Multi-Resolution Image Store – A Case of Size-Tiered Object Storage Systems

Ming Chen
mchen@cs.stonybrook.edu
Department of Computer Science, Stony Brook University

Draft of 2012/12/15 02:13

Abstract

The storage subsystem has been the performance bottleneck in many server workloads mostly because of slow mechanical HDDs, the primary storage device. Faster storage devices such as Flash are emerging as a hope of solving the problem. However, a drop-in replacement approach is not feasible because of their high price, asymmetric performance characteristics, and short life time. This project presents a multi-tiered object storage system, which integrates good characteristics of different kinds of storage devices in speed, capacity, and price. We distribute objects among multiple devices based on not only their hotness but also their sizes, so that they can be stored efficiently in terms of performance and cost. We implemented the system based on LevelDB, and benchmarked the system using a two-tiered store including a Flash tier and a HDD tier. Our system presents up to 5.86× speedup of throughput over the single-tier store using HDD. Particularly, for a workload emulating Facebook's photo requests, we observe a speedup of 3.73× of the throughput using only a small Flash of 5.9% of the total storage capacity.

1 Introduction

In many scenarios of storage systems, the access patterns and the sizes of structured objects presents a property that matches the property of the storage hierarchy. For example, in an image storage system, the attributes and thumbnails of stored images are small in size but frequently accessed, which makes them fits well in small but fast and expensive top-tiered storage devices, such as NVRAM and Flash. Whereas the original images are big but less accessed, which makes them suitable for large but slow and inexpensive bottom-tiered storage devices, such as HDD and tape. Moreover, slow seeks can usually be amortized by fast sequential access followed.

For small objects, throughput is important in term of op/sec. They tend to be accessed randomly because of their small sizes and the implication that they are likely to be thumbnails, metadata and attributes. Top-tiered devices, e.g., NVRAM, exhibit great IOPS performance, and they allow storage to be used in finer granularity which causes less inner fragmentation as well. But for large objects, throughput is more important in terms of mb/sec. Their I/Os tend to be sequential as well. Bottom-tiered devices, e.g., HDD, are large in capacity and exhibit satisfactory throughput for sequential I/Os.

When objects present such a size-tiered property, a corresponding size-tiered storage system can provide good trade-off between cost and performance as it gets the best from different tiers of the storage hierarchy. Size-tiered storage system is also able to support the popular out-of-place update optimization, as in log-structured filesystems, by turning small in-place updates to revision logs as metadata and compacting them into large batched I/O to bottom tiers.

Since multimedia files are primarily accessed sequentially, it may not be necessary to provide for efficient random access to every large file [6]. Facebook researchers also argued that it may even be worthwhile to investigate not caching large objects in the memory at all, to increase overall cache hit rates [2]. We implemented a size-tiered object storage system optimized for multi-resolution images, named MRIS (Multi-Resolution Image Store). MRIS aims at storing large amount of images, as well as their metadata and smaller versions (such as thumbnails), efficiently. However, the strategies employed in MRIS can be applied to other storage and web-serving systems as well because there also exist salient size characteristics following power-law distributions [2].

The rest of the paper is organized as follows. Section 2 presents the background of Flash SSD, multi-resolution images, and key/value store. Section 3 describe our design and implementation. We evaluate the performance of our system in Section 4. We analyze related work in Section 5 and conclude our work in Section 6.

2 Background

MRIS is a key/value store designed for multi-resolution image workloads using a multi-tier architecture consisting of Flash SSD and magnetic HDD.

2.1 Flash SSD

Flash is a type of non-volatile memory. There are two main types of flash memory, which are named after the NAND and NOR logic gates [8]. NAND is the more popular one used in SSDs [22]. NAND flash chip is able to trap electrons between its gates. The absence of electrons corresponds to a logical 0. Otherwise, it corresponds to a logical 1. NAND can be furthered divided into SLC and MLC by the number of bits can be represented in a cell.

NAND Flash has asymmetric read and write performance. Read is fast and takes roughly 50μs for a MLC [22]. Write is 10-20 times slower than read. However, write is complicated in the sense that bits cannot be simply overwritten. Before writes, a block has to undergo an erase procedure which is 2-3 order of magnitude slower than read. Moreover, NAND Flash cell can endure only limit cycles of erasing. Therefore, Flash chips are often used for storage in the form of SSD, which also contains internal controller, processor and RAM. Algorithms including log-structured writing, wear-leveling, and garbage collection are implemented inside SSD to make Flash writes faster and endures longer.

2.2 Key/Value Store

As we implemented MRIS using LevelDB [13], we use it as an example to analyze key/value stroe. LevelDB is an open source key/value database engine developed by Google. LevelDB is log-structured and organizes data into Sorted String Table (SSTable). SSTable, introduced in Bigtable [5], is an immutable data structure containing a sequence of key/value pairs sorted by the key as shown in Figure 1. Besides key and value, there might be optional fields such as CRC, compression type etc. SSTable are mostly saved as files and each of them can contains data structures, such as bloomfilter, to facilitate key lookup. SSTable have counterpart in the memory called Memtable. The key/value pairs in Memtable are often kept in data structures easy for insert and lookup such as red/black tree and skiplist.

LevelDB, as well as most other key/value engines, use Log-Structured Merge Trees (LSM) [16] for internal storage. When key/value pairs are first added, they are inserted into Memtable. Once the size of the Memtable grows beyond a certain threshold, the whole Memtable is flushed out into a SSTable, and a new Memtable is created for insertion. When key/value pairs get changed, the new pairs are inserted without modifying the old pairs. When a key/value pair is deleted, a marker of the deletion is inserted by setting a flag inside the key called `KeyType`. This way key/value can provide large insertion throughput because data is written out using sequential I/Os, which have good performance on Hard Disk Drives (HDD).

To serve a key lookup, Memtable is queried firstly.

Figure 1: SSTable

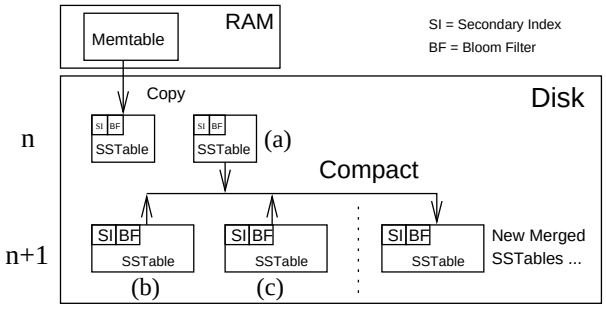


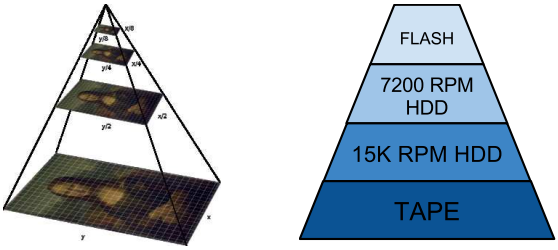
Figure 2: LevelDB Compaction

If not found in Memtable, the SSTables are queried in reverse chronological order. A naive implementation of such a lookup can be very slow because the whole database need be read and checked in the worst case. To make lookup fast, SSTable are organized into several layers with the size of each table increasing from the top layer to the bottom. Background jobs are launched periodically to sort and merge small SSTables into larger ones in the next layer. This is called compaction. Deleted pairs are also removed during compaction. Then a lookup iterates the SSTables layer by layer and returns once the key is found. Because SSTables are sorted by key, it enables fast lookup algorithm like binary search. There is also index for SSTables tells the key range covered by a particular SSTable so that it suffice just checking the SSTables whose key ranges cover the interested key. Inside each SSTable, we can have a bloomfilter to filter negative key lookup and a secondary index for faster search.

In LevelDB, there are two Memtables, once one is filled, the other one is used for further insertion. The filled one is flushed into a Memtable in background. Its compaction procedure is illustrated in Figure 2. One SSTable (a) at layer n is merged with the SSTables at layer $n+1$ that have overlapping keys with (a) into new SSTables at layer $n+1$.

2.3 Multi-Resolution Image

Multi-Resolution Images are different representations of the same image at different resolutions. A common form of multi-resolution images are images with different pixel resolutions, for instance, thumbnails at 100×100, small images at 600×800, and large images at 2048×3072. Multi-Resolution also describes a category of techniques used in image processing and analysis [15], including image pyramids (e.g., Laplacian pyramids), discrete wavelet transform, and wavelet image compression. Images are represented and stored with multi-resolution version not only because for fast preview but also for extracting different level of information from image content.



(a) Multi-Resolution Image (b) Storage Tiers

Figure 3: Multi-Resolution Image and Storage Tiers

As we can see from Figure 3, we see a match of the multi-resolution image and the storage tiers. The idea of MRIS is to matching the storage system for the objects stored in it. It also exploits the strength of different types of storage media.

3 Design and Implementation

A drawback of compaction is that pairs are read and written multiple times when they are gradually merged from the top layer to the bottom. Although compaction is scheduled in background, it still influences the overall system performance when the I/O traffic is heavy. Reads and writes of large pairs make compaction long and in turn introduce more negative effect on performance. Therefore, it is desirable to reduce the copying of large pairs. Because the log-structured nature of key/value store, once written, the pairs will never be updated. So it is reasonable to keep large values unmoved especially when the reduce of data copying can offset the cost of an extra disk seek.

As cloud computing is spreading widely, high throughput storage system is required because, in consolidated cloud environment, data are generated fast in great volume. It demands not only high storage throughput but also large capacity. It is a natural choice to compromise between cost and performance and save different data on different media. Some companies achieved this using big memory for hot data and storing not-so-hot data into second-level storage [20]. However, memory is expensive and volatile. New NVRAM storage media such as Flash SSD provides an alternative solution, which can be cheaper and safer.

Based on the idea of saving different data on different media, we modify LevelDB to save large values separately into a space called LargeSpace. LargeSpace is essentially a log and can be saved in media different from the SSTables. Upon a request of insertion, the value is appended into LargeSpace and an address is returned. The address is used for later retrieve of the value. To ease management, LargeSpace is splited into files called LargeBlocks. The threshold of split, called SplitThreshold, is a configurable parameter with a default value of 64MB. Although LargeSpace is split, we enforce that no

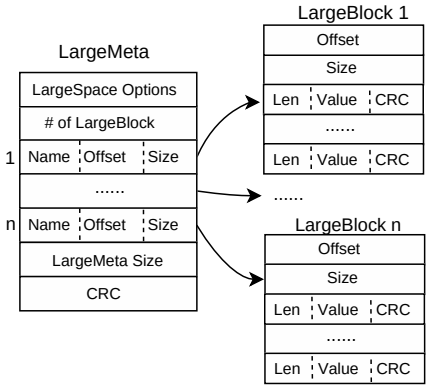


Figure 4: Large Space

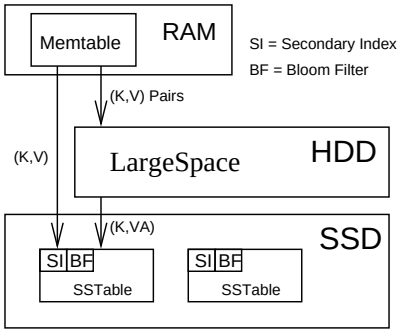


Figure 5: MRIS Insertion.

value will be splited among LargeBlocks. Therefore, it is possible that a LargeBlock is larger than the split threshold if there is a huge value in that LargeBlock. LargeBlocks are recorded in a metadata file called LargeMeta. The formats of LargeMeta and LargeBlock and their relationship is illustrated in Figure 4.

It is worth notice that LargeMeta is an immutable file. A new one is created every time it is updated. LargeMeta can be readily rebuilt using information from LargeBlocks. Many versions of LargeMeta are kept which provides a natural support for versioning. This makes the metadata resistant to hazardous situations such as system crash and file corruption.

A large key/value pair will be saved in LargeSpace when its value size is larger than a configurable threshold, named `SizeThreshold`. SizeThreshold provides a simple but effective knob to tradeoff between cost and performance when LargeSpace and SSTable are stored in different places. We plan to use more sophisticated algorithm to determine where to place a certain pair considering not only size but also hotness and access pattern.

When key/value pairs are firstly inserted into Memtable, they are treated equally. Large pairs in a Memtable are checked when the Memtable becomes full and is dumped into SSTable. Whereas small pairs go to SSTable without touch, the value of large pairs are firstly inserted into LargeSpace. For each large pair, a new pair is formed and inserted into SSTable. The key of the new

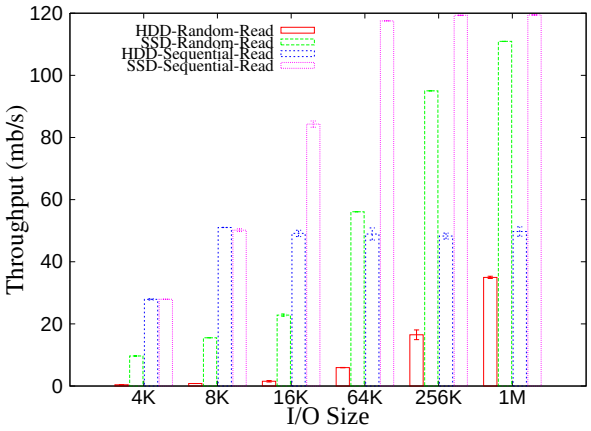


Figure 6: Read performance of SSD and HDD

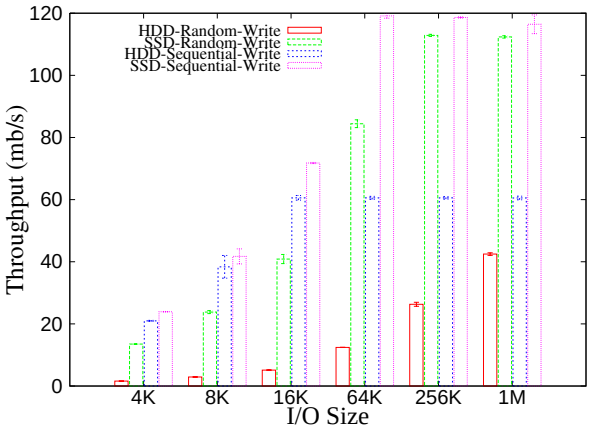


Figure 7: Write performance of SSD and HDD

pair is the same as the old key except its `KeyType` is changed to indicate that it represents a large pair. The value of the new pair contains the size of the original value, and its addresses in LargeSpace.

4 Evaluation

We have evaluated our system on a 64-bit Dell server with 1GB memory and a one-core Intel(R) Xeon(TM) CPU 2.80GH CPU. The OS, a Ubuntu server 8.04 with kernel version 3.2.9, is installed on a Maxtor 7L250S0 3.5-inch SATA HDD. We used the same but another SATA HDD and one Flash based SSD for MRIS store. The HDD is also a Maxtor 7L250S0 with a capacity of 250GB and a rotational speed of 7200 rpm. The SSD is an Intel SSDSA2CW300G3 2.5-inch with 300G capacity. The code and benchmark results are publicly available at https://github.com/brianchenming/mris.

4.1 Measure Drives

Firstly, we measured the two storage devices used in our benchmarks. The results we got support our argument that Flash is good in random I/O while HDD is not bad in sequential I/Os. We formated both devices using Ext4. We remount devices before each benchmark to make sure that all disk caches are dropped. The devices were measured using Filebench [7]. Random read and write were measured using Filebench built-in workloads `randomread` and `randomwrite`; Sequential read and write were measured using Filebench built-in workloads `singalstreamread` and `singalstreamwrite`.

The performance of read operations are presented in Figure 6. All benchmarks are performed for 3 times, and the standard deviation of the 3 runs are shown as error bar in the figures. The results are stable and most error bars are imperceptible or negligible. As we can see in Figure 6, SSD is much better than that of HDD for random read for small I/O sizes. SSD is 23.1× faster than HDD when I/O size is 4K, and 18.4× faster when I/O size is 8K. However, the speed advantage drops to 8.4× and 4.8× when the I/O size grows to 64K and 256K. This is because disk head seek happens less frequently as I/O size increases. Specifically, the HDD read throughput is 0.4mb/s with an I/O size of 4K. This agrees with the 9ms average read time shown in the HDD's specification.

For sequential read, the most interesting observation is that HDD provided the same throughput as SSD when I/O size is 4K and 8K. HDD throughput stop grow after 8K, whereas SSD throughput grows until 256K. 8K and 256K are the points when HDD and SSD achieve their respective maximum bandwidth.

The performances of write operations are presented in Figure 7. They present similar trend as of read. When I/O size is small (i.e. 4K, 8k, or 16k), SSD performs much better than HDD for random write, but their performance have no significant difference for sequential write. When I/O size becomes large, throughput of both drives are capped by their maximum bandwidth. However, we noticed that HDD performs better for write than for read. This is because the HDD has a 16MB internal cache, which is more helpful for write than for read.

Above-mentioned results will serve as baselines of throughput for our further benchmarking. They also validate our design premise that SSD performs much better than HDD for random I/Os and HDD performs well for sequential I/Os.

4.2 Wikipedia Image Workload

To study the size-tiered property in workloads, we analyzed the requests of images made to Wikipedia site. Wikipedia is the #5 website in the world [25], serving 492 million people every month [24]. It can reflect workloads of many large websites.

Figure 8 presents the requests of images made in January 2012. They are extracted from request log files from http://dumps.wikimedia.org/other/pagecounts-raw/2012/2012-01/. We extract requests of objects with a suffix of jpg, png, or gif, then round up their sizes into power of 2. The frequency of images of different sizes is

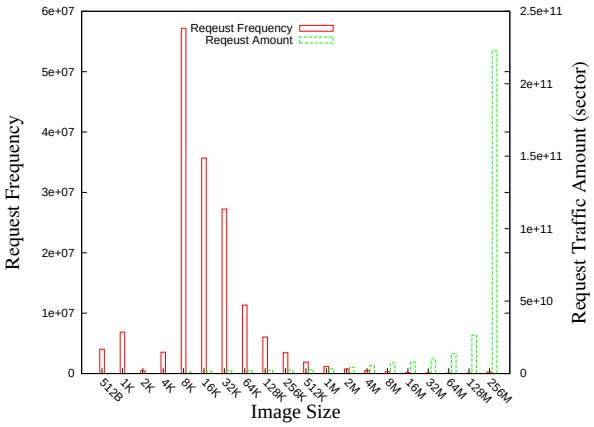


Figure 8: Image Requests to Wikipedia of January 2012

Setup Name	Placement of Objects	
	Small (in SSTable)	Large (in LargeSpace)
SSD	SSD	SSD
Hybrid	SSD	HDD
HDD	HDD	HDD

Table 1: Setups For Benchmarking. The three setups differ in the places objects are stored.

counted and plotted in Figure 8. All images larger than 256M is counted in the 256M bin. The first observation from Figure 8 is that the request image of different sizes vary widely. Particularly, images of sizes between $(4, 64]$ KB are most popular. They sum to 81.58% of the total number of image requests. Moreover, 94.57% of the requests are for images smaller than or equal to 128KB. Despite the fact that small images (smaller than 128KB) are the absolute majority in term of request numbers, the traffic (size$\times$frequency) introduced by them is just 2.96% of the total. Although not all the requests make their way to the storage layer because of memory cache (such as Memcached [14]). This salient size-tiered property of requests still makes size-tiered storage a close match for multi-resolution image workloads.

4.3 MRIS Write

We now show micro benchmark results on simple workloads of sequential and random write of multi-resolution images. Multi-resolution images of the same content are often created from a common source. Therefore, they are inserted into the image store at the same time. For example, In Facebook's Haystack [4], four resolutions, thumbnail, small, medium, and large, are created from each image uploaded by customers. As most images are in compressed format, we emulate the workload of writing MRI by inserting groups of random objects into MRIS with compression disabled. To be simple, each group consists of two objects, both of which are random strings. One of the two objects is 8KB large representing a small image, whereas the other is 128KB large representing a large image. The SizeThreshold is set to 128KB, so that the large one will be stored in LargeSpace.

Figure 9 presents the results of the workload of inserting 10,000 groups of objects into MRIS. We experimented the workload on three different setups in Table 1. The groups were inserted in random and sequential ways considering the order of the key of objects. For ops/sec of random insertion, SSD is 28% faster than HDD and Hybrid is 8% faster than HDD. Considering only SSD and HDD, the speedup of SSD is much lower than that shown in Figure 7. This is because of the Memtable and the log-structured feature of MRIS, which turns many random writes into fewer large sequential write. It also explains how random writes achieved a throughput of 23.9mb/sec even for HDD. As we can see in Figure 7, SSD is only slightly faster than HDD when it is for sequential write of 4K I/O size. Therefore, a slow speedup of insertion is reasonable.

For sequential insertion, SSD is 26% faster than HDD and Hybrid is just 3% faster than HDD. This is because sequential insertion causes less compactions than random insertion, which reduce the overall number of I/Os. Compactions merge multiple sorted SSTable into one larger sorted SSTable. When key/value pairs are inserted sequentially, no merge is necessary because all SSTable have no overlapping keys. This also explains why sequential insertion is significantly faster than random insertion for all the three setups (41%, 38%, and 45% faster for SSD, Hybrid, and HDD).

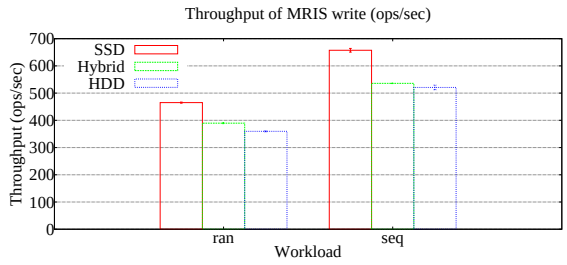
Ming

I really should have recorded the number of compactions happened during the benchmarking. I will do it in the future.

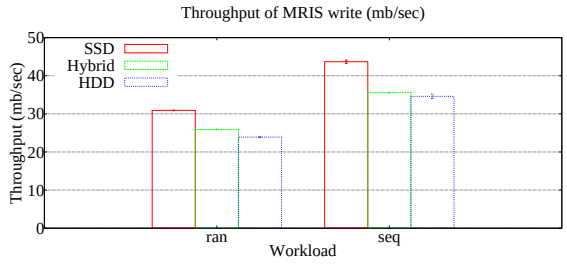
Figure 9 (b) also shows throughput in term of mb/sec. However, the speed-up ratio of mb/sec is almost identical to that of ops/sec (differ in less than 0.1%). This is what we are expecting because the average size of an operation is the same among all setups.

4.4 MRIS Read

Log-structured key/value stores generally have high insertion throughput, however, they also have to provide satisfactory lookup performance. We have developed workloads to evaluate the read performance of MRIS. One of the most interesting parameters in read benchmarking is the ratio of reading small and large images. In Facebook's workload [4], the ratio of requests of small and large images is around 17 (not considering thumbnails and medium). In Wikipedia's workload, the ratio of requests of 8K-image and 128K-image is around 9.5. In our benchmarks, we used different ratios and performed workloads in all the three setups in Table 1. Each benchmark randomly read 10,000 small images from a database of 100,000 groups of images, and following every $ratio$ reads of small images, one large version of the small image is also read. For instance, one large image will be read after every other read of small images when ratio is 2. This emulates the common scenario that cus-



(a) Objects written per second



(b) MB written per second

Figure 9: Write performance of MRIS.

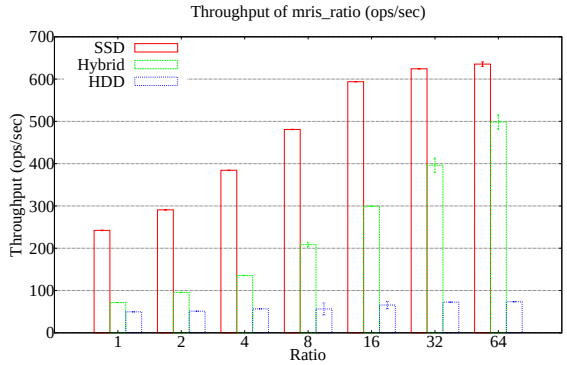


Figure 10: MRIS Read Performance (ops/sec). Each operation represents a read of an image.

Speedup over HDD	Ratio						
	1	2	4	8	16	32	64
SSD	3.89	4.68	5.79	7.52	8.04	7.60	7.64
Hybrid	0.44	0.86	1.39	2.69	3.56	4.46	5.78

Table 2: Speedup of SSD and Hybrid over HDD (ops/sec).

tomers request large images once become interested from the small ones.

Figure 10 presents the read throughput in term of ops/sec. For all ratios, SSD shows the highest throughput among the three setups, followed by Hybrid and then HDD. As shown in Table 2, the ops/sec of SSD is 3.89× to 7.64× faster than HDD. The speedup of SSD over HDD also grows as ratio gets larger (except one point). This is because as the ratio increases, the I/Os contain more small random reads, which exploits more of SSD's superior random performance to HDD. Another interesting observation is that the speedup of Hybrid over HDD grows rapidly from 0.44 to 5.78 as ratio increases.

Read Type	Flash SSD	SATA HDD
Small Image	$t_{SF} = 1482$	$t_{SH} = 13238$
Large Image	$t_{LF} = 6542$	$t_{LH} = 37599$

Table 3: Costs of read operations in time (μs). For instance, t_{SF} is the time of reading a Small image from the Flash SSD.

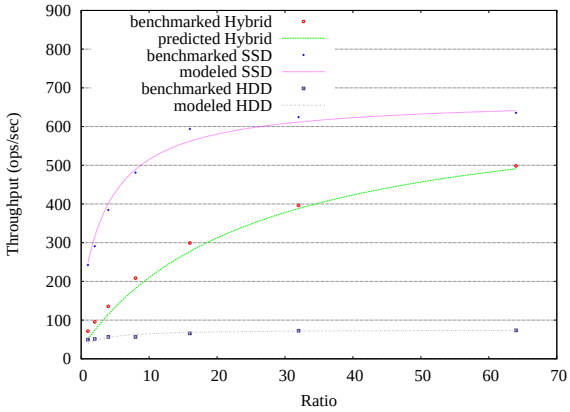


Figure 11: Modeled and benchmarked performance (ops/sec).

To help analyze the results, we use variables in Table 3 to represent the costs of involved read operations in term of time. Then, ops/sec of SSD and HDD can be expressed in (1) and (2).

$$1000000\frac{ratio+1}{t_{SF}*ratio+t_{LF}} \quad (1)$$

$$1000000\frac{ratio+1}{t_{SH}*ratio+t_{LH}} \quad (2)$$

Using linear regression, we estimated the values of the variables (also shown in Table 3) from our benchmark data. Approximately, the ops/sec of Hybrid can be expressed in (3).

$$1000000\frac{ratio+1}{t_{SF}*ratio+t_{LH}} \quad (3)$$

Use (3) and the values in Table 3, we can predict the ops/sec of Hybrid. The results, together with measured ops/sec in our benchmarks, are presented in Figure 11. We can see that the measured ops/sec matches the shape of the predicted ops/sec. Moreover, the measured ops/sec are all slightly higher than the predicted counterparts.

For the same workloads, the throughputs in term of mb/sec are presented in Figure 12. Different from the results of ops/sec, mb/sec is decreasing as ratio increases for both SSD and HDD. This is because more of the operations are reads of small images when ratio is large. As we can see in (4), the average size of an operation is a monotonically decreasing function of ratio where $large = 128$KB and $small = 8$KB are the sizes of large and small images respectively.

$$\begin{aligned} \text{avg\_op\_size} &= \frac{ratio*small+large}{ratio+1} \\ &= \text{small} + \frac{large-small}{ratio+1} \end{aligned} \quad (4)$$

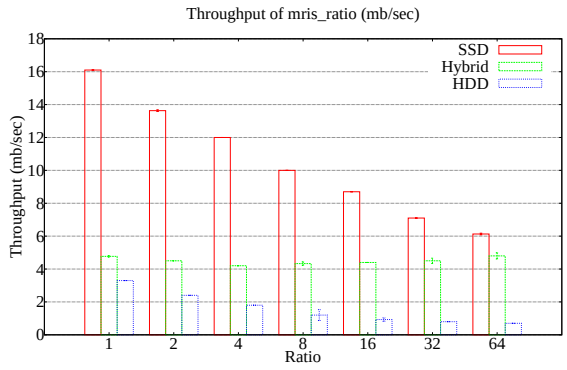


Figure 12: MRIS Read Performance (mb/sec).

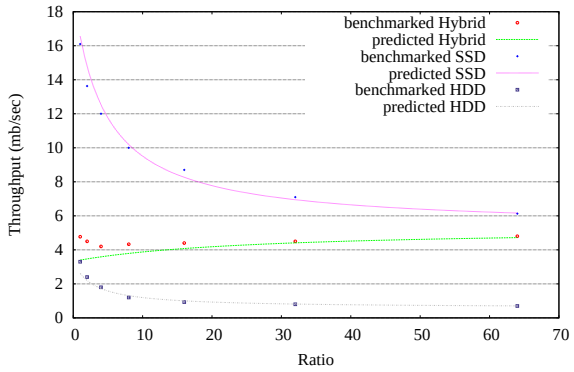


Figure 13: Predicted and benchmarked read performance (mb/sec).

However, for Hybrid, the throughputs in Figure 12 are relatively stable. Derived from (3), the mb/sec of Hybrid can be expressed in

$$1000000\frac{ratio * small + large}{t_{SF} * ratio + t_{LH}}. \quad (5)$$

Similarly, we can predict the mb/sec of SSD and HDD. The predicted mb/sec results, along with the benchmarked mb/sec results, are presented in Figure 13. We observed that there exists significant discrepancy in Hybrid's mb/sec results when ratio is small. This is because the average operation size is large when ratio is small, as we can see from (4). The distances between predicted ops/sec and benchmarked ops/sec of small ratios, in Figure 10, are magnified when turned to mb/sec by multiplying the average operation size.

As presented in Table 4, Hybrid improves HDD's mb/sec throughput by at least 45%. The improvement grows to as high as 5.86× when ratio is 64. Specifically, the speedup is 2.61× and 3.73× when ratio is 8 and 16, which approximate to the ratios in the workloads of Wikipedia and Facebook.

We have also measured the throughput (mb/sec) went to each drive using `iostat`. The results are shown in Figure 14. We observed that the throughput reported by iostat is much larger than that in Figure 10. Three factors

Speedup over HDD	Ratio						
	1	2	4	8	16	32	64
SSD	3.88	4.68	5.67	7.33	8.35	7.87	7.76
Hybrid	0.45	0.88	1.33	2.61	3.73	4.62	5.86

Table 4: Speedup of SSD and Hybrid over HDD (mb/sec).

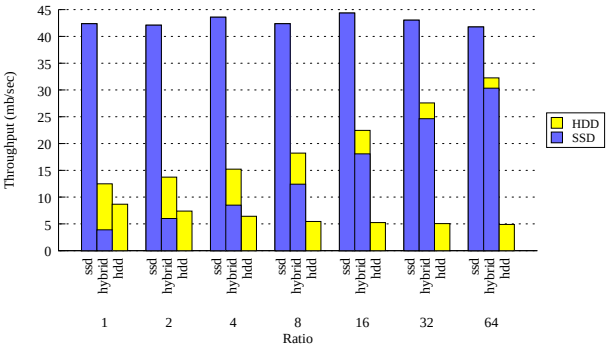


Figure 14: MRIS Read Performance (mb/sec) by iostat

contribute to the extra throughput: 1) the read-ahead in the filesystem, 2) extra read of filesystem metadata, and 3) extra read of database metadata. However, it is interesting to notice that the data read from SSD for the SSD setup is quite stable even when ratio varies dramatically. This is something we need to further investigate.

5 Related Work

Our work of optimizating performance of a key/value store using hybrid storage devices is related to (1) hybrid filesystems, (2) multi-tier storage, and (3) multi-level caching.

(1) Hybrid Filesystems. hFS [27] is a hybrid filesystem which treats data differently based on their size and type. Metadata and small files are stored separately in a log partition like log-structured filesystem; data blocks of large regular files are stored in a partition in a FFS-like fashion. Similar to hFS, Conquest [23] uses battery-backed RAM to hold metadata and small files. Only large files go to disk. Unlike hFS, UmbrellaFS [11] is a hybrid stackable filesystem sit bellow VFS but above general filesystems such as Ext2. UmbrellaFS is able to use different devices including SSD. TableFS [17] uses NoSQL store for metadata and small files. However, its main objective is to improve the performance of a filesystem using NoSQL store.

Whereas all of them integrate hybrid techniques into the filesystem layer, our system lies in the application layer which is above the filesystem layer. It optimizes the operations of an object store, which provides a different interface from the POSIX filesystem interface. This is an important difference because the filesystem interface lacks application level knowledge, which is very useful in optimizating application performance.

(2) Multi-tier Storage. GTSSL [21] presents an efficient multi-tier key/value storage architecture, wherein

Flash SSD is used for storing top layer SSTable above the disk. However, workload specific characters, such as size-tiered property are not exploited. As a follow-up research of GTSSL, MRIS can integrate nicely with GTSSL. Koltsidas and Viglas [12] improved the performance of database by using both Flash and disk drives. They decided either Flash or disk should be used in the database buffer manager, which is also unware of knowledge such as object size. Moreover, their study was based an old model, which considers Flash write to be 10 times slower than disk write. This is no longer true, as we see from Subsection 4.1, thanks to the development of hardware and software used in Flash SSD. FAWN [1] is a distributed multi-tier key/value store. They used a two-tier architecture of RAM and Flash SSD, but not Flash SSD and HDD. Most of their strategies are not applicable here because of the large performance disparity between RAM and disk. Furthermore, their design for energy saving is orthognal to our work.

(3) Multi-level Caching. Storage class memory such as Flash fills the gap between DRAM and HDD in termns of cost, capacity and performance. It can be considered either as backup for DRAM in the virtual memory layer or cache for HDD in the block layer. Zhang et al. [26] and Saxena et al. [18] consider using Flash as backup of DRAM for paging, whereas FlashTier [19], FlashCache [9] and Bcache [3] use Flash as block level cache. Our work resides in neither the virtual memory nor the block layer. It is agnostic to all the above-mentioned techniques. Moreover, both Zhang et. al. [26] and Saxena et al. [18] tried to reduce cost by replacing a portion of DRAM with SSD without cost penalty. This objective does not conflict with design of MRIS, however, it is not in our design concerns.

Forney et al. [10] proposed storage aware caching for heterogeneous storage systems. They made memory cache aware of the different replacement costs and partitioned the cache for different storage devices. However, their study is set in a different context which is a network-attached disk system. They did not consider data placement among different drives, which is an important strategy of our study.

6 Conclusions

We present a multi-tier key/value store using Flash SSD and HDD. It exploits the size-tiered feature in many workloads, especially multi-resolution image workload. It achieves good balance of performance and cost by leveraging SSD's high random I/O throughput as well as HDD's good sequential I/O performance. With a small Flash of 5.9% of the total storage capacity, it is able to improve the ops/sec of an emulated Facebook image workload by 3.73×.

Future Work. We plan to further study the model of multi-tier key/value considering cost, characteristics of different drives, as well as more workload specific features besides the ratio of read small and large objects. More threads will be used in benchmark to exploit better parallelism in Flash SSD.

Acknowledgement. The author would like to thank Vasily Tarasov and Mike Ferdman for their valuable comments.

References

[1] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '2009)*, pages 1–14. ACM SIGOPS, October 2009.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[3] A Linux kernel block layer cache. *http://bcache.evilpiepirate.org/*.

[4] D. Beaver, S. Kumar, H.C. Li, J. Sobel, P. Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. *Proc. 9th USENIX OSDI*, 2010.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[6] K.M. Evans and G.H. Kuenning. A study of irregularities in file-size distributions. In *Proceedings of the 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*. Citeseer, 2002.

[7] Filebench. *http://filebench.sourceforge.net*.

[8] Flash Memory. *http://en.wikipedia.org/wiki/Flash_memory*.

[9] A Write Back Block Cache for Linux. *https://github.com/facebook/flashcache/*.

[10] Brian C. Forney, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Storage-aware caching: Revisiting caching for heterogeneous storage systems. In *Proceedings of the 1st USENIX Confer-*

ence on File and Storage Technologies, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[11] John A. Garrison and A. L. Narasimha Reddy. Umbrella file system: Storage management across heterogeneous devices. *Trans. Storage*, 5(1):3:1–3:24, March 2009.

[12] Ioannis Koltsidas and Stratis D. Viglas. Flashing up the storage layer. *Proc. VLDB Endow.*, 1(1):514–525, August 2008.

[13] LevelDB, January 2012. *http://code.google.com/p/leveldb*.

[14] Memcached. *http://memcached.org*.

[15] Multiresolution Image Processing. *https://inst.eecs.berkeley.edu/~ee225b/fa12/lectures/multi-resolution-girod.pdf*.

[16] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[17] Kai Ren and Garth Gibson. Table fs: Embedding a nosql database inside the local file system. Technical Report CMU-PDL-12-103, Carnegie Mellon University, 2012.

[18] M. Saxena and M. M. Swift. Flashvm: Revisiting the virtual memory hierarchy. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, 2009.

[19] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: a Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, 2012.

[20] Vadim Skipin. New feature: Limited record lifetime. *https://groups.google.com/d/topic/leveldb/cIM9FdJGG1w/discussion*, 2012. [Online; accessed 10-Dec-2012].

[21] R. P. Spillane, P. J. Shetty, E. Zadok, S. Archak, and S. Dixit. An efficient multi-tier tablet server storage architecture. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)*, Cascais, Portugal, October 2011.

[22] Anatomy of SSDs. *http://www.linux-mag.com/id/7590/*.

[23] An-I Andy Wang, Geoff Kuenning, Peter Reiher, and Gerald Popek. The conquest file system: Better performance through a disk/persistent-ram hybrid design. *Trans. Storage*, 2(3):309–348, August 2006.

[24] Frequently asked questions - Wikimedia Foundation. *http://rvarchivo.blogspot.com/2012/11/frequently-asked-questions-wikimedia.html*.

[25] Wikipedia. *http://wikimediafoundation.org/wiki/WMFJA085/en*.

[26] Z. Zhang, Y. Kim, X. Ma, G. Shipman, and Y. Zhou. Multi-level hybrid cache: Impact and feasibility. Technical report, Oak Ridge National Laboratory (ORNL), 2012.

[27] Zhihui Zhang and Kanad Ghose. hfs: a hybrid file system prototype for improving small file and metadata performance. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 175–187, New York, NY, USA, 2007. ACM.