk to avoid cache pollution.

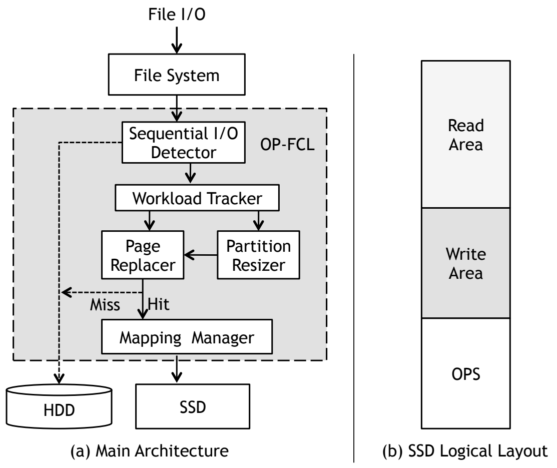


Figure 5: OP-FCL architecture

Besides the Page Replacer that manages the cached data, the Workload Tracker maintains LRU lists of ghost buffers to simultaneously measure hit rates of various cache sizes, following the method proposed by Patterson et al. [25]. Ghost buffers maintain only logical addresses, not the actual data and, thus, memory overhead is minimal requiring roughly 1% of the total flash memory cache. Part of the ghost buffer represents data in cache and others represent data that have already been evicted from the cache. Keeping information of evicted data in the ghost buffer makes it possible to measure the hit rate of a cache larger than the actual cache size. To simulate various cache sizes simultaneously, we use N-segmented ghost buffers. In other words, we divide the ghost buffer into N-segments corresponding to N cache sizes and thus, hit rates of N cache sizes can be obtained by combining the hit rates of the segments. From the hit rates of N cache sizes, we obtain the read/write hit rate curves by interpolating the missing cache sizes.

Note that though we use the LRU cache replacement policy for this study, our model can accommodate any replacement policy so long as they can be implemented in the flash cache and the ghost buffer management layers. Different replacement policies will generate different read/write hit rate curves and, in the end, affect the results. However, a replacement policy only affects the read/write hit rate curves, and thus, our overall cost model is not affected.

These hit rate curves are obtained per period. In the current implementation, a period is the logical time to process 65536 (2^{16}) read and write requests. When the period ends, new hit rate curves are generated, while a

Algorithm 1 Optimal Partitioning Algorithm

```
1: procedure OPTIMAL_PARTITIONING
2:     $step \leftarrow segment\_size / total\_cache\_size$
3:     INIT_PARMS($op\_cost$, $op\_u$, $op\_r$)
4:     for $u \leftarrow step$; $u < 1.0$; $u \leftarrow u + step$ do
5:         for $r \leftarrow 0.0$; $r \leq 1.0$; $r \leftarrow r + step$ do
6:             $cur\_cost \leftarrow C_{HY}(u, r)$                ▷ Call Eq. 6
7:             if $cur\_cost < op\_cost$ then
8:                 $op\_cost \leftarrow cur\_cost$
9:                 $op\_u \leftarrow u$, $op\_r \leftarrow r$
10:            end if
11:        end for
12:    end for
13:    ADJUST_CACHE_SIZE($op\_u$, $op\_r$)
14: end procedure
```

new period starts. Then, with the hit rate curves generated by the Workload Tracker in the previous period, the Partition Resizer gradually adjusts the sizes of the three spaces, that is, the read and write cache space and the OPS for the next period. To make the adjustment, the Partition Resizer determines the optimal *u* and *r* as described in Section 4, and those optimal values in turn decide the optimal size of the three spaces.

To obtain the optimal *u* and *r*, we devise an iterative algorithm presented in Algorithm 1. Starting from *u*=step, the outer loop iterates the inner loop increasing *u* in 'step' increments while *u* is less than 1.0. The two extreme configurations that we do not consider are where OPS is 0% and 100%. These are unrealistic configurations as OPS must be greater than 0% to perform garbage collection, while OPS being 100% would mean that there is no space to cache data. The inner loop starting from *r*=0 iterates, calculating the access cost of the hybrid storage system as derived in Equation 6, while increasing *r* in 'step' increments until *r* becomes greater or equal to 1.0. The 'step' value can be calculated as the segment size divided by the total cache size, as shown in the second line of Algorithm 1. The nested loop iterates $N \times M$ times to calculate the costs, where N is the outer loop count, 1/step-1, and M is the inner loop count, 1/step+1. A single cost calculation consists of 10 ADD, 4 SUB, 11 MUL, and 4 DIV operations. Finer 'step' values may be used resulting in finer *u* and *r* values, but with increased cost calculation overhead. However, computational overhead for executing this algorithm is quite small because they run once every period and the calculations are just simple arithmetic operations.

Once the optimal *u* and *r* and, in turn, the optimal sizes are determined, the Partition Resizer starts to progressively adjust the sizes of the three spaces. To increase OPS size, it gradually evicts data in the read or write caches. To increase cache space, that is, decrease OPS, GC is performed to produce empty blocks. These empty blocks are then used by the read and/or write caches.

The key role of our Mapping Manager is translating the logical address to a physical location in the flash cache layer. For this purpose, it maintains a mapping table that keeps the translation information. In our implementation, we keep the mapping information at the last page of each block. As we consider flash memory blocks with 64 pages, the overhead is roughly 1.6%. Moreover, we implement a crash recovery mechanism similar to that of LFS [27]. If a power failure occurs, it searches for the most up-to-date checkpoint and goes through a recovery procedure to return to the checkpoint state.

6 Performance Evaluation

In this section, we evaluate OP-FCL. For comparison, we also implement two other schemes. The first is the Fixed Partition-Flash Cache Layer (FP-FCL) scheme. This is the simplest scheme where the read and write cache is not distinguished, but unified as a single cache. The OPS is available with a fixed size. This scheme is used to mimic a typical SSD of today that may serve as a cache in a hybrid storage system. Normally, the SSD would not distinguish read and write spaces and it would have some OPS, whose size would be unknown. We evaluate this scheme as we vary the percentage of the caching space set aside for the (unified) cache. The best of these results will represent the most optimistic situation in real life deployment.

The other scheme is the Read and Write-Flash Cache Layer (RW-FCL) scheme. This scheme is in line with the observation made by Kgil et al. [11] in that read and write caches are distinguished. This scheme, however, goes a step further in that while the sum of the two cache sizes remain constant, the size between the two are dynamically adjusted for best performance according to the cost models described in Section 4. For this scheme, the OPS size would also be fixed as the total read and write cache size is fixed. We evaluate this scheme as we vary the percentage of the caching space set aside for the combined read and write cache. Initial, all three schemes start with an empty data cache. For OP-FCL, the initial OPS size is set to 5% of the total flash memory size.

The experiments are conducted using two sets of traces. We categorize them based on the size of requests. The first one, 'Small Scale', are workloads that request less than 100GBs of total data. The other set, 'Large Scale', are workloads with over 100GBs of data requests. Details of the characteristics of these workloads are in Table 1.

The first two subsections discuss the performance aspects of the two class of workloads. Then, in the next

Type	Workload	Working Set Size (GB)			Avg. Req. Size (KB)		Request Amount (GB)		Read Ratio
		Total	Read	Write	Read	Write	Read	Write	
Small Scale	Financial [33]	3.8	1.2	3.6	5.7	7.2	6.9	28.8	0.19
	Home [6]	17.2	13.5	5.0	22.2	3.9	15.3	66.8	0.18
	Search Engine [33]	5.4	5.4	0.1	15.1	8.6	15.6	0.001	0.99
Large Scale	Exchange [22]	79.35	74.12	23.29	9.89	12.4	114.36	131.69	0.46
	MSN [22]	37.98	30.93	23.03	11.48	11.12	107.23	74.01	0.59

Table 1: Characteristics of I/O workload traces

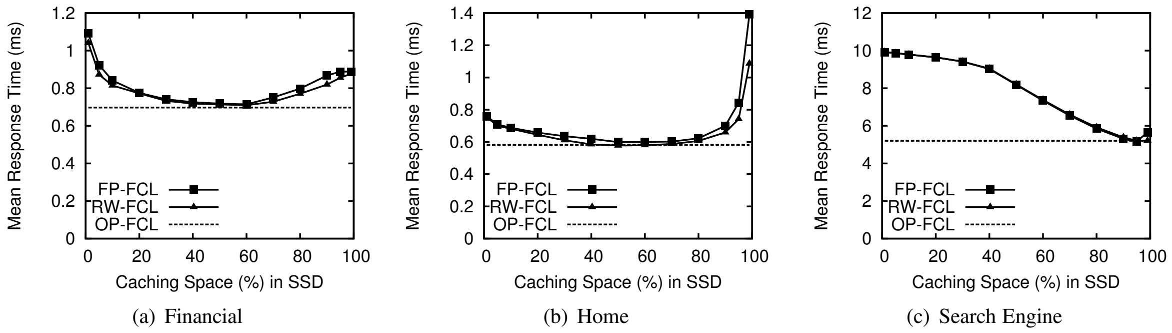


Figure 6: Mean response time of hybrid storage

Type	Description	Config. 1	Config. 2
OP-FCL	N_P	64	
	C_{PROG}	300us	
	C_{PR}	125us	
	C_{CP}	225us	
	C_E	1.5ms	
	C_{D_RPOS}	4.5ms	
	C_{D_WPOS}	4.9ms	
	B	72MB/s	
	P	4KB	
	segment_size	256MB	
SSD	Total Capacity	4GB	16GB
	No. of Packages	1	4
	Blocks Per Package	16384	
	Planes Per Package	1	
	Cleaning Policy	Greedy	
	GC Threshold	1%	
	Copyback	On	
HDD	Model	Maxtor Atlas 10K IV	
	No. of Disks	1	3

Table 2: Configuration of Hybrid Storage System

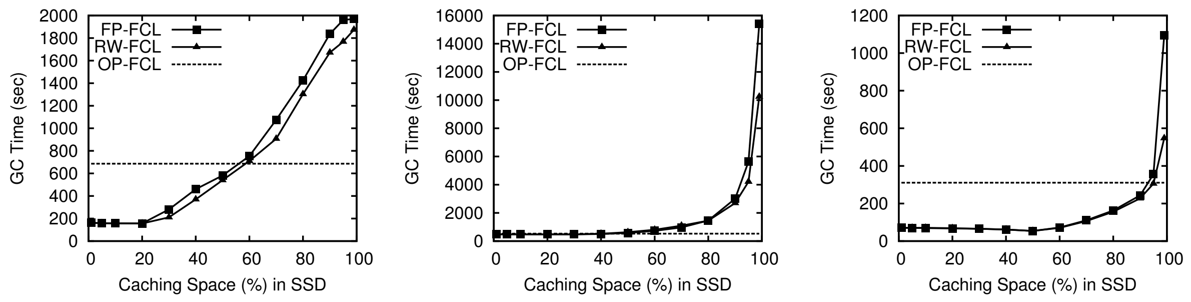
subsection, we present the effect of OP-FCL on the lifetime of SSDs. In the final subsection, we present a sensitivity analysis of two parameters that needs to be determined for our model.

6.1 Small scale workloads

The experimental setting is as given in Fig. 5 described in Section 5. The specific configuration of the HDD and SSD used in these experiments is shown in Table 2 denoted as 'Config. 1'. All other parameters not explicitly mentioned are set to default values. We assume a single SSD is employed as the flash memory cache and a single HDD as the main storage. This configuration is similar to that of a real hybrid drive [30].

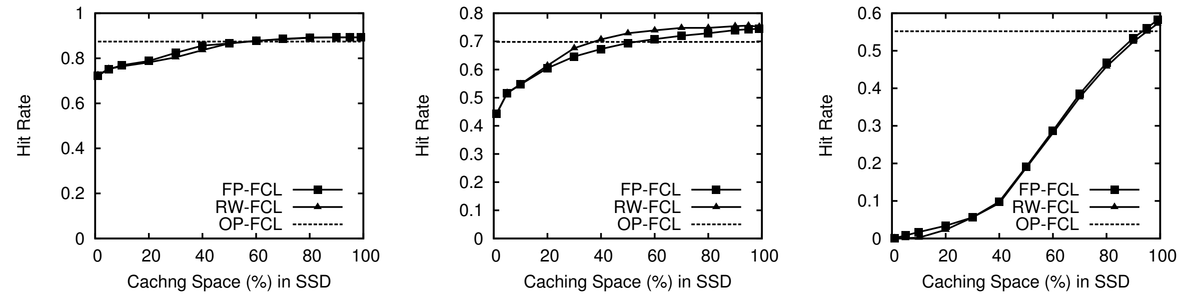
For small scale workloads we use three traces, namely, Financial, Home, and Search Engine that have been used in numerous previous studies [7, 11, 15, 16, 17]. The Financial trace is a random write intensive I/O workload obtained from an OLTP application running at a financial institutions [33]. The Home trace is a random write intensive I/O workload obtained from an NFS server that keeps home directories of researchers whose activities are development, testing, and plotting [6]. The Search Engine trace is a random read intensive I/O workload obtained from a web search engine [33]. The Search Engine trace is unique in that 99% of the requests are reads while only 1% are writes.

Fig. 6 shows the results of the cache partitioning schemes, where the measure is the response time of the hybrid storage system. The *x*-axis here denotes the ratio of caching space (unified or read and write combined) for the FP-FCL and RW-FCL schemes. For example, 60 in the *x*-axis means that 60% of the flash memory capacity is used as caching space and 40% is used as OPS. The *y*-axis denotes the average response time of the read and write requests. In the figure, the response times of FP-FCL and RW-FCL schemes vary according to the ratio of the caching space. In contrast, the response time of OP-FCL is drawn as a horizontal line because it reports



(a) Financial (b) Home (c) Search Engine

Figure 7: Cumulative garbage collection time



(a) Financial (b) Home (c) Search Engine

Figure 8: Hit rate

only one response time regardless of the ratio of caching space as it dynamically adjusts the three spaces according to the workload.

Let us first compare FP-FCL and RW-FCL in Fig. 6. In cases of the Financial and Home traces, we see that RW-FCL provides lower response time than FP-FCL. This is because RW-FCL is taking into account the different read and write costs in the flash memory cache layer. This result is in accord with previous studies that considered different read and write costs of flash memory [11]. However, in the case of the Search Engine trace, discriminating read and write requests has no effect because 99% of the requests are reads. Naturally, FP-FCL and RW-FCL show almost identical response times.

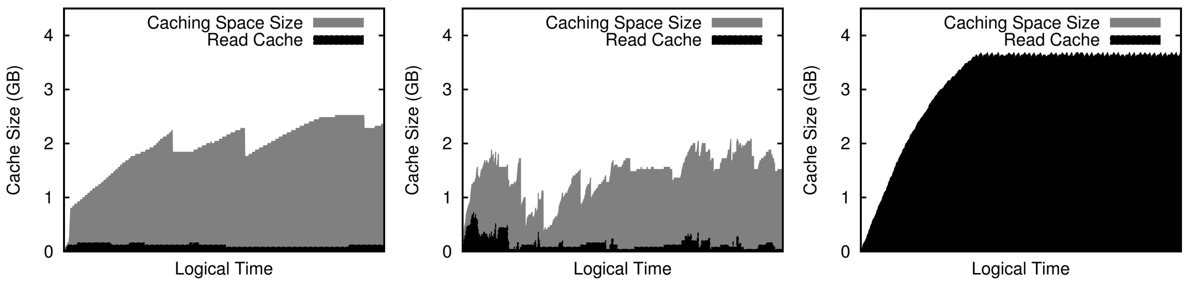
Now let us turn our focus to the relationship between the size of caching space (or OPS size) and the response time. In Fig. 6(a) and (b), we see that the response time decreases as the caching space increases (or OPS decreases) until it reaches the minimal point, and then increases beyond this point. Specifically, for FP-FCL and RW-FCL, the minimal point is at 60% for the Financial trace and at 50% for the Home trace for both schemes. In contrast, for the Search Engine trace, response time decreases continuously as the cache size increases and the optimal point is at 95%. The reason behind this is that the trace is dominated by read requests with rare modifications to the data. Thus, the optimal configuration for this trace is to keep as large a read cache as possible with only a small amount of OPS and write cache.

For the FP-FCL and RW-FCL schemes, the response time at the optimal point can be regarded as the off-line optimal value because it is obtained after exploring all possible configurations of the scheme. Let us now compare the response time of OP-FCL and the off-line optimal results of RW-FCL. In all traces, OP-FCL has almost the same response time as the off-line optimal value of RW-FCL. This shows that the cost model based dynamic adaptation technique of OP-FCL is efficient in determining the optimal OPS and the read and write cache sizes.

We now discuss the trade-off between garbage collection (GC) cost and the hit rate at the flash cache layer. Fig. 7 and 8 depict these results. In Fig. 7, we see that for all traces, GC cost increases, that is, performance degrades, continuously as caching space increases. The hit rate, on the other hand, increases, thus improving performance as caching space increases for all the traces as we can see in Fig. 8. For clear comparisons, we report the sum of the read and write hit rates for RW-FCL and OP-FCL. Note that both schemes measure read and write hit rates separately.

These results show the existence of two contradicting effects as caching space is increased, that is, increasing cache hit rate, which is a positive effect, and increasing GC cost, which is a negative effect. The interaction of these two contradicting effects leads to an optimal point where the overall access cost of the hybrid storage system becomes minimal.

To investigate how well OP-FCL adjusts the caching



(a) Financial (b) Home (c) Search Engine

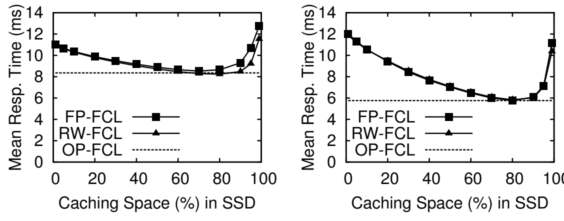
Figure 9: Dynamic size adjustment of read and write caches and OPS

space and OPS sizes, we continuously monitor their sizes as the experiments are conducted. Fig. 9 shows these results. In the figure, the x-axis denotes logical time that elapses upon each request and the y-axis denotes the total (read + write) caching space size and the read cache size. For the Financial and Home traces, we see that the caching space size increases and decreases repeatedly according to the reference pattern of each period as the cost models maneuver the caching space and OPS sizes. Notice that out of the 4GB of flash memory cache space, only 2 to 2.5GBs are being used for the Financial trace and less than half is used for the Home trace. Even though cache space is available, using less of it helps performance as keeping space to reduce garbage collection time is more beneficial. Note, though, that for the Search Engine trace, most of the 4GB are being allotted to the caching space, in particular, to the read cache. This is a natural consequence as reads are dominant, garbage collection rarely happens. Also note that it is taking some time for the system to stabilize to the optimal allocation setting.

6.2 Large scale workloads

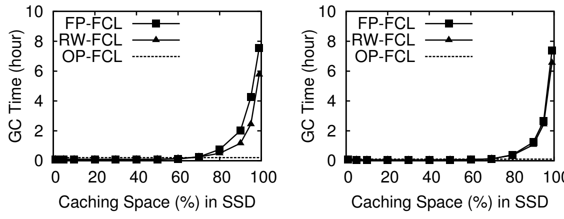
Our experimental setting for large scale workloads is as shown in Fig. 5 with the configuration summarized as 'Config. 2' in Table 2. In this configuration the SSD is 16GBs employing four packages of flash memory and the HDD consists of three 10K RPM drives.

To test our scheme for large scale enterprise workloads, we use the Exchange and MSN traces that have been used in previous studies [15, 21, 22]. The Exchange trace is a random I/O workload obtained from the Microsoft employee e-mail server [22]. This trace is composed of 9 volumes of which we select and use traces of volumes 2, 4, and 8, and each volume is assigned to each HDD. The MSN trace is extracted from 4 RAID-10 volumes on an MSN storage back-end file store [22]. We choose and use the traces of volumes 0, 1, and 4, each assigned to one HDD. The characteristics of the two traces are summarized in Table 1.



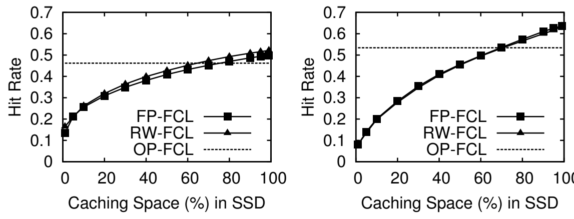
(a) Exchange (b) MSN

Figure 10: Response time of hybrid storage



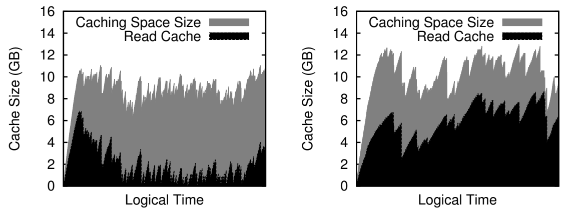
(a) Exchange (b) MSN

Figure 11: Cumulative garbage collection time



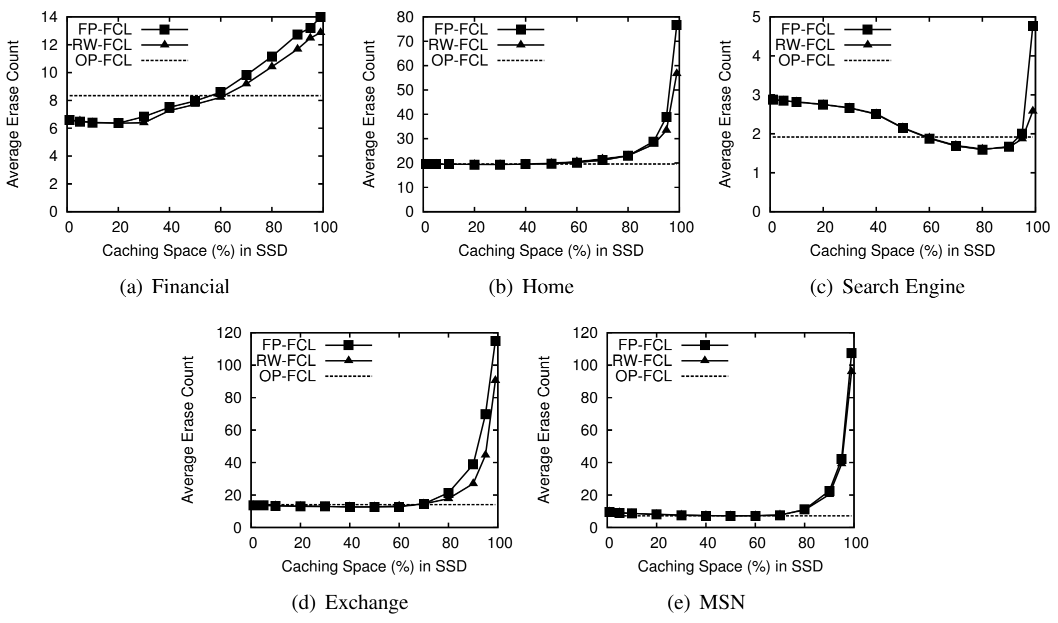
(a) Exchange (b) MSN

Figure 12: Hit rate



(a) Exchange (b) MSN

Figure 13: Dynamic size adjustment of read and write caches and OPS



(a) Financial (b) Home (c) Search Engine (d) Exchange (e) MSN

Figure 14: Average erase count of flash memory blocks

Fig. 10, which depicts the response time for the two large scale workloads, show similar trends that we observed with the small scale workloads, in that, as caching space increases, response time decreases to a minimal point, and then increases again. The response time of OP-FCL, which is shown as a horizontal line in the figure, is close to the smallest response times of FP-FCL and RW-FCL. From these results, we confirm again that a trade-off between GC cost and hit rate exists at the flash cache layer.

Specifically, for the Exchange trace shown in Fig. 10(a), the minimal point for FP-FCL is at 70%, while it is at 80% for RW-FCL. The reason behind this difference can be found in Fig. 11 and Fig. 12. Fig. 12(a) shows that RW-FCL has a higher hit rate than FP-FCL at cache size 80%. On the other hand, Fig. 11(a) shows that for cache size of 70% to 80% the GC cost increase is steeper for FP-FCL than for RW-FCL. These results imply that, for RW-FCL, the positive effect of caching more data is greater than the negative effect of increased GC cost at 80% cache size, and vice versa for FP-FCL. These differences in positive and negative effect relations for FP-FCL and RW-FCL result in different minimal points.

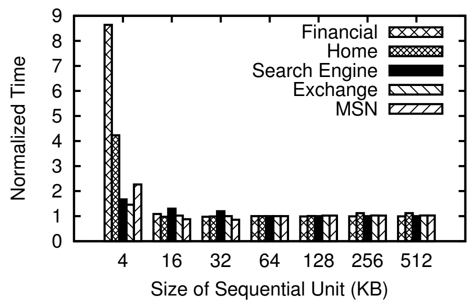
From the results of the MSN trace shown in Fig. 10(b), we observe that FP-FCL and RW-FCL have almost identical response times. This is because they have almost the same hit rate curves, which means that discriminating read and write requests has no performance benefit for the MSN trace. The minimal points for FP-FCL and RW-FCL are at cache size 80% for this trace.

As with the small scale workloads, Fig. 13 shows how OP-FCL adjusts the cache and OPS sizes according to the reference pattern for the large scale workloads. Initially, the cache size starts to increase as we start with an empty cache. Then, we see that the scheme stabilizes with OP-FCL dynamically adjusting the caching space and OPS sizes to their optimal values.

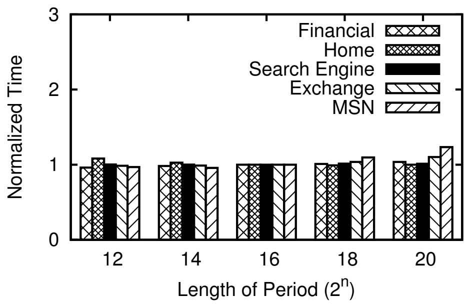
6.3 Effect on lifetime of SSDs

Now let us turn our attention to the effect of OP-FCL on the lifetime of SSDs. Generally, block erase count, which is affected by the wear-levelling technique used by the SSDs, directly corresponds to SSD lifetime. Therefore, we measure the average erase counts of flash memory blocks for all the workloads, and the results are shown in Fig. 14. With the exception of the Search Engine, we see that, for FP-FCL and RW-FCL, the average erase count is low when caching space is small. As caching space becomes larger, the average erase count increases only slightly until the caching space reaches around 70%. Beyond that point, the erase count increases sharply as OPS size becomes small and GC cost rises. In contrast, OP-FCL has a low average erase count drawn as a horizontal line in Fig. 14.

In contrast to the other traces, the average erase count for the Search Engine trace is rather unique. First, the overall average erase count is noticeably lower than that of the other traces. Also, instead of a sharp increase observed for the other traces, we first see a noticeable drop as the cache size approaches 80%, before a sharp increase. The reason behind this is that 99% of the Search Engine trace are read requests and the footprint is so



(a) Effect of sequential unit size



(b) Effect of period length

Figure 15: Sensitivity analysis of sequential unit size and period length on OP-FCL performance

huge that the cache hit rate continuously increases almost linearly with larger caches as shown in Fig. 8(c). This continuous increase in hit rate continuously reduces new writes resulting in reduced garbage collection, and then eventually to reduced block erases. Beyond the 80% point, block erases increase because GC cost increases sharply as the OPS becomes smaller.

6.4 Sensitivity analysis

In this subsection, we present the effect on the choice of the sequential unit size and the length of the period on the performance of OP-FCL. The results for all the workloads are reported relative to the parameter settings used for all the results presented in the previous subsections: the sequential unit size of 128 and period length of 2^{16}.

Recall that the sequential unit size determines the consecutive request size that the Sequential I/O Detector regards as being sequential, and that these requests are sent directly to the HDD. Fig. 15(a) show the effect of the sequential unit size. When the sequential unit size is 4 KB, OP-FCL performs very poorly. This is because too many requests are being considered to be sequential and are sent directly to the HDD. However, when the sequential unit size is between 16 KB $\sim$ 512 KB, OP-FCL shows similar performance showing that performance is relatively insensitive to the parameter of choice.

Fig. 15(b) shows the performance of OP-FCL as the length of the period is varied from 2^{12} to 2^{20} requests. Overall, the performance is stable. The Home trace performance deteriorates somewhat for periods of 2^{14} and below, with worse performance as the period shortens. The reason behind this is that the workload changes frequently as observed in Fig. 9. As a result, by the time OP-FCL adapts to the results of the previous period, the new adjustment becomes stale, resulting in performance reduction. We also see that performance is relatively consistent and best for periods between 2^{14} to 2^{16}. For periods beyond 2^{18}, OP-FCL performance deteriorates slightly. As the period increases to 2^{20}, performance of the Exchange and MSN traces start to degrade. This is because the change in the workload spans a relatively large range compared to those of small scale workloads as shown in Fig. 13. For this reason, OP-FCL of longer periods is not dynamic enough to reflect these workload changes effectively. Overall though, we find that for a relatively broad range of periods performance is consistent.

7 Conclusions

NAND flash memory based SSDs are being used as non-volatile caches in hybrid storage solutions. In flash based storage systems, there exists a trade-off between increasing the benefits of caching data and increasing the benefit of reducing the update cost as garbage collection cost is involved. To increase the former, caching space, which is cache space that holds normal data, must be increased, while to increase the latter, over-provisioned space (OPS) must be increased. In this paper, we showed that balancing the caching space and OPS sizes has a significant impact on the performance of hybrid storage solutions. For this balancing act, we derived cost models to determine the optimal caching space and OPS sizes, and proposed a scheme that dynamically adjusts sizes of these spaces. Through experiments we show that our dynamic scheme performs comparatively to the off-line optimal fixed partitioning scheme. We also show that the lifetime of SSDs may be extended considerably as the erase count at SSDs may be reduced.

Many studies on non-volatile cache have focussed on cache replacement and destaging policies. As a miss at the flash memory cache leads to HDD access, it is critical that misses be reduced. When misses do occur at the write cache, intelligent destaging should help ameliorate miss effects. Hence, we are currently focusing our efforts on developing better cache replacement and destaging policies, and in combining these policies with our cache partitioning scheme. Another direction of research that we are pursuing is managing the flash memory cache layer to tune SSDs to trade-off between performance and lifetime.

8 Acknowledgments

We would like to thank our shepherd Margo Seltzer and anonymous reviewers for their insight and suggestions for improvement. 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), by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (No. 2009-0085883), and by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(2010-0025282).

References

[1] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design Tradeoffs for SSD Performance. In *Proc. of USENIX ATC* (2008), pp. 57–70.

[2] BUCY, J. S., SCHINDLER, J., SCHLOSSER, S. W., AND GANGER, G. R. DiskSim 4.0. *http://www.pdl.cmu.edu/DiskSim/*.

[3] CHEN, F., JIANG, S., AND ZHANG, X. SmartSaver: Turning Flash Drive into a Disk Energy Saver for Mobile Computers. In *Proc. of ISLPED* (2006), pp. 412–417.

[4] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems. In *Proc. of ICS* (2011), pp. 22–32.

[5] DEBNATH, B., SENGUPTA, S., AND LI, J. ChunkStash: Speeding Up Inline Storage Deduplication using Flash Memory. In *Proc. of ATC* (2010).

[6] FIU TRACE REPOSITORY. *http://sylab.cs.fiu.edu/projects/iodedup*.

[7] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *Proc. of ASPLOS* (2009), pp. 229–240.

[8] HONG, S., AND SHIN, D. NAND Flash-Based Disk Cache Using SLC/MLC Combined Flash Memory. In *Proc. of SNAPI* (2010), pp. 21–30.

[9] HU, X.-Y., ELEFTHERIOU, E., HAAS, R., ILIADIS, I., AND PLETKA, R. Write Amplification Analysis in Flash-based Solid State Drives. In *Proc. of SYSTOR* (2009).

[10] KANG, J.-U., JO, H., KIM, J.-S., AND LEE, J. A Superblock-based Flash Translation Layer for NAND Flash Memory. In *Proc. of EMSOFT* (2006), pp. 161–170.

[11] KGIL, T., ROBERTS, D., AND MUDGE, T. Improving NAND Flash Based Disk Caches. In *Proc. of ISCA* (2008), pp. 327–338.

[12] KIM, J., KIM, J. M., NOH, S. H., MIN, S. L., AND CHO, Y. A Space-Efficient Flash Translation Layer for CompactFlash Systems. *IEEE Trans. on Consumer Electronics 48*, 2 (2002), 366–375.

[13] KIM, J. M., CHOI, J., KIM, J., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C. S. A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References. In *Proc. of OSDI* (2000).

[14] KIM, S.-H., JUNG, D., KIM, J.-S., AND MAENG, S. HeteroDrive: Reshaping the Storage Access Pattern of OLTP Workload Using SSD. In *Proc. of IWSSPS* (2009), pp. 13–17.

[15] KIM, Y., GUPTA, A., URGAONKAR, B., BERMAN, P., AND SIVASUBRAMANIAM, A. HybridStore: A Cost-Efficient, High-Performance Storage System Combining SSDs and HDDs. In *Proc. of MASCOTS* (2011), pp. 227–236.

[16] KOLLER, R., AND RANGASWAMI, R. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *Proc. of FAST* (2010).

[17] KWON, H., KIM, E., CHOI, J., LEE, D., AND NOH, S. H. Janus-FTL: Finding the Optimal Point on the Spectrum Between Page and Block Mapping Schemes. In *Proc. of EMSOFT* (2010), pp. 169–178.

[18] LEE, H. J., LEE, K. H., AND NOH, S. H. Augmenting RAID with an SSD for Energy Relief. In *Proc. of HotPower* (2008).

[19] LEE, S.-W., PARK, D.-J., CHUNG, T.-S., LEE, D.-H., PARK, S., AND SONG, H.-J. A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation. *ACM Trans. on Embedded Computer Systems 6*, 3 (2007).

[20] MENON, J. A Performance Comparison of RAID-5 and Log-Structured Arrays. In *Proc. of HPDC* (1995).

[21] NARAYANAN, D., DONNELLY, A., THERESKA, E., ELNIKETY, S., AND ROWSTRON, A. Everest: Scaling Down Peak Loads Through I/O Off-Loading. In *Proc. of OSDI* (2008), pp. 15–28.

[22] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *Proc. of EuroSys* (2009), pp. 145–158.

[23] PARK, C., CHEON, W., KANG, J., ROH, K., CHO, W., AND KIM, J.-S. A Reconfigurable FTL (Flash Translation Layer) Architecture for NAND Flash-Based Applications. *ACM Trans. on Embedded Computer Systems 7*, 4 (2008).

[24] PARK, J., LEE, H., HYUN, S., KOH, K., AND BAHN, H. A Cost-aware Page Replacement Algorithm for NAND Flash Based Mobile Embedded Systems. In *Proc. of EMSOFT* (2009), pp. 315–324.

[25] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed Prefetching and Caching. In *Proc. of SOSP* (1995), pp. 79–95.

[26] PRITCHETT, T., AND THOTTETHODI, M. SieveStore: A Highly-Selective, Ensemble-level Disk Cache for Cost-Performance. In *Proc. of ISCA* (2010), pp. 163–174.

[27] ROSENBLUM, M., AND OUSTERHOUT, J. K. The Design and Implementation of a Log-Structured File System. *ACM Trans. on Computer Systems 10*, 1 (1992), 26–52.

[28] SAXENA, M., AND SWIFT, M. M. FlashVM: Virtual Memory Management on Flash. In *Proc. of ATC* (2010).

[29] SCHINDLER, J., SHETE, S., AND SMITH, K. A. Improving throughput for small disk requests with proximal I/O. In *Proc. of FAST* (2011).

[30] SEAGATE MOMETUS ®XT. *http://www.seagate.com/www/en-us/products/laptops/laptop-hdd*.

[31] SHIM, H., SEO, B.-K., KIM, J.-S., AND MAENG, S. An Adaptive Partitioning Scheme for DRAM-based Cache in Solid State Drives. In *Proc. of MSST* (2010).

[32] SOUNDARARAJAN, G., PRABHAKARAN, V., BALAKRISHNAN, M., AND WOBBER, T. Extending SSD Lifetimes with Disk-Based Write Caches. In *Proc. of FAST* (2010).

[33] UMASS TRACE REPOSITORY. *http://traces.cs.umass.edu*.

[34] UNDERSTANDING THE FLASH TRANSLATION LAYER (FTL) SPECICATION. *Intel Corporation*, 1998.

[35] WANG, W., ZHAO, Y., AND BUNT, R. HyLog: A High Performance Approach to Managing Disk Layout. In *Proc. of FAST* (2004), pp. 145–158.