# Why are state-of-the-art flash-based multi-tiered storage systems performing poorly for HTTP video streaming?

Moonkyung Ryu
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
mkryu@gatech.edu

Hyojun Kim
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
hyojun.kim@cc.gatech.edu

Umakishore Ramachandran
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
rama@cc.gatech.edu

## ABSTRACT
MLC flash memory is a promising technology for building a high-performance and cost-effective video streaming system when it is used as an intermediate level cache in a multi-tiered storage hierarchy. Therefore, we were quite surprised when through extensive measurements we found that two state-of-the-art flash-based multi-tiered storage systems (namely, flashcache and ZFS) have quite disappointing performance for HTTP video streaming using the DASH protocol. We have conducted a thorough analysis to understand the reasons for the poor performance of these two systems. In a nutshell, unless attention is paid to the unique performance characteristics of flash memory-based SSDs, we could end up with suboptimal or even poor performance as we discovered through experimentation with these two systems. Based on the analysis, we present design guidelines for building a cost-effective high-performance HTTP video streaming server.

## Categories and Subject Descriptors
C.4 [**Computer Systems Organization**]: Performance of Systems

## General Terms
Experimentation, Measurement, Performance

## Keywords
Flash Memory, Solid-State Drive, Video-on-Demand, HTTP Video Streaming, Content Distribution Network

## 1. INTRODUCTION
There is proliferation of video on the Internet. Hulu [7], a web service that streams premium contents such as news, TV series, and shows, and Netflix [9], the largest subscription service for DVD rental and streaming video over the web, are huge successes.

Dynamic Adaptive Streaming over HTTP (DASH) [8] is a new paradigm of video streaming over the web currently being used by major content distributors, including Netflix. DASH does not depend on specialized video servers. Instead, DASH exploits off-the-shelf web servers. The advantage of this paradigm is that it can easily exploit the widely deployed content distribution network (CDN) infrastructure for the scalable video streaming service, and the firewall and NAT traversal problems can be greatly simplified. On the other hand, its weakness is that there is no QoS mechanism on the server side to guarantee video delivery. It relies on the large resource provisioning (i.e., CPU, RAM, storage capacity, storage and network bandwidth, etc.) that is customary with CDNs. This approach achieves simple system design, scalability, and jitter-free video streaming at the cost of large resource over-provisioning.

The video streaming system requires a number of hard disk drives (HDDs) both for capacity (to store the video library) and for bandwidth (to serve the video library). While the cost per gigabyte of HDDs has decreased significantly, the cost per bits-per-second of HDDs has not. Moreover, an array of HDDs consumes a lot of power (approx. 5-15 watts per drive) and generates a large amount of heat; therefore, more power is required for cooling a data center hosting a large array of disks. The amount of video content and the corresponding number of viewers are increasing explosively on the web. In addition, the DASH approach requires storage bandwidth over-provisioning for reliable video streaming service; therefore, the cost of storage for a large scale service is significant. For these reasons, the storage component of the video streaming system needs a careful re-evaluation to achieve higher throughput for the same dollar investment while lowering the power consumption and cooling costs.

Solid-State Drive (SSD) is a new storage device that is comprised of semiconductor memory chips (e.g., DRAM, Flash Memory, Phase Change Memory) to store and retrieve data rather than using the traditional spinning platters, a motor, and moving heads found in conventional magnetic disks. Among various types of SSDs, flash-based SSDs currently have the maximum penetration into modern computer systems. The advantages of flash-based SSDs are fast random read, low power consumption (approx. 0.1-1.3 watts per drive), and low heat dissipation due to the absence of the mechanical components. On the other hand, its high cost per gigabyte compared to magnetic disks, poor small random write performance, and limited lifetime are major concerns compared to the disks.

Though flash-based SSDs are attractive as an alternative to HDDs for video storage for all the above reasons, the cost per gigabyte for SSD is still significantly higher than HDDs. Moreover, despite the increasing affordability of SSDs, the ratio of capacity costs of SSD to HDD is expected to remain fairly constant in the future since the bit density of HDDs is still continuously improving. Therefore, a viable architecture is to use the flash-based SSDs as an intermediate level between RAM and HDDs for caching hot contents. In our prior work [19], we explored the efficacy of such a design. In particular, we showed via simulation that low-end flash devices are ideal to incorporate in the design of such multi-tier video servers to reduce the overall capital expenditure and operating costs while achieving a high throughput.

While enterprise-level multi-tier storage systems (incorporating high-end Single Level Cell (SLC) flash memory-based SSDs) have been around for a while, they are not cost-effective for use as streaming video servers. Recently, commercial products have emerged that have incorporated low-cost Multi Level Cell (MLC) flash memory SSDs. Zettabyte File System (ZFS) [10] and Flashcache [5] are two state-of-the-art solutions that serve as good examples.

Our goal in this paper is to experimentally verify the performance of two commercial solutions that incorporate flash in their storage hierarchy, namely ZFS [10] and Flashcache [5], to serve as HTTP streaming servers embodying the DASH protocol. Based on our prior simulation work [19], our expectation is that these two systems will be ideal vehicles for realizing our vision of low-cost high-performance HTTP servers. However, our experimental results are surprising since neither of these two systems met our performance expectations. The core of the intellectual contribution of this paper is shedding light on the reasons for the poor performance of these two systems for HTTP video streaming. Unless otherwise mentioned, a *cache* refers to a flash memory SSD rather than RAM in this paper.

The unique contributions of our work are as follows:

1. We measure performance of two state-of-the-art flash memory caching systems, which are Zettabyte File System (ZFS) [10] and Flashcache [5]. By running an Apache [1] web server on these two systems, we evaluate the storage performance of the systems for video streaming using the DASH approach.

2. The performance of both these systems is surprisingly disappointing. We undertake an in-depth analysis to understand the reasons for the poor performance of both these systems for the video streaming workload.

3. Armed with the knowledge of the sources of poor performance of these state-of-the-art systems, we propose design guidelines for a high-performance HTTP video streaming server.

The rest of the paper is organized as follows. Section 2 provides the background about flash memory SSD and dynamic adaptive streaming over HTTP. In Section 3, we measure the performance of 3 different storage configurations, HDDs only, ZFS, and Flashcache, for HTTP video streaming using the DASH protocol. Section 4 analyzes the measurement result in detail. Section 5 presents guidelines for a high-performance HTTP video streaming server stemming from our experimental study. Our concluding remarks are presented in Section 6.

## 2. BACKGROUND

In this section, we briefly review two technologies central to the problem being addressed in this paper: flash-based SSDs and DASH protocol for video streaming.

### 2.1 Flash-based SSD

Flash memories, including NAND and NOR types, have a common physical restriction, namely, they must be erased before being written [12]. Flash memory can be written or read a single page at a time, but it has to be erased in an *erase block* unit. An erase block consists of a certain number of pages. In NAND flash memory, a page is similar to a HDD sector, and its size is usually 2 to 4 KBytes, while an erase block is typically 128 pages or more.

Flash memory also suffers from a limitation on the number of erase operations possible for each erase block. In SLC NAND flash memory, the expected number of erasures per block is 100,000 but is only 10,000 in two-bit MLC NAND flash memory.

An SSD is simply a set of flash memory chips packaged together with additional circuitry and a special piece of software called the Flash Translation Layer (FTL) [14, 17]. The additional circuitry may include a RAM buffer for storing meta-data associated with the internal organization of the SSD and a write buffer for optimizing the write performance of the SSD.

To avoid erasing and re-writing an entire block for every page modification, an FTL writes data out-of-place, remapping the logical page to a new physical location and marking the old page invalid. This requires maintaining some amount of free blocks into which new pages can be written. These free blocks are maintained by erasing previously written blocks to allow space occupied by invalid pages to be made available for new writes. This process is called *garbage collection.* FTL tries to run this process in the background as much as possible while the foreground I/O requests are idle, but it is not guaranteed, especially when a new clean block is needed instantly to write a new page. Due to random writes emanating from the upper layers of the operating system, a block may have valid pages and invalid pages. Therefore, when the garbage collector reclaims a block, the valid pages of the block need to be copied to another block. Thus, an external write may generate some additional unrelated writes internal to the device, a phenomenon referred to as *write amplification.*

### 2.2 Dynamic Adaptive Streaming over HTTP

Dynamic Adaptive Streaming over HTTP (DASH) [8] is a new paradigm for video streaming over the web. Rather than rely on the dedicated video servers of yesteryears, DASH exploits off-the-shelf web servers for video streaming. The protocol works as follows: A *video object* can be in two different forms; a single large file or multiple small *segment* files that have the same play-out duration, typically a few seconds long. Further, there may be multiple versions of the same video object, supporting different bitrates and different quality levels. Web servers on the Internet that constitute a CDN store these multiple versions of videos. A player (i.e., a client) can then request different segments at different bitrates depending on the state of the underlying network. If the video object is in the form of a single file, the player requests a segment using the byte ranges protocol of HTTP/1.1 [6]. On the other hand, when the video object is in the form of multiple small segment files, the player requests a segment file download. Notice that it is the player that decides the bitrate to request for any segment. The server treats requests for segments of video files similar to any other normal web request and does not do anything special. This greatly simplifies the server design, and allows the use of existing CDN systems without modification.

## 3. MEASUREMENT

In this section, we measure the performance of 3 differ-

| | SSD A | SSD B |
|---|---|---|
| Model | INTEL X25-M G1 | OCZ Core V2 |
| Capacity | 80 GB | 124 GB |
| 4KB Random Read Throughput | 15.5 MB/s | 14.8 MB/s |
| 4KB Random Write Throughput | 3.25 MB/s | 0.02 MB/s |

**Table 1: MLC Flash Memory SSDs that were used for our experiments. Both SSDs have similar performance for small random reads. On the other hand, the different SSDs have significantly different small random write performance.**

ent storage configurations of an HTTP streaming server using the DASH protocol: HDDs only, Flashcache, and ZFS. Apache [1] is used as the web server. We implement a workload generator that emulates a large number of concurrent DASH clients. To measure the storage subsystem performance exactly and to avoid network subsystem effects, the Apache web server and the workload generator run on the *same* machine communicating via a loop-back interface. The machine has Xeon 2.26 GHz Quad core processor with 4 GB RAM, and Linux kernel 2.6.32 is installed on it. Unless otherwise noted, *cache* in this paper refers to the flash device used as an intermediate level in the storage hierarchy and not RAM.

## 3.1 MLC Flash Memory SSDs

Table 1 lists the MLC flash memory SSDs we used for our experiments. Due to space limitations, we present only results with SSD A. However, the results with SSD B are similar.

## 3.2 DASH Workload

Zipf distribution is generally used in modeling the video access pattern of a video-on-demand (VoD) system, and typically a parameter value between 0.2 and 0.4 is chosen for the distribution [16]. We use 0.271 for the parameter in our study. For a test video sequence, we use *Valkaama* [3] which is available in the public domain. The video object is segmented into 10-second long segments, the total length of the video is about 78 minutes, the size is 1.06 GB, and the average bitrate is 2 Mbps. The video object is composed of 466 segment files. Though each segment has the same play-out time, the size of each segment is different since the video is encoded with a variable bit rate (VBR). In the DASH approach, a video object is encoded in a few different bitrates so that a client can adaptively select a bitrate according to the dynamically changing network state. In our experimental setup, the workload generator that emulates a number of client requests and the web server run on the same machine. Hence the network state is stable and the different bitrate profiles available in the web server for supporting the DASH protocol are not necessary. Moreover, for a controlled experiment and a fair comparison of results, it is better to use a single bitrate profile for the video, as we do in this study. We copy the video, and make 300 individual video objects.

In every $t$ seconds (which is the inverse of the request rate), the workload generator selects a video object according to the zipf distribution. Next, it chooses a segment of the video object according to the uniform distribution; each segment of the video object has the same probability. The reason for using a uniform distribution is as follows. A large scale HTTP video streaming service like Netflix relies on the CDN infrastructure that widely deploys web cache servers near the edge networks. For an effective load balancing, a video segment (or object) is replicated to a number of web cache servers, and a client's request for the segment is directed to a web server holding the segment via request routing techniques such as DNS routing, HTML rewriting [13], or anycasting [18]. For this reason, there is no guarantee that a client who downloaded a segment *i* of a video will download a next segment *i+1* from the same server. The next segment can be served by other web servers that hold a replica. Therefore, it is reasonable to assume a uniform distribution of segment requests to any given web server.

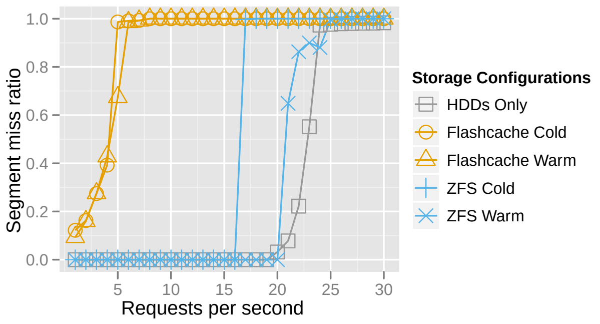


**Figure 1: Segment miss ratio as a function of Request Rate.**

The workload generator sends an HTTP request for the chosen segment to the web server. When the segment is not downloaded to the client within the segment's play-out time (i.e., 10 seconds for our test video), the client counts it as a segment deadline miss. We measure the segment miss ratio against different request rates. *Segment miss ratio* is defined as the ratio of the number of segment deadline misses to the total number of segment requests for a given request rate. Therefore, the *requests per second* is a control parameter, and the segment miss ratio is the measured figure of merit of the storage subsystem.

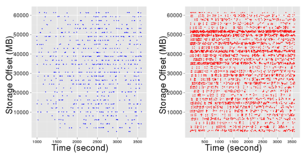
## 3.3 HDDs Only

As a base configuration for comparison, we use two 7200 RPM HDDs striped per Linux's software RAID-0 configuration. Software RAID is implemented in Linux using the device mapper, which is a Linux storage stack infrastructure. The ext4 file system [4] is installed on these RAID-0 disks. We measure the segment miss ratio with different request rates, and each measurement is run for an hour. Figure 1 shows for this configuration that the system could serve up to 19 requests per second when the required QoS is 0% segment miss ratio. Since the observed average CPU utilization during the measurements was below 10%, we can conclude that storage is the bottleneck beyond 19 requests per second for this configuration.

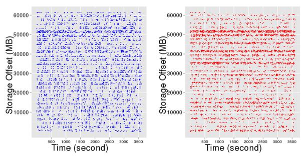
## 3.4 Flashcache

Flashcache [5], developed by facebook, is a write-back persistent block cache designed to accelerate reads and writes from slow storage like HDDs by caching data in faster storage like SSDs. While ZFS is a file system level solution for caching in flash memory, flashcache is a block device level solution, therefore, it is very general and can be utilized at different levels of the software stack (e.g., file systems, applications).

We create a single logical block device by having flashcache use the two 7200 RPM HDDs striped per RAID-0 configuration and SSD A (see Table 1). Out of the total 80 GB available in the SSD, 60 GB are used as flashcache for the experiment. We measure the performance in two different cache states; a cold cache and a warm cache. For the cold cache, we flush the cache at the beginning of each measurement. For the warm cache, we run the workload



**(a)** Cold cache.



**(b)** Warm cache.

**Figure 2: Flashcache's Read (Blue) and Write (Red) access pattern on the cache during 1 hour with 1 request per second. The x-axis is time and the y-axis is the storage offset. Both read and write access patterns are severely random.**

generator until the cache filling rate falls below 100 KB/sec. We filled the cache up to 60% (i.e., 36GB) by running the workload generator with 1 request per second for 12 hours. Each measurement is run for an hour. Figure 1 shows that flashcache could not serve even 1 request per second when the required segment miss ratio is 0%. The flashcache performance is the same whether the cache is cold or warmed up.

## 3.5 ZFS

ZFS [10] is a file system that has an intermediate layer to serve as a read cache between the RAM and the HDDs, called L2ARC. Storage devices that are faster than HDDs are used for L2ARC devices. Though not a requirement, flash memory SSDs can be used as L2ARC devices. ZFS was originally introduced and implemented in the Solaris operating system, but it has been ported to other operating systems such as FreeBSD and Linux. We use ZFS on Linux for this measurement.

ZFS creates a single storage pool using the two 7200 RPM HDDs in RAID-0 configuration together with SSD A to serve as the intermediate read cache layer. Like the flashcache experiment, we use 60 GB of the SSD capacity for the L2ARC cache, and measure the performance in two different cache states; a cold cache and a warm cache. For the cold cache, we flush the cache at the beginning of each measurement. For the warm cache, we have run the workload generator until the cache is filled in full. Each measurement is run for an hour.

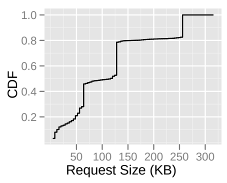
Figure 1 shows that ZFS with the cold cache could serve up to 16 requests per second when the required segment miss ratio is 0%. On the other hand, when the cache is warmed up, ZFS could serve up to 20 requests per second. ZFS shows a lot better performance than flashcache, but it is still worse than the base configuration (HDDs-only) when the cache is cold and only slightly better after the cache is warmed up.

## 4. ANALYSIS

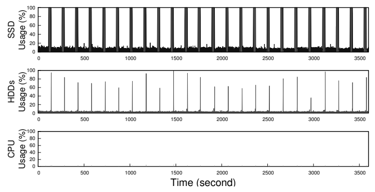
The poor performance of flashcache and ZFS is very surprising since both ZFS and flashcache are designed with flash memory SSDs in mind. In this section, we investigate the reasons for this result.

## 4.1 Flashcache

Flashcache organizes the flash memory as a set associative cache. The block size, set associativity, and cache size are configurable parameters, specified at cache creation time. The default block size is 8 sectors (i.e., 4 KB), and the default set associativity is 512 (i.e., a given disk block could be in one of 512 members of a given set) Replacement policy is either FIFO or LRU within a set, and FIFO is the default. We use default values for all the parameters in our flashcache measurements.



**(a)** Cumulative Distribution Function (CDF) plotted against the write request sizes sent to the cache over a period of 1 hour.



**(b)** Resource utilization for SSD, HDD, and CPU (respectively, from top to bottom) as a function of time. SSD (top graph) is 100% busy periodically.

**Figure 3: Flashcache's write request size distribution and resource utilization during 1 hour with 1 request per second and cold cache. The median write request size is 116 KB.**

In what follows, dbn refers to *disk block number*, the logical device block number in sectors. To compute the target set for a given dbn

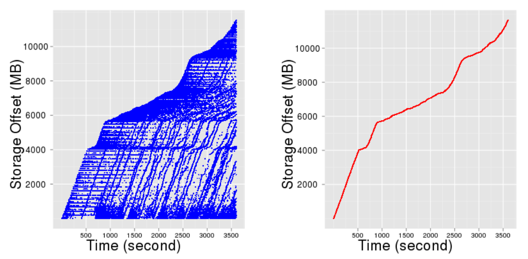
$$target\ set = \left(\frac{dbn}{block\ size \times associativity}\right)\ mod\ (number\ of\ sets)$$

Once we have the target set, flashcache does a sequential search of the set (linear probing) to find the desired disk block. Note that a sequential range of disk blocks will all map onto a given set. On the other hand, disk blocks that have a dbn difference greater than the associativity will map onto different sets.

When flashcache gets a read request, it calculates the target set from the dbn of the request. Then, it probes the cache set to find the requested block. If it finds the block in the cache, it returns the block. Otherwise, flashcache reads the block from the disk, copies the block into the appropriate cacheline, and returns the block. A block request that misses in the cache, will first write the block into the cache before returning it to the application. Thus the cache write operation is in the critical path of satisfying an application layer block request that misses in the cache. Therefore, random reads on disks that miss the cache will generate random writes to the cache, and the read could be delayed and miss its deadline if the cache write operation takes a long time to complete.

Figure 2[1] depicts the read and write access patterns on SSD A for flashcache in two different cache states for an hour period with 1 request per second. This graph is plotted based on the trace data collected using *blktrace* [2], which comes with the Linux 2.6 kernel, to trace the I/O activities at the block device level. To remind the reader, read requests to the flash come from the client requesting video segments that are already cached in the flash; the write requests come from the need to copy blocks from the disk to the flash for requests that miss the flash. The upshot is that both the read and write access patterns are random, and the median write request size is 116 KB (See Figure 3(a)). The request size is defined by the I/O request size sent by the block device driver to the physical block device, which we can determine using *blktrace*. Flash memory shows the best write performance when the request size is a multiple of the erase block size, which is 32 MB for SSD A. A write request size of 116 KB is much smaller than the optimal write request size (i.e., 32 MB). The effect of these small random writes is disastrous on the flash memory performance. In particular, since flash

[1]The purpose of this figure is to show that both read and write access patterns for cold and warm cache settings are completely random, as is evident from the distribution of the data points in the scatter plot shown in the figure.



**(a)** Read access pattern. **(b)** Write access pattern.

**Figure 4: ZFS's Read (Blue) and Write (Red) access pattern on the cache during 1 hour with 16 requests per second and cold cache. The write access pattern is sequential while the read access pattern is random. The median write request size is 128 KB.**

memory does not support in-place writes, requires erase-before-write, and also requires the erase size be larger than the page size, the small and random write operations will consume fresh pages in a clean block. Ultimately this will lead to the situation where all available clean blocks are used up, thus kick-starting the garbage collection process. Looking at the top chart in Figure 3(b), which shows the SSD utilization, we can see the utilization peaking up to 100% periodically (around every 140 seconds). It is highly likely that this is due to the garbage collection process. Overall, flashcache generates a very inefficient access pattern for the MLC flash memory SSD. Considering that SSD A is known to handle small random writes much better than the other MLC flash memory SSDs (see Table 1), flashcache generates extremely inefficient write access pattern that even SSD A cannot handle very well.

The read hit ratio of flashcache with a cold cache (Figure 2(a)) is 0.7% while the read hit ratio of flashcache with warm cache (Figure 2(b)) is 25.7%. However, as we have seen in Figure 1, flashcache shows absolutely no difference in performance as measured by the application level segment miss ratio. This can be explained as follows. From the scatter plot for write requests shown in Figure 2(b), we notice that there is a significant volume of write requests to the cache even when the cache is warm. Thus, the application generated reads and system generated writes (a consequence of miss handling) are competing for resources on the SSD device (RAM buffer and CPU). Further, the device processes the requests in order. Thus, if the reads are queued up behind writes, they are likely to miss their deadline even though they hit in the cache since random writes take a long to complete. This is the most likely reason for the poor application level segment miss ratio observed even with a warm cache in Figure 1 for flashcache.

The reasons for the high segment miss ratio for flashcache while dealing with the DASH workload can be summarized as follows:

1. Upon a cache miss, the block read from the disk has to be first written to the cache before being served to the application. The ensuing small and random write pattern results in long latencies.

2. Second, since flashcache does not give priority to reads over writes to the cache, reads which would be hits in the cache get queued up behind ongoing writes.

Serving the requested segments should be the top priority for a video streaming system. Not respecting this criterion is ultimately the failing of flashcache for this workload.

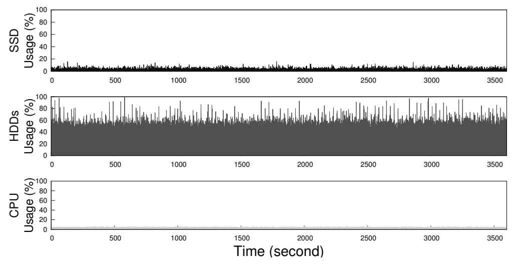


**Figure 5: ZFS's resource utilization during 1 hour with 20 requests per second and warm cache. SSD, HDDs, and CPU from top to bottom.**

## 4.2 ZFS

ZFS is smarter than flashcache in handling the flash memory. By analyzing the source code for ZFS L2ARC, we have determined that ZFS basically uses the L2ARC as a FIFO buffer. By definition, the blocks written into the cache are clean (i.e., they are just copies of the disk blocks) and therefore they never have to be written back to the disk. ZFS maintains a *write pointer* to the FIFO buffer that L2ARC represents. Once the write pointer reaches the end of the FIFO buffer it simply wraps around to the beginning overwriting the existing blocks in the cache. In other words, ZFS L2ARC converts the random writes to sequential writes. This is evident from Figure 4(b), where it can be seen that ZFS L2ARC sequentially fills up the cache over time.

We have measured the median write size to be 128 KB from the block traces. Interestingly, we have ascertained by executing a micro-benchmark for SSD A that its sequential write throughput peaks at 128 KB. Figure 5 also corroborates our analysis. The top graph in Figure 5 is the activity observed on the SSD. It can be seen that this activity is very low indicating that the SSD performance is not the bottleneck. Thus overall ZFS L2ARC seems to be optimized in handling the MLC flash memory SSDs. That begs the question as to why ZFS performs much worse than the base configuration (HDDs-only) as shown in Figure 1?

The answer is simply due to the fact that L2ARC is being used as a FIFO buffer. In other words, the replacement policy for the L2ARC cache is FIFO. As we noted treating L2ARC as a FIFO buffer is great for write throughput considering the nature of the flash device. However, this leads to a very poor hit ratio. Unfortunately, it is not possible to empirically verify this hypothesis since Linux port of ZFS does not provide statistics such as L2ARC hit ratio. Therefore, we approximate the hit ratio from the amount of read size requested from the cache and from the disks, which can be obtained by analyzing the block trace. We use following formula to approximate the hit ratio:

$$Hit\ Ratio = \frac{Amount\ of\ read\ on\ SSD}{Amount\ of\ read\ on\ SSD + Amount\ of\ read\ on\ disks}$$

Using this approximation, L2ARC hit ratio is 4.2% and 11.1% for the cold cache and the warm cache, respectively. The low hit ratio is reflected in the low SSD utilization shown in Figure 5.

We have also applied the above formula to flashcache, and the hit ratio is 0.67% and 23.6% for the cold cache and the warm cache respectively. Compare these numbers to the statistics that flashcache provides. They are 0.7% and 25.7% for the cold cache and the warm cache respectively (Refer to Section 4.1), therefore, we can say that our approximation for the hit ratio derived from block traces is close to reality.

## 5. RECOMMENDATIONS FOR AN HTTP VIDEO SERVER DESIGN

By studying and analyzing the performance of state-of-the-art multi-tier storage systems for DASH video streaming, we have learned a number of lessons. We summarize these lessons in the form of recommendations for constructing a cost-conscious high-performance HTTP streaming server adhering to the DASH protocol.

**Recommendation 1**: **No small random writes**: Small random writes are extremely inefficient for MLC flash memory SSDs. There are two possible solutions to this problem. First solution is a logging approach. This approach transforms the random writes to sequential writes that the MLC flash memory SSDs can handle very efficiently. However, logging necessitates frequent invocation of the garbage collection anytime it needs to clean obsolete blocks and make room for new data blocks that need to be written. Clearly, this is detrimental to real-time performance such as timely video delivery, since foreground read operations can be stymied and delayed resulting in missing application deadlines. Therefore, logging is not an appropriate solution for the small random write problem in a real-time system like video streaming. A more preferred solution is to write using a much larger granularity. If write operations are requested in multiples of the flash memory's erase block size, and their offset is aligned with multiples of the erase block size, write amplification will not occur, and MLC flash memory SSDs can handle the writes very efficiently even if the access pattern is completely random.

**Recommendation 2**: **No flash writes on the critical path**: Writes to the cache while servicing a cache miss should not be in the critical path of serving the requested segment to the application. Hardware caches in a processor are routinely designed in this fashion, wherein the missing data (due to load instruction) is supplied to the processor in parallel with updating the cache. Failing to do this in a video server guarantees that a read request that misses in the cache, will incur the additional penalty of write to the cache when the data is brought from the hard disk. ZFS solves this problem using an evict-ahead policy by which a separate thread copies blocks that are supposed to be evicted soon from RAM to the flash [15, 11].

**Recommendation 3**: **Higher priority for reads**: Flash reads are more important than flash writes because the former needs to be served before their deadline while the latter is not time critical. Therefore, flash reads should have a higher priority than flash writes when they compete for resources. Both flashcache and ZFS do not consider this point.

**Recommendation 4**: **Object-level caching**: Cache replacement policies such as LRU or its variants which operate at the block level (e.g., single page) have been successfully used for the OS buffer cache. However, caching at such a fine granularity is not appropriate for flash memory since it would trigger frequent small writes to the flash device that are detrimental to performance. For a given price, flash provides a much larger capacity than RAM; therefore, we can increase the granularity of caching for video objects. For example, with a 60 GB flash, we could cache up to 56 most popular video objects in the the flash memory. Assuming a 0.271 zipf distribution, 55.4% of the video accesses could be served out of the flash memory. This hit ratio is much better than the hit ratios which we observed on flashcache (25.7%) and ZFS (11.1%). Moreover, a conservative cache replacement based on the long term history of video access frequency can reduce the amount of write operations to the flash, and consequently, it can lower the chance of interference with the read operations serving videos from the flash.

## 6. CONCLUSIONS

Due to the cost and size advantage of flash memory compared to DRAM, a multi-tiered storage hierarchy, wherein a flash-based SSD serves as an intermediate level cache appears to be an attractive strategy for constructing a cost-efficient high performance video streaming server. However, we found through extensive performance studies that two state-of-the-art multi-tiered storage systems exhibited disappointingly poor performance for HTTP video streaming using the DASH protocols. We performed careful analysis to uncover the sources of poor performance in both these systems. Based on our analysis, we have recommendations for constructing an HTTP streaming server that avoids the pitfalls in designing such a server. Our future work includes building such a server embodying these design principles and carrying out experimental studies to validate these design ideas.

## 7. REFERENCES

[1] Apache. `http://httpd.apache.org`.
[2] Blktrace. `http://linux.die.net/man/8/blktrace`.
[3] DASH dataset. `http://www-itec.uni-klu.ac.at/dash/?page_id=207`.
[4] Ext4 file system. `https://ext4.wiki.kernel.org`.
[5] Flashcache. `http://www.facebook.com/note.php?note_id=388112370932`.
[6] Http/1.1. `http://www.ietf.org/rfc/rfc2068.txt`.
[7] Hulu. `http://www.hulu.com`.
[8] ISO/IEC DIS 23009-1.2. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57623`.
[9] Netflix. `http://www.netflix.com`.
[10] Zettabyte file system. `http://solaris-training.com/classp/200_HTML/docs/zfs_wp.pdf`.
[11] ZFS L2ARC. `https://blogs.oracle.com/brendan/entry/test`.
[12] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.
[13] A. Barbir, B. Cain, R. Nair, and O. Spatscheck. Known content network (cn) request-routing mechanisms. RFC 3568, `http://tools.ietf.org/html/rfc3568`.
[14] Intel Corporation. Understanding the Flash Translation Layer (FTL) Specification. White Paper, `http://www.embeddedfreebsd.org/Documents/Intel-FTL.pdf`, 1998.
[15] A. Leventhal. Flash storage memory. *Communications of the ACM*, 51(7):47–51, July 2008.
[16] T. R. G. Nair and P. Jayarekha. A rank based replacement policy for multimedia server cache using zipf-like law. *Journal of Computing*, 2(3):14–22, 2010.
[17] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim. A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications. *Trans. on Embedded Computing Sys.*, 7(4):1–23, 2008.
[18] C. Partridge, T. Mendez, and W. Milliken. Host anycasting services. RFC 1546, `http://tools.ietf.org/html/rfc1546`.
[19] M. Ryu, H. Kim, and U. Ramachandran. Impact of flash memory on video-on-demand storage: Analysis of tradeoffs. In *Proceedings of the ACM Multimedia Systems*, San Jose, CA, USA, February 2011.