

# FlashStream: A Multi-tiered Storage Architecture for Adaptive HTTP Streaming

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

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

## ABSTRACT

Video streaming on the Internet is popular and the need to store and stream video content using CDNs is continually on the rise thanks to services such as Hulu and Netflix. Adaptive HTTP streaming using the deployed CDN infrastructure has become the de facto standard for meeting the increasing demand for video streaming on the Internet. The storage architecture that is used for storing and streaming the video content is the focus of this study. Hard-disk as the storage medium has been the norm for enterprise-class storage servers for the longest time. More recently, multi-tiered storage servers (incorporating SSDs) such as Sun's ZFS and Facebook's flashcache offer an alternative to disk-based storage servers for enterprise applications. Both these systems use the SSD as a cache between the DRAM and the hard disk. The thesis of our work is that the current-state-of-the-art in multi-tiered storage systems, architected for general-purpose enterprise workloads, do not cater to the unique needs of adaptive HTTP streaming. We present *FlashStream*, a multi-tiered storage architecture that addresses the unique needs of adaptive HTTP streaming. Like ZFS and flashcache, it also incorporates SSDs as a cache between the DRAM and the hard disk. The key architectural elements of FlashStream include optimal write granularity to overcome the write amplification effect of flash memory SSDs and a QoS-sensitive caching strategy that monitors the activity of the flash memory SSDs to ensure that video streaming performance is not hampered by the caching activity. We have implemented FlashStream and experimentally compared it with ZFS and flashcache for adaptive HTTP streaming workloads. We show that FlashStream outperforms both these systems for the same hardware configuration. Specifically, it is better by a factor of two compared to its nearest competitor, namely ZFS. In addition, we have compared FlashStream with a traditional two-level storage architecture (DRAM + HDDs), and have shown that, for the same investment cost, FlashStream provides 33% better performance and 94% better energy efficiency.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

MM'13, October 21–25, 2013, Barcelona, Spain.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

Copyright 2013 ACM 978-1-4503-2404-5/13/10 ...\$15.00.

<http://dx.doi.org/10.1145/2502081.2502122>

## Categories and Subject Descriptors

D.4.2 [Operating Systems]: Storage Management—*Storage hierarchies*; H.5.1 [Information Interfaces And Presentation]: Multimedia Information Systems—*Video*

## General Terms

Design, Performance, Measurement

## Keywords

Video-on-Demand, Storage, Flash Memory, Solid-State Drive, Adaptive HTTP streaming

## 1. INTRODUCTION

Hulu [3] a web service that streams premium contents such as news, TV series, and shows, and Netflix [6] the largest subscription service for DVD rental and streaming video over the web are huge successes. In November 2011, Hulu had more than 30 million unique viewers in the U.S. [4]. Netflix currently has 20 million subscribers in the U.S., and they consume 30% of North American Internet traffic during the peak time [7].

Adaptive HTTP streaming (AHS) is a new paradigm of video streaming over the web currently being used by major content distributors, including Netflix. AHS does not depend on specialized video servers. Instead, AHS 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.

In the 90's and early 2000's, Internet-based video streaming for high-bitrate video was challenging due to technological limitations. In that era, every component of the video server needed to be carefully optimized to meet the volume of clients, the data rate of video, and the real-time guarantees for jitter-free video delivery. For example, real-time scheduling was needed to allocate precious CPU, disk, and network resources for the simultaneous video streams; efficient memory caching was needed to ensure that the disks are not overloaded; data placement strategy over multiple disks was needed to evenly distribute the disk load; and

an admission control mechanism was needed to ensure that quality of service guarantees can be met without overloading the various system components. However, most of these software solutions have become less important with the remarkable hardware improvements over the past two decades. There is  $100\times$  speedup in CPU speeds; RAM capacity has increased by  $1,000\times$ ; hard disk drive (HDD) capacity has grown by  $10,000\times$ ; and network bandwidth has improved  $10,000\times$ . The only thing that has been stagnant is the access latency of the HDD. The access latency has *only* improved four times over the last two decades! [2]

The video streaming system requires a number of 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 bit-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 user generated content (UGC) and the corresponding number of viewers are increasing explosively on the web. In addition, the AHS 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 researchers' attention again for more throughput, lower power consumption, and lower cooling costs.

Solid-State Drive (SSD) is a new storage technology 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 also 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. One interesting characteristic of video service is that only a small number of videos relative to the entire collection is accessed frequently. Therefore, we can serve frequently accessed hot contents using flash-based SSDs, serve cold contents by slow HDDs, and store the entire video library in HDDs. In this way, we can build a high-performance Video-on-Demand (VoD) system cost-effectively.

Recently, Facebook introduced flashcache, and Sun Solaris introduced ZFS, both of which are multi-tiered storage systems using SSDs as an intermediate cache between the DRAM and the hard disk. Our prior study [27] analyzing these two systems for use as a storage server for adaptive HTTP streaming showed that both systems do not perform well and presented design guidelines for building a high-performance

multi-tiered video storage system using flash memory SSDs. The design guidelines can be summarized as follows. First, while low-end flash memory SSDs are well-suited to serve the caching needs of videos in an HTTP streaming server, it is of paramount importance to avoid small random writes to the flash memory SSDs due to the unique performance characteristic of flash memory. Second, SSD write operations should not be in the critical path of serving missed video segments (brought from the hard disk) to the clients. Third, SSD read operations should have higher priority than SSD write operations because the former needs to be served before their deadline to the clients, while the latter is not time critical.

In this paper, we propose *FlashStream*, a multi-tiered storage architecture incorporating flash memory SSDs that caters to the needs of adaptive HTTP streaming. The unique contributions of our work are the following:

1. To minimize write amplification, writes to SSDs are always requested at the granularity of an optimal block size and aligned on the optimal block size. In addition, we present a novel reverse-engineering technique to find out the optimal block size for any flash-based SSDs. This accomplishes the first design guideline.
2. Utilization-aware SSD admission and ring buffer mechanisms control the write request rate to SSDs and give higher priority to read requests. We adopt ZFS's evict-ahead policy to avoid SSD write operations from the critical path of serving missed data. This fulfills the second and third design guidelines.
3. Metadata for video segments are embedded in the SSD blocks to quickly restore data in SSD upon power failure.

We have built FlashStream as a web server based on pioneer [9] open source network library. We have experimentally evaluated the performance of FlashStream and compared it to both ZFS and flashcache. We show that FlashStream outperforms both these systems for the same hardware configuration.

The rest of the paper is organized as follows. Section 2 explains the background about flash memory SSDs. We describe the FlashStream architecture for adaptive HTTP streaming using flash memory SSDs in Section 3, and discuss our implementation in Section 4. Section 5 presents extensive experimental results comparing FlashStream with other systems. Related work is presented in Section 6 and the final section presents our concluding remarks.

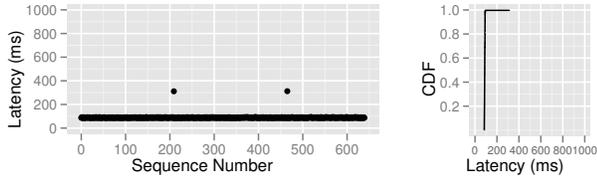
## 2. BACKGROUND

In this section, we briefly review a technology central to the problem being addressed in this paper: a Flash Memory SSD.

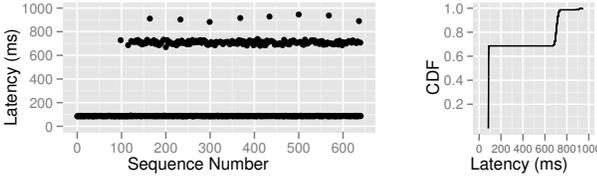
### 2.1 Flash Memory SSD

Flash memories, including NAND and NOR types, have a common physical restriction, namely, they must be erased before being written [11]. 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



(a) Latency sequence for sequential writes (b) Latency CDF for sequential writes



(c) Latency sequence for random writes (d) Latency CDF for random writes

**Figure 1: Latency distribution for sequential/random writes with 8 MB request size. OCZ Core V2 is used for the measurement.**

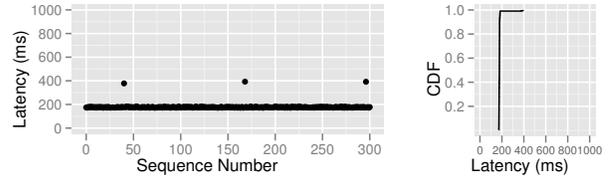
block is 100,000 but is only 10,000 in two-bit MLC NAND flash memory.

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) [16, 17, 24]. 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.

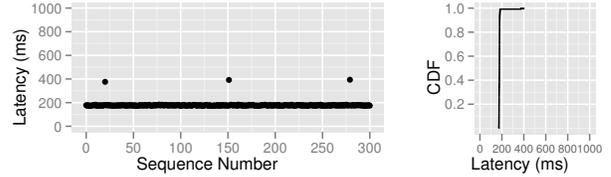
To avoid erasing and re-writing an entire block for every page modification, an FTL writes data out-of-place, remapping the logical page to a new physical location and marking the old page invalid. This requires maintaining some amount of free blocks into which new pages can be written. These free blocks 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*.

### 3. FLASHSTREAM ARCHITECTURE

We build on our prior work reported in [27] wherein we presented design guidelines for building a high-performance multi-tiered video storage system using flash memory SSDs. Keeping the guidelines in mind, we describe the architecture of *FlashStream* in the following subsections. We use MLC SSDs instead of SLC because the SSDs are used as cache in FlashStream, not for permanent storage. Therefore, we do not need the better reliability of SLC. Moreover, MLC is significantly cheaper than SLC.



(a) Latency sequence for sequential writes (b) Latency CDF for sequential writes



(c) Latency sequence for random writes (d) Latency CDF for random writes

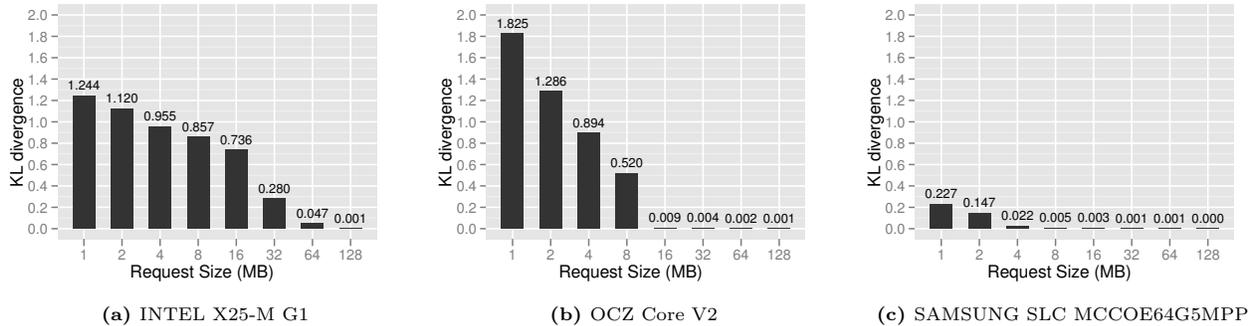
**Figure 2: Latency distribution for sequential/random writes with 16 MB request size. OCZ Core V2 is used for the measurement.**

### 3.1 Optimal Block Size

As we discussed in Section 2.1, the reasons for the poor performance of flash memory for small random writes are two fold: (a) a block has to be erased before a page within it can be written, and (b) an erase block consists of several pages, valid pages in the block being erased must be copied to other clean pages before erasing the block (*write amplification*), and page write and block erasure are inherently slow operations. Due to the write amplification effect, small random writes not only degrade the SSD performance but also reduces the flash memory lifetime.

One way to completely eliminate both “write amplification” and poor performance due to random page writes is to ensure that write requests to the SSD are always in multiples of the block size of the flash memory that is used in the SSD, and that the writes are aligned on block boundaries. This strategy will also have the additional benefit of fast garbage collection (due to the elimination of write amplification) and longer flash memory lifetime. However, there is a catch. SSD manufacturers put great efforts to fully exploit all available hardware resources and features to improve performance, and they use different designs [11]. In addition, the internal organization of an SSD and their techniques are proprietary and not readily available. How do we choose the right write granularity in the software architecture of the storage system when we do not know the internal architecture of the SSD? To this end, we define the *optimal block size* for write requests to the SSD as the minimum write size that makes the random write performance similar to the sequential write performance.

To determine the optimal block size, we do a reverse-engineering experiment on any given flash-based SSD. The idea is to perform random writes with different request sizes until the performance matches that with a sequential workload. To illustrate the technique, we use an OCZ Core V2 flash-based SSD. Figure 1 shows the latency of 640 write operations for sequential and random workloads when the request size is 8 MB on this SSD. For the random writes (see Figure 1(c)), we notice frequent high latencies (i.e., 800-1000 ms) due to the write amplification effect. On the other hand, when the request size is 16 MB (see Figure 2), the latency distributions



**Figure 3: KL divergence values for different request sizes and SSDs. The optimal block size for INTEL SSD is 64 MB, that for OCZ SSD is 16 MB, and that for SAMSUNG SSD is 4 MB.**

for the sequential write and the random write workloads are very similar. Figures 1(b) and 1(d) show the CDF graphs for the 8 MB request size; Figures 2(b) and 2(d) show the CDF graphs for the 16 MB request size. While the CDF graphs are useful to visually see the similarity of two distributions, we need a quantitative measure to determine the similarity. Kullback-Leibler (KL) divergence [19] is a fundamental equation from information theory that quantifies the proximity of two probability distributions: the smaller the KL value the more similar the two distributions. In flash memory SSDs, write operation latency is a random variable and hard to estimate because it is affected by various factors such as mapping status, garbage collection, and wear-leveling. We use the KL divergence value as the metric for comparing the similarity of the latency distributions obtained with the sequential and random write workloads.

Figure 3 shows how the KL divergence value changes with different request sizes for three different SSDs. For all the SSDs, the KL divergence value converges to zero as the request size increases. We let the optimal block size be the request size where the KL divergence value becomes lower than a threshold that is close to zero. The threshold is a configuration parameter. A lower threshold can give a block size that shows more similar latency between sequential writes and random writes. We choose 0.1 for the threshold because Figure 3 shows the KL divergence value quickly converges to zero once it becomes lower than 0.1. Therefore, 64 MB is the optimal block size for INTEL X25-M G1, 16 MB for OCZ Core V2, and 4 MB for SAMSUNG SLC MCCOE64G5MPP. The optimal block size is a function of the internal architecture of the SSD (page size, block size, available internal parallelism for access to the flash memory chips, etc.). However, this experimental methodology allows us to reverse-engineer and pick a write size that is most appropriate for a given SSD. Once the optimal block size has been determined, FlashStream divides the logical address space of flash memory into identical allocation blocks that are of the same size as the optimal block size determined with the experimental methodology.

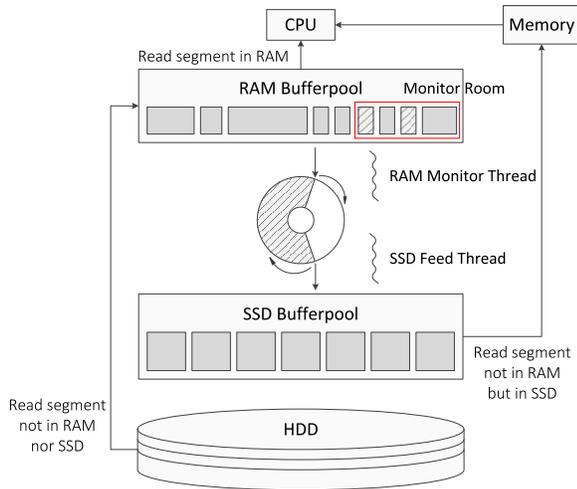
## 3.2 System Design

In adaptive HTTP streaming systems, a segment is a basic unit for accessing videos. Accordingly, FlashStream organizes its storage structure at segment granularity. Figure 4 depicts the FlashStream architecture. The first cache layer is a RAM cache. The RAM buffer pool is an array of memory blocks, and each block stores data for a video segment

file. Because segments have different sizes, the size of each block in the RAM buffer pool is different. The second cache layer is an SSD cache. The SSD’s address space is divided into equal-sized blocks (same as the optimal block size). All write operations to the SSD are requested with the optimal block size aligned on the block boundary. All segment requests that miss the two caches are served by disks. The segments read from the disks are pushed into the RAM buffer pool, and then they are returned to the CPU. On the other hand, the segments read from the SSD are not inserted into the first-level cache (RAM buffer pool) but they are directly returned to the CPU (i.e., DMA’ed into its memory). In this way, we can maximally utilize the capacity of the first-level RAM cache and the second-level SSD cache because we avoid the same segment from being stored in both levels. One of our design goals is to minimize the I/O requests sent to the disks since the hard disk is the slowest element in the three-level hierarchy of FlashStream, and thus could hurt real-time streaming performance. By storing data either in the DRAM or the SSDs (but not both), FlashStream maximizes the cumulative cache hit ratio in the first two levels and minimizes the read requests sent to the disks.

### 3.2.1 RAM Buffer Pool

The RAM buffer pool is the first-level in the three level hierarchy of FlashStream. Segments cached in the buffer are ordered according to the replacement policy used. For example, when a least recently used (LRU) replacement policy is employed, the most recently used segment will be at the head of the buffer pool and the least recently used segment will be at the tail. LRU is the default replacement policy for the RAM buffer pool because it is simple and works well in most cases. However, it should be noted that other replacement policies (such as LRU-Min and GDS [26]) can be easily used for the RAM buffer pool without affecting the design decisions in the other components of FlashStream. Whenever a new segment is inserted into the pool and the pool has insufficient space, then segments at the end of the pool are evicted until the pool has space to host the new segment. A RAM monitor thread periodically monitors the tail part of the RAM buffer pool (Figure 4). This tail portion of the pool is called the *monitor room*. The size of the monitor room is a system configuration parameter, and we use 10% of the total number of segments in the RAM buffer pool for the monitor room size by default. When the thread finds segments in the monitor room that are not in the SSD cache, then those segments are copied to a ring



**Figure 4: FlashStream Architecture.** A finite-sized RAM buffer serves as the first-level cache of the server. SSD is the second-level cache. Upon a miss in the first-level cache, the data is served from the SSD (i.e., it is not copied into the first-level cache). A miss in both the caches results in reading the missing segment from the hard disk and into the first-level RAM buffer cache.

buffer. In Figure 4, such segments are shown as shaded squares in the monitor room. Segments that are in the ring buffer are candidates for being written into the second level cache. The monitor thread simply throws away the segments in the monitor room to make room for new segments coming from the disk even if they have not been copied into the ring buffer by the monitor thread. This is similar to the *evict-ahead policy* used in the ZFS file system. We assume that the video data is immutable and thus read-only (e.g., DB of movies stored at Netflix). When a video object is updated (e.g., new version of a movie is distributed to the CDN servers by the content provider), the cached copies of the segments pertaining to the old version of the video object will simply be removed from the memory hierarchy of FlashStream. This is an out-of-band administrative decision outside the normal operation of FlashStream and does not interfere with the *evict-ahead policy* for managing the buffer pool. The advantage of the *evict-ahead policy* is that it can avoid flash write operations from the critical path for handling cache miss [27]. On the other hand, the drawback of the *evict-ahead policy* is that there is no guarantee that the evicted segments will be written to the second level cache (i.e., SSD). When a segment misses both the RAM and SSD caches, it is read from the disks and placed in the RAM buffer pool. A thread that is inserting a segment into the RAM buffer pool does not have to wait for a victim segment (in the monitor room) to be copied into the ring buffer (for writing eventually to the SSD). The hope is that the victim would have already been copied into the ring buffer by the monitor thread *ahead of time*.

### 3.2.2 SSD Buffer Pool

Segments that are to be written to the SSD are buffered in the ring buffer between the RAM buffer pool and the SSD buffer pool. While the RAM monitor thread fills up the ring buffer, an SSD feed thread consumes the ring buffer and writes data to the SSD in units of the optimal block size

(Section 3.1). In addition, the SSD feed thread employs a simple admission mechanism. For a video streaming system, read operations have higher priority than write operations. Therefore, when the SSD is busy for serving read requests, the SSD feed thread should not make write requests to the SSD. Details of the admission mechanism are presented in Section 4.4. If the second-level cache (SSD) is full, the victim block chosen for eviction from the SSD is simply thrown away since the block is already present in the disk, and we are always dealing with read-only data in the storage server.

### 3.2.3 SSD Block Replacement Policies

A *Least Recently Used (LRU)* replacement policy and a *Least Frequently Used (LFU)* replacement policy are appropriate for evicting blocks from the SSD cache. However, there is an interesting design dilemma. Recall that our unit of management of the second-level SSD cache is the optimal block size of the SSD. Since segments can be of variable sizes, a single block in the SSD cache (of optimal block size) may hold multiple segment sizes. How do we update the LRU list of blocks when a segment that is a small part of a block is accessed? How do we maintain the LFU book-keeping for multiple segments that may be in the same SSD block? We suggest three different replacement policies:

1. *Least Recently Used Block (LRU)*: In this policy, a segment access is considered the same as an access to the containing SSD block. Therefore, when any segment in a block is accessed, the block is moved to the front of an LRU list. When a block needs to be evicted, the last block in the list is chosen as a victim.
2. *Least Frequently Used Block on Average (LFU-Mean)*: This scheme keeps track of the access count for each segment. When a block needs to be evicted, the average access count of the segments in a block is calculated, and the block with the least average count is chosen as a victim.
3. *Least Frequently Used Block based on Median (LFU-Median)*: Similar to the LFU-Mean policy, this scheme keeps track of the access count for each segment as well. Because the mean is not robust to outliers, this policy uses median of the access counts of the segments in a block.

The performance of these SSD block replacement policies are evaluated in Section 5.

### 3.2.4 Difference from ZFS

FlashStream employs an *evict-ahead policy* similar to the ZFS file system, however, there are fundamental differences between the two.

1. ZFS uses FIFO replacement policy for the SSD. The motivation is same as FlashStream; FIFO replacement policy generates sequential writes to the SSD and avoids small random writes to the SSD. However, the FIFO replacement policy of ZFS shows a low hit ratio. On the other hand, FlashStream employs different kinds of block replacement policies, and it shows a lot higher hit ratio than ZFS. The results are presented in Section 5.2.
2. ZFS does not differentiate the priority of writes and reads to/from the SSD because it is not designed for a video streaming system. Therefore, reads which would be hits in the SSD cache get queued up behind ongoing writes, and miss the deadline that reads need to be serviced. On the other hand, FlashStream uses an

Segment Id		
Video Id	Segment Number	Bitrate

**Table 1: Each video segment is uniquely identified by a tuple: {Video Id, Segment Number, Bitrate}.**

admission policy for writes to the SSD to give a higher priority for read requests.

- FlashStream decouples monitoring the RAM buffer and writing evicted segments in the ring buffer to the SSD. These two functions are inter-twined in ZFS, and it could lead to lost opportunity. Recall that the evict-ahead policy used in ZFS will simply throw away pages from the RAM buffer even if they have not been written to the SSD. Thus, if the RAM insertion rate from the disk is faster than the rate at which evicted pages can be written to the SSD, then a larger population of evicted pages will be thrown away. On the other hand, in FlashStream, the monitor thread and the SSD feed thread cooperate through the ring buffer and thus there is an opportunity for more evicted segments to be cached in the SSD (despite the evict-ahead policy) even during periods when the RAM insertion rate is higher than the SSD write throughput. The amount of time taken to fill the SSD cache for both systems is presented in Section 5.2.

## 4. IMPLEMENTATION

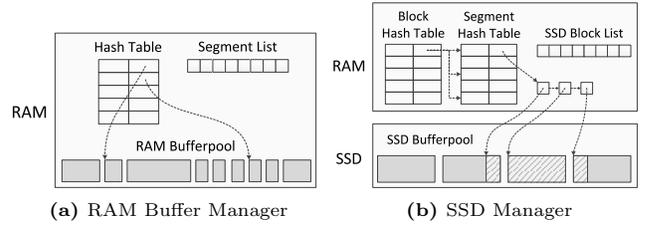
In this section, we describe the details of FlashStream implementation. We have built our FlashStream web server based on *pion-net* [9] open source network library. FlashStream directly manipulates a flash memory SSD as a raw device. In addition, FlashStream bypasses the operating system’s page cache and manages the RAM for caching segment files on its own. Each video segment is uniquely identified by a tuple: {video id, segment number, bitrate} (see Table 1). The tuple represents the id of a segment. Buffer Manager and SSD Manager are the two most important components, and their details are explained in the following subsections.

### 4.1 RAM Buffer Manager

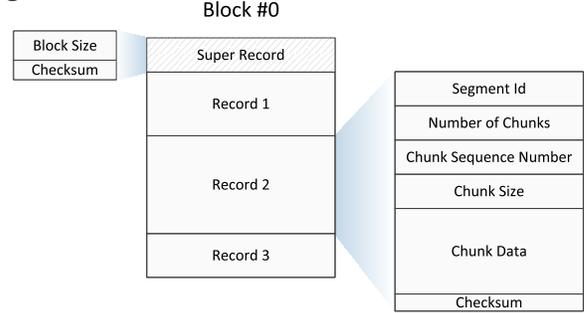
A buffer manager manipulates the available RAM buffer for caching segments. Figure 5(a) shows data structures used by the buffer manager. It keeps a pool of cached segments in RAM. For fast lookup, a hash table (key, value store) is maintained in RAM. The key is a segment id, and the value is a tuple: {memory address of the block in RAM, size of the block}. The segment list maintains a list of segment ids for victim selection. The buffer manager uses an LRU replacement policy. A hit in the RAM results in the segment being moved to the head of the segment list. The last element of the segment list is a victim. When a new segment read from the disk is inserted to the RAM buffer pool, a new entry is inserted into the hash table, and the id of the new segment is inserted at the head of the segment list. If needed, a victim is evicted from the buffer pool and the entry for the victim in the hash table is removed.

### 4.2 SSD Manager

An SSD manager manipulates the available SSD blocks for caching segments. Figure 5(b) shows the data structures used by the SSD manager. SSD’s address space is divided into equal-size blocks, and the block is same as the optimal block size (see Section 3.1). A block hash table (another key, value store) is maintained to lookup the segments that



**Figure 5: Data structures used by the RAM Buffer Manager and the SSD Manager. Replacement from the RAM buffer is in units of variable-sized video segments.**



**Figure 6: Data structure of an SSD block: An SSD block is composed of records, with each record containing the metadata as well as the data pertaining to the segment id. Block #0 (the first block) of the SSD is special in that it contains a super record in addition to the data records.**

are stored in a block. The key is an SSD block number, and the value is a list of segment ids that are stored in the same block. A segment hash table is used to lookup the logical block address of the SSD for a segment. When the segment size is larger than the size of an SSD block, then the segment is stored using multiple blocks. In addition, a part of a segment can be written at the end of a block and the rest can be written in the front of another block. We call such partial data of a segment a *chunk* of the segment. Therefore, the value field in the segment hash table is a list of tuples; each tuple contains {SSD block number, offset into the block, size of a segment chunk}. In Figure 5(b), the shaded blocks in the SSD buffer pool represents a single segment stored in multiple SSD blocks. An SSD block list maintains a list of block numbers for victim selection. The block hash table, the segment hash table, and the SSD block list data structures are stored in physical memory (i.e., RAM) for fast access.

### 4.3 SSD Block Data Layout

Flash memory SSDs provide capacity from hundreds of gigabytes to a few terabytes in a single drive these days. After a system crash or administrative downtime, warming up such a large capacity SSD takes a long time. Since flash memory is non-volatile, a warmed up SSD can still serve as a second-level cache despite power failures as long as we have metadata for the segments stored in the SSD as well. To recover the stored data, FlashStream embeds metadata in the SSD blocks. Figure 6 shows the data layout in the SSD blocks. The first SSD block (block number 0) has a special record, namely, *super record*. The super record includes information about the block size and its checksum. Only the first

	SSD A	SSD B
Model	INTEL X25-M G1	OCZ Core V2
Capacity	80 GB	120 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 2: MLC Flash Memory SSDs that are used for our experiments. SSD A shows significantly better random write performance than SSD B. We intentionally use a very low-end SSD (SSD B) to show how well FlashStream works with low-end SSDs.**

SSD block has the super record, and the other SSD blocks do not. Following the super record, a series of records are stored. Each record represents a chunk of a segment. The record consists of *segment id*, *number of chunks*, *chunk sequence number*, *chunk size*, *chunk data*, and *checksum*. From the segment id, the system can figure out the segment that the chunk belongs to. The number of chunks tells how many chunks the segment is divided into, and the chunk sequence number tells the order of the chunk in its segment. Checksum is used to verify the integrity of each record. After system crash, by scanning sequentially all SSD blocks, FlashStream can reconstruct the in-memory data structures of the SSD manager (i.e., block hash table and segment hash table).

#### 4.4 Utilization-Aware SSD Admission

The SSD feed thread employs an admission mechanism to avoid writes to the SSD when the SSD is busy for serving reads. To measure how busy the SSD device is, we use `/proc/diskstats` wherein the Linux kernel reports information for each of the storage devices. `/proc/diskstats` provides the amount of time (in milliseconds) spent doing I/O for each storage device. We read the number at the beginning of a certain epoch and again at the end of the epoch. Because this is an accumulated number, the difference between the two readings gives the time spent in I/O for that device for the current epoch. The ratio of the time spent in I/O to the periodicity of the epoch serves as a measure of the storage device utilization for that epoch. The SSD feed thread stops writing data to the SSD when the SSD utilization exceeds a threshold  $\lambda$ , and the default value for the threshold is 60%.

## 5. EVALUATION

In this section, we evaluate the performance of the FlashStream system. We would like to mimic the request pattern that is fielded and serviced by an HTTP server (i.e., a CDN server) that is serving video segments to distributed clients under the adaptive HTTP streaming mechanism. To this end, the workload offered to the FlashStream server is a large number of independent requests from clients. To measure the storage subsystem performance exactly and to avoid network subsystem effects, the workload generator program and the FlashStream server run on the same machine communicating via a loop-back interface. The machine has Xeon 2.26 GHz Quad core processor with 8 GB DRAM, and Linux kernel 2.6.32 is installed on it. We use a 7200 RPM HDD, and the capacity of the HDD is 550 GB. We have evaluated five system configurations. In addition to our FlashStream system with three SSD cache replacement policies, we have measured the performance of two state-of-the-art multi-tiered storage systems (that use flash

Configuration	Web Server	File System
FlashStream(LRU)	FlashStream	EXT4
FlashStream(LFU-Mean)	FlashStream	EXT4
FlashStream(LFU-Median)	FlashStream	EXT4
ZFS	Apache	ZFS
Flashcache	Apache	EXT4

**Table 3: Five system configurations used for our evaluation.**

memory caching), namely, ZFS [10] and flashcache [1]. Table 3 shows the web server and the file system that are used for each system configuration. The version of Apache web server is 2.2.14, and the version of ZFS is `zfs-fuse 0.6.0-1`. The flash memory SSDs we used for our experiments are listed in Table 2. Note that we intentionally use a very low-end SSD (SSD B) to show how well FlashStream works with low-end SSDs. In our evaluation, FlashStream and ZFS have worked slightly better with SSD B while Flashcache has worked better with SSD A. Therefore, we present only results of FlashStream and ZFS used with SSD B and results of flashcache used with SSD A. We use only 75 GB out of the total capacity for both SSDs for fair comparison.

### 5.1 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 [23]. We use 0.2 for the parameter in our study. Dynamic Adaptive Streaming over HTTP (DASH) [5] is an ISO/IEC MPEG standard of the adaptive HTTP streaming paradigm. We use a DASH dataset [20] for our test which is available in the public domain. As shown in Table 4, the dataset comprises six distinct video objects with five different bitrates for each video. However, for our experimentation we needed to create a larger dataset comprising of 2000 distinct videos. We have generated the 2000 videos by uniformly using the six videos and five bitrates. The total size of our dataset is 545 GB. Since we use five different bitrates and videos are encoded with variable bitrate (VBR), the segments of our generated dataset have very different sizes.

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 a uniform distribution; each segment of the video object has the same probability. The reason behind the uniform distribution is as follows. A large scale adaptive 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 a request routing technique. There are many different request routing techniques such as DNS routing, HTML rewriting [12], and anycasting [25]. For this reason, there is no guarantee that a client who downloaded a segment  $i$  of a video will download the next segment  $i+1$  from the same server. The next segment request may be directed to a different web server that holds a replica. Therefore, it is reasonable to assume that the probability that a web server gets a request for a segment of a video object is totally random.

The workload generator sends an HTTP request for the chosen segment to the web server. When the segment request is

Name	Bitrates	Segment Length	Source Quality	Length	Genre
Big Buck Bunny	200/400/700/1500/2000 Kbps	10 sec	1080p YUV	09:46	Animation
Elephants Dream	200/400/700/1500/2000 Kbps	10 sec	1080p YUV	10:54	Animation
Red Bull Playstreets	200/400/700/1500/2000 Kbps	10 sec	1080p 6Mbit AVC	01:37:28	Sport
The Swiss Account	200/400/700/1500/2000 Kbps	10 sec	1080p 6Mbit AVC	57:34	Sport
Valkaama	200/400/700/1400/2000 Kbps	10 sec	1080p 6Mbit AVC	01:33:05	Movie
Of Forest and Men	200/400/700/1400/2000 Kbps	10 sec	SD	10:53	Movie

Table 4: Source DASH dataset that is used to generate our 2000 videos.

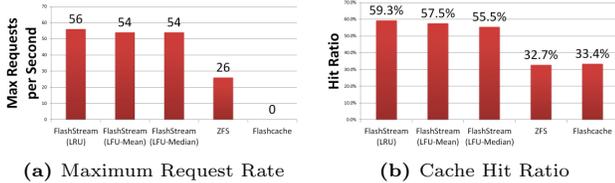


Figure 7: Maximum request rate and SSD cache hit ratio of the 5 different system configurations with a warmed up cache for 0% segment miss ratio. FlashStream performs 2 times better than ZFS.

not satisfied (i.e., it is not downloaded to the client) within the segment’s play-out time (10 seconds for our test video), the client counts it as a segment deadline miss. We choose the 10 second segment size because it is the default size used by Apple HTTP Live Streaming [13]. 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.

## 5.2 Performance Comparison

In this section, we measure the maximum request rate that the five systems can support when the cache is warmed up. This is the best performance each of the systems can provide. We have fully filled up the SSD cache by running the workload generator for a sufficient amount of time before each measurement. FlashStream (regardless of replacement policy) takes 94 minutes to fully fill up the 75 GB SSD cache while ZFS takes 1200 minutes (20 hours) and flashcache takes 1320 minutes (22 hours) for the same amount of SSD cache. We assume the QoS requirement for video streaming is 0% segment miss ratio. By increasing the request rate gradually, we measure the segment miss ratio for each request rate until the segment miss ratio gets higher than 0%. In this way, we can get the maximum request rate for 0% segment miss ratio. We run each measurement with a different request rate for an hour.

Figure 7(a) shows that FlashStream performs two times better than ZFS. Different SSD cache replacement policies of FlashStream perform similarly while LRU is slightly better than the other two. ZFS is able to support a maximum of 26 requests per second, and flashcache cannot support even 1 request per second.

Next, we measure the SSD cache hit ratio of the five systems with a warmed up cache when the request rate is the maximum request rate that the systems can support. Each measurement is run for an hour. Figure 7(b) shows that FlashStream with LRU SSD cache replacement policy has

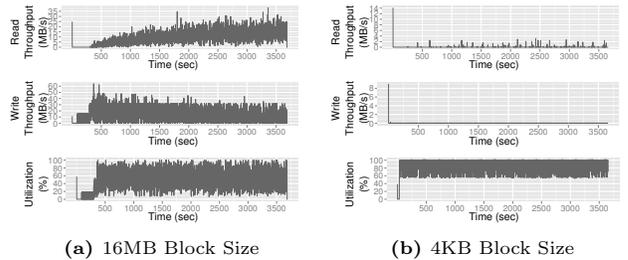


Figure 8: Effect of different block sizes on Read throughput, Write throughput, and Utilization of the SSD (second-level cache of FlashStream). When the block size matches the optimal block size (Figure 8(a)), the SSD is not overloaded (i.e., there is no write amplification), and the system fills segments gradually, and serves the segments that hit the SSD. When the block size is 4KB (Figure 8(b)), the SSD is overloaded (due to write amplification), and shows very low read and write throughput.

the best hit ratio. The other two policies (i.e., LFU-Mean and LFU-Median) show slightly lower hit ratio than LRU. ZFS’s lower maximum request rate compared to FlashStream correlates directly with ZFS’s lower SSD cache hit ratio. Page reads that miss the SSD cache of ZFS go to disks, makes the disks overloaded, and the reads from the disks miss their deadline. This result tells us, for the SSD cache, block-level replacement policies (that FlashStream uses) are able to achieve a higher hit ratio than FIFO replacement policy (that ZFS uses), and results in a higher maximum request rate. The reason for the poor performance of flashcache is due to the fact that SSD write operations are in the critical path of serving read operations that miss the SSD cache. Thus though the observed hit ratio for flashcache is 33.4%, it does not translate to performance (as measured by the maximum request rate) due to the fact that SSD cache reads could be backed up behind long latency SSD cache write operations. FlashStream and ZFS systems address this problem by the evict-ahead mechanism (see Section 3.2.1). For a more detailed analysis of the poor performance of ZFS and flashcache, please refer to our prior study [27].

## 5.3 Effect of Block Size

In this section, we measure how the choice of the SSD block size affects the FlashStream performance. We run FlashStream with a cold cache for an hour using SSD B. Even though the SSD cache is empty, for this experiment, we intentionally make FlashStream return random free blocks (not in a sequential order) when the SSD feed thread needs to write a block of data to the SSD. It is because we want to see quickly the effect of the SSD writes with random offsets without fully filling up the SSD cache. This is for a quick

Configuration	DRAM	SSD	HDD	RPS	Requests Joule
FlashStream	2GB	100GB	1	64	2.91
Traditional A	12GB	-	1	34	0.57
Traditional B	2GB	-	2	48	1.50

**Table 5: Three system configurations for energy efficiency comparison. SSD is a low-end SSD and HDD is 7200 RPM. RPS represents the maximum request rate.**

measurement and it does not harm the correctness of the results. As we have shown in Figure 3, the optimal block size of SSD B is 16MB. Figure 8(a) shows read throughput, write throughput, and utilization (top to bottom) of the SSD for an hour. From the write throughput graph (middle), we notice that the system gradually fills segments into the SSD cache. The read throughput graph (top) shows that the read throughput increases as segment requests hit the SSD cache. The utilization graph (bottom) tells us that the SSD device is not overloaded during this period.

On the other hand, Figure 8(b) presents the same SSD’s read throughput, write throughput, and utilization when the block size is 4KB. SSD B is a low-end SSD that shows very low performance with small random writes (see Table 2). Due to the write amplification triggered by the small block writes with random offsets, the SSD is overloaded during the whole period (100% utilization), and hence provides very low read and write throughput.

When the block size is 16MB, during the one hour period, 70.4% of the total capacity of the SSD is filled, and the SSD hit ratio is 34.4%. On the contrary, when the block size is 4KB, only 0.08% of the total SSD capacity is written and the SSD hit ratio is 0.65%. To make matters worse, 3% of the read requests that hit the SSD miss their deadline due to the poor read throughput. This experiment underscores the importance of using the optimal block size for the SSD to achieve good performance.

## 5.4 Energy Efficiency

In this section, we compare the energy efficiency for three different systems: FlashStream and two “traditional” systems (one for more DRAM and another for more disks). Table 5 shows the hardware configuration of the three systems. FlashStream has 2 GB DRAM, 100 GB SSD, and a single 7200 RPM 1 TB HDD. Traditional A has 12 GB DRAM and a single 7200 RPM 1 TB HDD. Traditional B has 2 GB DRAM and two 7200 RPM 1 TB HDDs striped per Linux’s software RAID-0 configuration. Compared to FlashStream, Traditional A uses more DRAM for caching instead of SSDs, while Traditional B uses more HDDs for more disk throughput via parallel access. All three systems have the similar total capital cost according to the capital cost of devices in Table 6 (Traditional B is \$10 more expensive than the other two). Apache web server is used for traditional systems. Table 6 shows the energy cost of each device in the active state. Each DRAM module in our test machine has 2 GB; 12 GB DRAM corresponds to 6 DIMMs. For FlashStream, we measure the maximum request rate after the SSD is filled up. For traditional systems, we warm up for an hour that is sufficiently long to fill up the given DRAM. We use the same workload as in Section 5.1.

Table 5 shows that FlashStream achieves the best energy efficiency. FlashStream serves 64 requests per second with 0% segment miss ratio and its energy efficiency is 2.91 requests per joule. For Traditional configurations, using additional HDD (i.e., Traditional B) is better than using more

	Capital Cost	Energy Cost
DRAM	\$8/GB	8W/DIMM
Low-end SSD	\$0.8/GB	2W/drive
7200RPM 1TB HDD	\$90/drive	12W/drive

**Table 6: Capital cost and energy cost of devices. Data comes from the specification of commodity DRAM, SSDs, and HDDs that are commercially available [8].**

DRAM (i.e., Traditional A). FlashStream is 94% more energy efficient than Traditional B. Moreover, Traditional B wastes disk capacity because it uses more HDDs only for more throughput; the dataset size is 545 GB while Traditional B has 2 TB HDDs in total. In addition, we ignore the cooling cost and the cost associated with rack space in a data-center. FlashStream that uses less DRAM and HDDs but more flash-based SSDs would generate less heat and occupy less volume in a rack. We do not consider these benefits in this analysis.

## 6. RELATED WORK

FlashStore [14] is a high throughput key-value storage that uses flash memory as a cache between the RAM and the hard disk. Kgil, et al. [18] have studied energy efficient web server using flash memory as an extended system memory. Their work is based on simulations with text and image workloads. Lee, et al. [22, 21] have researched the application of flash memory SSD as a database server. The authors claim that a single enterprise class SSD can be on par with or far better than a dozen hard drives with respect to transaction throughput, cost effectiveness, and energy consumption. Do, et al. [15] have proposed three different design alternatives that use an SSD to improve the performance of a database server’s buffer manager. They empirically have shown that they could speedup 3-9 times than the default HDD configuration depending on system parameters. Singleton et al. [28] have shown how much power can be saved when flash memory is used as a write buffer along with hard disks for mobile multimedia systems.

Different from the all previous works, FlashStream shows how to construct a high-performance and energy efficient multi-tiered storage system for adaptive HTTP streaming through the judicious use of flash memory SSDs as an intermediate level in the storage hierarchy.

## 7. CONCLUSIONS

In this paper, we have proposed a set of guidelines for effectively incorporating flash memory SSDs for building a high throughput and power efficient multi-tiered storage system for adaptive HTTP streaming. We have implemented FlashStream based on these guidelines and have evaluated its performance using realistic workloads. We have compared the performance of FlashStream to two other system configurations: Sun’s ZFS, and Facebook’s flashcache. Similar to FlashStream, the two configurations also incorporate flash memory SSDs in their respective storage hierarchy. FlashStream performs twice as well as its closest competitor, namely, the ZFS configuration, using segment miss ratio as the figure of merit for comparison. In addition, we have compared FlashStream with a traditional two-level storage architecture (DRAM + HDDs), and have shown that, for the same investment cost, FlashStream provides 33% better performance and 94% better energy efficiency.

There are several avenues for future research building on our novel multi-tiered storage system with flash memory SSDs for adaptive HTTP streaming. Discrepancy of the caching granularity (i.e., optimal block size) and the access granularity (i.e., segment size) of the SSD cache opens up new research opportunities for effective cache replacement algorithms. We proposed three initial algorithms for the purpose, but more studies are warranted. We used storage device's utilization as an indicator for finding opportune moments to schedule caching related I/O requests to the SSD. We would like to explore other dynamic indicators as this technique requires the threshold value for device utilization to be chosen statically by a system administrator, which could lead to the under-utilization or overloading of the SSDs if the threshold is not chosen properly. Lastly, given the promise shown by FlashStream, we would like to develop a distributed video server using the multi-tiered storage approach. Such a server will open up a host of new research issues including initial placement of video objects, migration policies for load balancing, etc.

## 8. ACKNOWLEDGMENTS

We thank our shepherd Prof. Cheng-Hsin Hsu and the anonymous reviewers for their valuable and constructive comments to improve the quality of this paper.

## 9. REFERENCES

- [1] Flashcache. [http://www.facebook.com/note.php?note\\_id=388112370932](http://www.facebook.com/note.php?note_id=388112370932).
- [2] Hdd technology trends. <http://www.storagenewsletter.com/news/disk/hdd-technology-trends-ibm>.
- [3] Hulu. <http://www.hulu.com>.
- [4] Hulu viewers. [http://www.comscore.com/Press\\_Events/Press\\_Releases/2011/12/comScore\\_Releases\\_November\\_2011\\_U.S.\\_Online\\_Video\\_Rankings](http://www.comscore.com/Press_Events/Press_Releases/2011/12/comScore_Releases_November_2011_U.S._Online_Video_Rankings).
- [5] ISO/IEC DIS 23009-1.2. Information technology - Dynamic adaptive streaming over HTTP (DASH) - Part 1: Media presentation description and segment formats.
- [6] Netflix. <http://www.netflix.com>.
- [7] Netflix traffic. <http://www.techspot.com/news/46048-netflix-represents-327-of-north-americas-peak-web-traffic.html>.
- [8] Newegg. <http://www.newegg.com>.
- [9] pion-net. <http://www.pion.org/projects/pion-network-library>.
- [10] Zettabyte file system. [http://solaris-training.com/classp/200\\_HTML/docs/zfs\\_wp.pdf](http://solaris-training.com/classp/200_HTML/docs/zfs_wp.pdf).
- [11] 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.
- [12] A. Barbir, B. Cain, R. Nair, and O. Spatscheck. Known content network (cn) request-routing mechanisms. RFC 3568, <http://tools.ietf.org/html/rfc3568>.
- [13] A. C. Begen, T. Akgul, and M. Baugher. Watching video over the web: Part1: Streaming protocols. *IEEE Internet Computing*, 15(2):54–63, 2011.
- [14] B. Debnath, S. Sengupta, and J. Li. Flashstore: High throughput persistent key-value store. In *Proceedings of the 36th International Conference on Very Large Data Bases*, Singapore, September 2010.
- [15] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging dbms buffer pool using ssds. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1113–1124, 2011.
- [16] Intel Corporation. Understanding the Flash Translation Layer (FTL) Specification. White Paper, <http://www.embeddedfreebsd.org/Documents/Intel-FTL.pdf>, 1998.
- [17] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *USENIX Winter*, pages 155–164, 1995.
- [18] T. Kgil and T. Mudge. Flashcache: a nand flash memory file cache for low power web servers. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 103–112, 2006.
- [19] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, March 1951.
- [20] S. Lederer, C. Müller, and C. Timmerer. Dynamic adaptive streaming over http dataset. In *Proceedings of the third annual ACM conference on Multimedia Systems*, MMSys '12, pages 89–94, Chapel Hill, North Carolina, USA, February 2012.
- [21] S.-W. Lee, B. Moon, and C. Park. Advances in flash memory ssd technology for enterprise database applications. In *Proceedings of the ACM SIGMOD*, pages 863–870, June 2009.
- [22] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In *Proceedings of the ACM SIGMOD*, pages 1075–1086, June 2008.
- [23] 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.
- [24] 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.
- [25] C. Partridge, T. Mendez, and W. Milliken. Host anycasting services. RFC 1546, <http://tools.ietf.org/html/rfc1546>.
- [26] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, December 2003.
- [27] M. Ryu, H. Kim, and U. Ramachandran. Why are state-of-the-art flash-based multi-tiered storage systems performing poorly for http video streaming? In *Proceedings of the 22nd SIGMM Workshop on Network and Operating Systems Support for Digital Audio and Video*, Toronto, Ontario, Canada, June 2012.
- [28] L. Singleton, R. Nathuji, and K. Schwan. Flash on disk for low-power multimedia computing. In *Proceedings of the ACM Multimedia Computing and Networking Conference*, January 2007.