HyCache: A Hybrid User-Level File System with SSD Caching

Student Name: Dongfang Zhao
Advisor: Dr. Ioan Raicu
Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA

***Abstract*—For many decades, a huge performance gap has existed between volatile memory and mechanical hard disk drives. This will be a critical issue with extreme scale computing systems. Although non-volatile memory has been around since the 1990's, mechanical hard disk drives are still dominant due to large capacity and relatively low cost. We have designed and implemented HyCache, a user-level file system that leverages both mechanical hard disk drive cost-effectiveness and solid-state drive performance. We adopted FUSE to deliver a user-level POSIX-compliant file system that does not require any OS or application modifications. HyCache allows multiple devices to be used together to achieve better performance while keeping costs low. An extensive evaluation is performed using synthetic benchmarks as well as real world applications, which shows that HyCache can achieve up to 7X higher throughput and 76X higher IOPS over traditional Ext4 file systems on mechanical hard disk drives.**

I. Introduction

The mechanical spinning hard disk drive (HDD) has been the performance bottleneck of computer systems for decades. Their slow increase in bandwidth, slow decrease in latency, and exponential increase in capacity, have made modern storage devices quite unbalanced. Even though Solid State Drive (SSD) has been introduced for over a decade, HDDs are still dominant storage media in most systems because of their large capacities and low costs. Some high-end HDDs have up to 200 MB/s peak bandwidth, which is significantly smaller than main memory (RAM) bandwidths that range in the GB/s to tens of GB/s. Making matters worse is that the trends of HDD bandwidth and latency improvements are much smaller than the comparable speedup of other electronic counterparts [32] e.g. CPU still follows Moore's Law [8]. This phenomenon can be best explained by the fact that HDD has limitations imposed by physics with respect to its moving magnetic components, as opposed to other electronic components e.g. processors, memory, etc.

SSD can alleviate the bottleneck issue of HDD to some degree by offering a bandwidth up to several GB/s [26], but their high cost (e.g. 44X higher costs per GB as shown in Table I: $3.28 vs. $0.075) makes them impractical for many cost sensitive applications. Because of the high cost, SSD capacity is significantly smaller than HDD. At the point of this writing the largest HDD has 4 TB space from Hitachi [9] whereas mainstream SSD tops out at 960 GB from OCZ [26].

Recently a hybrid storage structure was proposed to combine SSD and HDD to take advantages of both types of drives. Seagate has just released a product Momentus XT [21] which encapsulates both SSD and HDD into a single physical drive a.k.a. hybrid hard disk (HHD). However the major drawback of this commercial product is its proprietary nature, and closed algorithms and architecture. For example, it is not possible to explicitly address the SSD and HDD portion within the device. In other words, it is not programmable to explicitly indicate which data to put on which partition i.e. SSD or HDD. Furthermore, the SSD is often small (e.g. 4GB) in relation to a large HDD (e.g. 500 GB to 1 TB). The small SSD portion typically have inexpensive and relatively slow controllers in order to keep the costs low. And these drives often limit the I/O operations that use the SSD cache (e.g. just read operations). These drawbacks limit the applicability to fully leverage the SSD advantages for superior performance at a low cost, for a large variety of workloads. More recently OCZ released Synapse [27] for a hybrid solution which has more limitations since it is only supported by Windows systems.

To get a more concrete impression of the aforementioned types of hard drives in the real world we list some key specifications of some mainstream commercial products in Table I. All pricing information are obtained from Newegg [22]. The first one is a high end SSD OCZ RevoDrive RVD3X2-FHPX4-960G with a price tag of $3,150. The second one is a mainstream SSD OCZ Octane OCT1-25SAT3-512G which costs $800. The third one was the first hybrid drive in the market released in 2011 from Seagate: Momentus XT, which consists of 4 GB SSD and 500 GB HDD for sale at $110. The last device is currently the largest HDD in the market: Hitachi Deskstar 7K4000 with a price tag of $350.

Even though SSD technology is not a perfect replacement for HDD, it could potentially serve as a cache buffer between RAM and HDD. First of all, its bandwidth lands just between the RAM and HDD, offering approximately one order of magnitude higher bandwidth than HDD and one order of magnitude lower than memory. Furthermore, SSDs can offer up to three orders of magnitude better latencies than HDD. If price is not a concern, SSDs are a great storage technology. However, if price does influence the storage technology, a hybrid architecture that uses both SSDs and HDDs could achieve a large percentage of the SSDs performance while at a cost that is only slightly more expensive than the HDD.

Adding a new level of SSD cache is not trivial because the caching scheme of the operating system will need to be modified. For most of applications and setups, it is not appropriate to modify the kernel of the operating system.

TABLE I
KEY SPECIFICATIONS OF SOME HARD DRIVES ON THE MARKET

Hard Drive	Unit Price (per GB)	Capacity (GB)	Read (MB/s)	Write (MB/s)	IOPS
OCZ RevoDrive	$3.28	960	1,500	1,300	230,000
OCZ Octane	$1.56	512	480	330	26,000
Seagate Momentus XT	$0.22	504	131	101	238
Hitachi Deskstar	$0.075	4,096	144	142	360

Moreover, modifying kernel will introduce other issues like portability, maintenance, security, cost, etc.

An alternative to avoid modifying the kernel is to only leverage SSD in the user space. For example, File System in User Space (FUSE) [6] is a framework to help develop customized file system in most of modern operating systems e.g. UNIX, Mac OS and Windows, etc. Over the years many user-level file systems ([1], [11], [13], [20], [23], [28], [31], [38]) have all adopted FUSE. FUSE module has been officially merged into the Linux kernel tree since kernel version 2.6.14 [18].

Deploying FUSE-based file systems on hybrid SSD+HDD offers two technical benefits. First, users can simply take advantages of SSD performance without privilege permissions. There would be no development or administration efforts needed to deploy the file system. It also circumvents possible security issues related to user authentications. Second, the file systems can be maintained effectively with failure transparency. This transparency is achieved by the fact that end users can only see the file system in the user space and the underlying physical storage is invisible. Therefore the failed physical storage devices can be replaced without affecting the end users.

In this paper we present the design and implementation of a file system in user space under FUSE framework to leverage the performance advantage of SSD in hybrid hard drives. In particular, the SSD is treated as a persistent cache between RAM and HDD. Our implementation is called HyCache, and it stands for Hybrid User-Level File System with SSD Caching. The purpose of HyCache is to allow users have a cost effective and faster file system without modifying the operating system kernel. The contribution of this work is twofold as summarized in the following:

1) *Design and implement a user-level POSIX-compliant file system with high throughput, low latency, single name space, and strong consistency.*
2) *Extensive performance evaluation showing that user-level file systems can be competitive with kernel-level file systems as well as embedded hybrid hard drive technologies.*

The structure of this paper is as follows. A brief introduction to FUSE framework is given in Section II. Section III describes the architecture of HyCache. Section IV details the implementation of HyCache. The experimental results of HyCache performance are presented in Section V. Section VI reviews some previous work on hybrid storage systems. Section VII concludes the paper and discusses our future plan.

II. FUSE

FUSE is a loadable kernel module for UNIX-like computer operating systems that lets non-privileged users create their own file systems without editing kernel code. As an example, Figure 1 (from Wikipedia [39]) shows the flow chart of a typical FUSE-based program. End users try to list all the files under directory /tmp/fuse. This request is sent to the kernel and caught by the FUSE kernel module because /tmp/fuse is a FUSE mount point. The kernel module then makes another context switch to call the user-defined executable ./hello in the user space. The executable, linked with the libfuse library, enters the kernel mode again to make the system call. Finally the result is returned to the caller (i.e. ls -l /tmp/fuse) in the user space to complete the request.

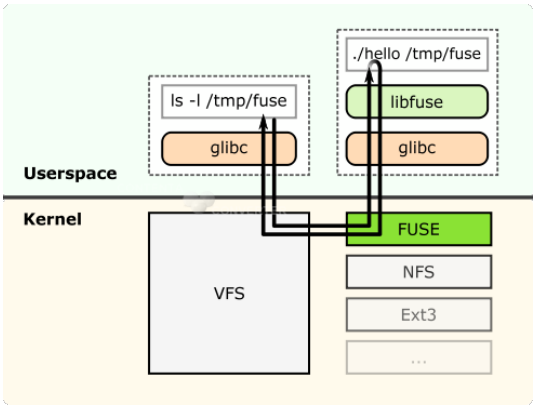


Fig. 1. A flow chart of ls command in FUSE framework

There are extra context switches involved in FUSE framework which result in some overheads compared to the native file system. In native UNIX file systems (e.g. Ext4) there are only two context switches between the caller in user space and the system call in kernel spaces. On the other hand, as shown in Figure 1 there are four context switches: two switches between the caller and VFS; and another two between libfuse and FUSE. Therefore, theoretically FUSE-based file systems have degraded performance when compared to native ones. A detailed comparison between FUSE-enabled and native file systems was reported in [30], which shows that a Java implementation of FUSE has about 60% overhead compared to the native file system.

The overhead can be potentially reduced to get a comparable performance of native file systems because FUSE provides a

few tuning parameters that can significantly adjust its performance. We will show that our implementation of FUSE on HDD has almost the same read and write bandwidth as Ext4. As reported in [30] it is even possible for FUSE-based file system to outperform the native file system because a finely tuned FUSE might have a better read-ahead mechanism than what the kernel uses. We treat Ext4 as the baseline because it is the most widely used Linux file system.

The FUSE framework consists of two major components to facilitate the development of file systems in user space: the FUSE kernel module and the *libfuse* library. Both components are described in Figure 1. The FUSE kernel module provides the low-level API that interacts with VFS interface closely. This is useful for file systems that are developed from scratch like ZFS-FUSE [31]. In contrast, the *libfuse* library can interpret the customized implementations of the FUSE standard interfaces and take over the file operations. In other words, *libfuse* offers a high level API to develop the file system in user space. Although it has less flexibility than the first option, in most cases it provides sufficient functionality that a file system needs. Like the vast majority of FUSE-based file systems, HyCache also adopts the second means i.e. implementing the high level APIs provided by FUSE.

III. Design

Figure 2 shows a bird's view of HyCache. At the highest level there are three logical components: request handler, file dispatcher and persistent storage. Request handler interacts with end users and passes the user requests to the file dispatcher. File dispatcher takes file requests from request handler and conducts the operations on the persistent storage. The persistent storage is just a mix of high- and slow-speed storage block devices, in our case the SSD and HDD drives.

A. Request Handler

The request handler is the first component of the whole system that interacts with end users. It allows end users to input POSIX file commands. The HyCache mount point can be any directory in a UNIX-like system as long as end users have sufficient permissions on that directory. This mount point is monitored by the FUSE kernel module, so any POSIX file operations on this mount point is passed to the FUSE kernel module. Then the FUSE kernel module will import the FUSE library and try to transfer the request to FUSE API in the file dispatcher.

The HyCache mount point itself is not only a single local directory but a virtual entry point of two mount points for SSD partition and HDD partition, respectively. Figure 3 shows how to mount HyCache in a UNIX-like system. Assuming HyCache would be mounted on a local directory called *hycache_mount*, and another local directory (e.g. *hycache_root*) has been created and has at least two subdirectories: the mount point of the SSD partition and the mount point of the HDD partition, users can simply execute *./hycache <root> <mount>* where *hycache* is the executable for HyCache, *root* is the physical directory and *mount* is the virtual directory.

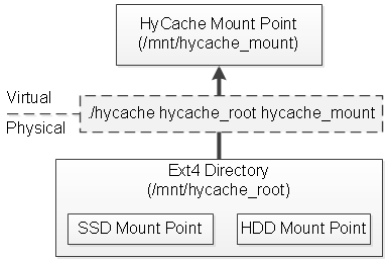


Fig. 3. HyCache Mount Point

B. File Dispatcher

File dispatcher is the core component of HyCache, as it redirects user-provided POSIX requests into customized handlers of file manipulations. FUSE only provides POSIX interfaces and file dispatcher is exactly the place where these interfaces are implemented, e.g. some of the most important file operations like *open()*, *read()* and *write()*, etc. File dispatcher manages the file locations and determines with which hard drive a particular file should be dealing. Some replacement policies, i.e. cache algorithms, need to be provided to guide the File Dispatcher.

Cache algorithms are optimizing instructions that a computer program can follow to manage a cache of information stored on the computer. When the cache is full, the algorithm must choose which items to discard to make room for the new ones. In case of HyCache, cache algorithm determines which file(s) in SSD are swapped to HDD when the SSD space is intensive. Different cache algorithms have been extensively studied in the past decades. There is no one single algorithm that suppresses others in all scenarios. We have implemented LRU (Least Recently Used) and LFU (Least Frequently Used) [36] in HyCache and the users are free to plug in their own algorithms for swapping files.

C. Persistent Storage

Persistent storage is like the traditional hard disk that is used to store files persistently. In HyCache it consists of at least two types of disks: the slower one (i.e. HDD) works like a permanent storage medium and the faster one (i.e. SSD) acts like a cache between the RAM and HDD. The SSD is not exactly the same as the conventional cache which is volatile: the data are only transient copies of the data in the persistent disk and will be lost when the system is restarted. Unlike traditional volatile cache where we have to write back the cache data to disks at some point, HyCache only needs to write data to SSD cache once and it becomes persistent (even after reboots). There is also nothing architecturally that prohibits us from leveraging more than two levels of storage with different characteristics.

D. Characteristics

This subsection describes some of the important characteristics of HyCache such as multithreading, naming and consistency.

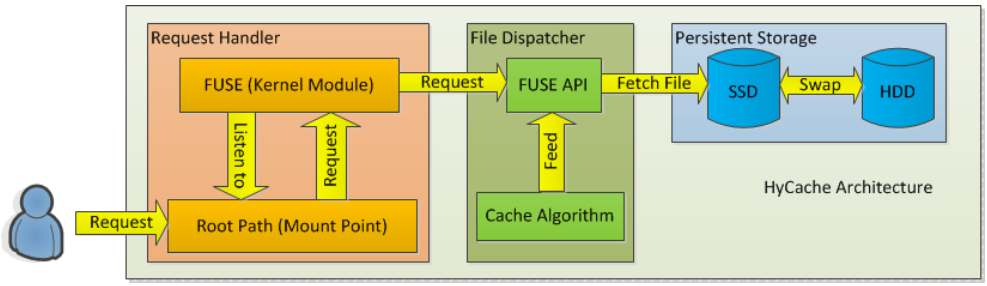


Fig. 2. HyCache Architecture

HyCache fully supports multithreading, although users have the option to disable it to run applications in the single-thread mode. Even though there are cases where multithreading does not help and only introducing overheads by switching contexts, by default multithreading is enabled in HyCache because in most cases this would improve the overall performance by keeping the CPU busy. We will see in the evaluation section how the aggregate throughput is significantly elevated with the help of concurrency.

Naming in HyCache follows the tree structure similarly to UNIX-like systems. In fact, one of the key features of HyCache is to provide an identical POSIX interface to end users. Therefore the name space is pretty much the same as a standard UNIX file system from the perspective of end users. However the infrastructure of HyCache has a more complicated mechanism because it needs to manage multiple name spaces and convert them to a single view to end users.

One common issue with cache is how to handle the consistency between the cached data and the persistent copy. HyCache does not have this problem because the SSD cache itself is persistent, and there is only one copy of the data at any moment. Certainly at some point the SSD cache data needs to swapped to HDD when the SSD usage is too high. In this case HyCache simply moves the cached file from SSD to HDD and creates a symbolic link in SSD to the physical file in HDD. Similarly, when removing a file which has been moved to HDD, HyCache deletes both the physical file in HDD and the symbolic link in SSD. In short, HyCache offers a strong consistency between the SSD cache and the HDD persistent storage.

IV. IMPLEMENTATION

HyCache has about 2,500 lines of C code, together with some scripts and configuration files. It was compiled with GCC version 4.6.3. and is published as an open source project [10]. The implementation language was chosen to be C due to several reasons. One major reason was that the FUSE framework is natively written in C, and we thought it would be most efficient and simple to keep the HyCache implementation in the same language. Second, it is hard to argue against C from a performance point of view, which was one of the primary goals of this work. And finally, we are planning to run the HyCache file system in many HPC systems, which many only support Fortran, C, and C++, making higher level languages such as Java, C#, and Python [6] not viable options.

A. Achieving POSIX through FUSE

FUSE provides 35 interfaces to fully comply with POSIX file operations and we implemented all of these 35 interfaces to ensure HyCache supports all POSIX operations. Some of these are called more frequently e.g. some essential file operations like *open()*, *read()*, *write()* and *unlink()* whereas others might be less popular or even remain optional in some Linux distributions like *getxattr()* and *setxattr()* which are to get and set extra file attributes, respectively.

Other than implementing FUSE interfaces some supplementary functionalities are needed to make HyCache a fully functional POSIX-like file system. For example a replacement policy should be available to swap stale SSD files to HDD when the SSD usage breaks some threshold. This actually indicates that another function is in need to continuously check the SSD usage as well as other system states. For the same reason mentioned before all these modules are also written in C. The implementations of cache algorithms are loosely coupled to the FUSE interfaces thus end users can easily plug in their own cache algorithms to satisfy their application-specific needs. This kind of flexibility is unheard of in commercial hybrid drives, such as those from Seagate [21] and OCZ [25].

There are two top level mount points in HyCache: *hycache_mount* and *hycache_root*. *hycache_mount* is the layer that connects the user and the underlying physical storages. From the user's perspective it provides end users the POSIX file operations and returns the results. From the perspective of *hycache_root*, it works like a soft link or mount point of the underlying physical files. *hycache_root* is the physical directory where HyCache manages files and is further divided into two subdirectories: a SSD partition and a HDD partition.

For manipulating files across multiple storage devices we use symbolic links to track file locations. Another possibility is to adopt hash tables. In this initial release we preferred symbolic links to hash tables for two reasons. First, symbolic link itself is persistent, which means that we do not need to worry about the cost of swapping data between memory and hard disk. It also offers a stronger consistency because there is only one copy of metadata at any given time, as opposed to multiple copies if hash tables were used. Second, symbolic

link is natively supported by UNIX-like systems and FUSE framework. Ideally, if we can make hash table persistent then it would be the best choice for managing metadata. We have been working on a persistent hash table project called NoVoHT [24] and will incorporate it in the next release.

Figure 4 shows a typical scenario of file mappings when the space of SSD cache is intensive so some file(s) needs to be swapped into the HDD. End users only see virtual files in HyCache mount point (i.e. *hycache_mount*) and every single file in the virtual directory is mapped to the underlying SSD physical directory. SSD only has a limited space so when the usage is beyond some threshold then HyCache will move some file(s) from SSD to HDD and only keep symbolic link(s) to the swapped files. The replacement policy, e.g. LRU or LFU, determines when and how to do the swapping.

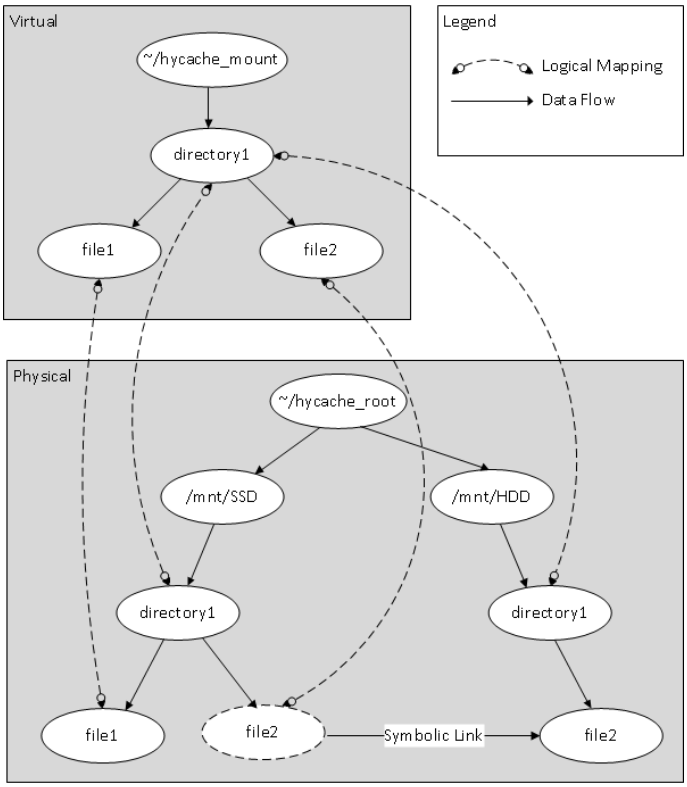


Fig. 4. File movement in HyCache. When free space of SSD cache is below some threshold, based on some caching algorithm file2 is evicted out of the SSD. The SSD cache still keeps a symbolic link of file2 which has been moved to the HDD drive.

Formally, Algorithm 1 describes how HyCache updates SSD cache when end users open files. The first thing is to check if the requested file is physically in HDD in Line 1. If so the system needs to reserve enough space in SSD for the requested file. This is done in a loop from Line 2 to Line 5 where the stale files are moved from SSD to HDD and the cache queue is updated. Then the symbolic link of the requested file is removed and the physical one is moved from HDD to SSD in Line 6 and Line 7. We also need to update the cache queue in in Line 8 and Line 10 for two scenarios, respectively. Finally the file is opened in Line 12.

Algorithm 1 Open a file in HyCache

Require: F is the file requested by the end user; Q is the cache queue used for the replacement policy; SSD is the mount point of SSD drive; HDD is the mount point of HDD drive

Ensure: F is appropriately opened

```
1:  if F is a symbolic link in SSD then
2:      while SSD space is intensive and Q is not empty do
3:          move some file(s) from SSD to HDD
4:          remove these files from the Q
5:      end while
6:      remove symbolic link of F in SSD
7:      move F from HDD to SSD
8:      insert F to Q
9:  else
10:     adjust the position of F in Q
11: end if
12: open F in SSD
```

Another important file operation in HyCache that is worth mentioning is file removal. We explain how HyCache removes a file in Algorithm 2. Line 4 and Line 5 are standard instructions used in file removal: update the cache queue and remove the file. Lines 1-3 check if the file to be removed is actually stored in HDD. If so, this regular file needs to be removed as well.

Algorithm 2 Remove a file in HyCache

Require: F is the file requested by the end user for removal; Q is the cache queue used for the replacement policy; SSD is the mount point of SSD drive; HDD is the mount point of HDD drive

Ensure: F is appropriately removed

```
1: if F is a symbolic link in SSD then
2:     remove F from HDD
3: end if
4: remove F from Q
5: remove F from SSD
```

We will not show the algorithm of each every POSIX file operation supported by HyCache. The idea is similar to Algorithm 1 and Algorithm 2: manipulate files in SSD and HDD back and forth to make users feel they are working on a single file system. We show one more example - *rename()*, which is to rename a file in Algorithm 3. If the file to be renamed is a symbolic in SSD, the corresponding file in HDD needs to be renamed as shown in Line 2. Then the symbolic link in SSD is outdated and needs to be updated in Lines 3-4. On the other hand if the file to be renamed is only stored in SSD then the renaming occurs only in SSD and the cache queue, as shown in Lines 6-7. In either case the position of the newly accessed file F' in the cache queue needs to be updated in Line 9.

Algorithm 3 Rename a file in HyCache

Require: F is the file requested by the end user to rename; F' is the new file name; Q is the queue used for the replacement policy; SSD is the mount point of SSD drive; HDD is the mount point of HDD drive

Ensure: F is renamed to F'

```
1: if F is a symbolic link in SSD then
2:     rename F to F' in HDD
3:     remove F in SSD
4:     create the symbolic link F' in SSD
5: else
6:     rename F to F' in SSD
7:     rename F to F' in Q
8: end if
9: update F' position in Q
```

B. Caching Algorithms

HyCache provides two built-in cache algorithms: LRU and LFU. End users are free to plug in other cache algorithms depending on their data patterns and/or application characteristics. As shown in Algorithms 1 - 3, all the implementations are independent of specific cache algorithms. LRU is one of the most widely used cache algorithms in computer systems. It is also the default cache algorithm used in HyCache. LFU is an alternative to facilitate the SSD cache if the access frequency is of more interests. In case all files are only accessed once (or for equal times), LFU is essentially the same as LRU, i.e. the file that is least recently used would be swapped to HDD if SSD space becomes intensive.

We implement LRU and LFU with the standard C library *<search.h>* instead of importing any third-party libraries for queue-handling utilities. This header supports doubly-linked list with only two operation: *insque()* for insertion and *remque()* for removal. We implement all other utilities from scratch e.g. check the queue length, search for a particular element in the queue, etc. Each element of LRU and LFU queues stores some metadata of a particular file like filename, access time, number of access (only useful for LFU though), etc.

We show how LRU is implemented in Figure 5. A new file is always created on SSD. This is possible because HyCache ensures the SSD partition has enough space for next file operation after current file operation. For example after editing a file, the system checks if the usage of SSD has hit the threshold of being considered as "SSD space is intensive". Users can define this value by their own, for example 90% of the entire SSD. When the new file has been created on SSD it is also inserted in to the tail of LRU queue. On the other hand, if the SSD space is intensive we need to keep swapping the heads of LRU queue into HDD until the SSD usage is below the threshold. Both cases are pretty standard queue operations as shown in the top part of Figure 5. If a file already in the LRU queue gets accessed then we need to update its position in the LRU queue to reflect the new time stamp of this file. In particular, as shown in the bottom part of Figure 5, the newly accessed file needs to be removed from the queue and re-inserted into the tail.

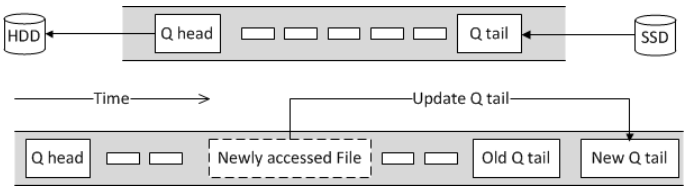


Fig. 5. LRU in HyCache

LFU is implemented in a similar way as LRU with a little more work. In LFU, the position of a file in the queue is determined by two criteria: frequency and timestamp. LFU first checks the access frequency of the file. The more frequently this file has been touched, the closer it will be positioned to the queue tail. If there are multiple files with the same frequency, for this particular set of files LRU will be applied, i.e. based on timestamp.

V. Evaluation

The experiments are carried out on a system comprised of an AMD Phenom II X6 1100T Processor (6 cores at 3.3 GHz) and 16 GB RAM. The spinning disk is Seagate Barracuda 1 TB. The SSD is OCZ RevoDrive×2 100 GB. The HHD is Seagate Momentus XT 500 GB (with 4 GB built-in SSD cache). The operating system is 64-bit Fedora 16 with Linux kernel version 3.3.1. The native file system is Ext4 with default configurations (i.e. *mkfs.ext4 /dev/device*).

We have tested the functionality and performance of HyCache in three experiments. The first two are benchmarks with synthetic data to test the raw bandwidth of HyCache. In particular, these benchmarks can be further categorized into micro-benchmarks and macro-benchmarks. Micro-benchmarks are used to measure the performance of some particular file operations and their raw bandwidths. Macro-benchmarks, on the other hand, are focused on application-level performance of a set of mixed operations simulated on a production server. For both types of benchmarks we pick two of most popular ones to demonstrate HyCache performance: IOzone [4] and PostMark [14]. The third experiment is to test the functionality of HyCache with a real application. We achieve this by successfully installing MySQL on HyCache and deployed a couple of TPC-H databases [37] with different scales and time the execution for loading tables and making queries.

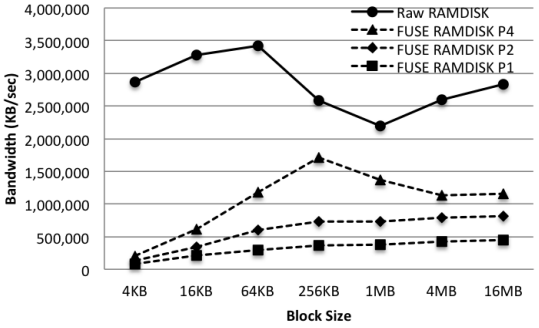
In the remaining of this paper we will use terms throughput and bandwidth interchangeably, which basically means the rate of data transferring. Unless otherwise specified all bandwidths are with respect to sequential read and write operations.

A. FUSE overhead

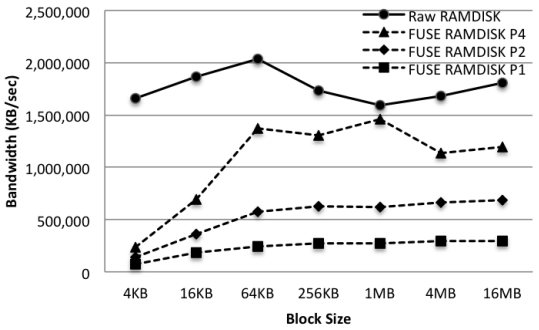
To understand the overhead introduced by FUSE, we compare the I/O performance between raw RAMDISK (i.e. tmpfs [34]) and a simple FUSE file system mounted on RAMDISK. By experimenting on RAMDISK we completely eliminate all

factors affecting performance particularly from the spinning disk, disk controllers, etc. Since all the I/O tests are essentially done on the memory, any noticeable performance differences between the two setups are solely from FUSE itself.

We mount FUSE on */dev/shm*, which is a built-in RAMDISK in UNIX-like systems. The read and write bandwidth on both raw RAMDISK and FUSE-based virtual file system are reported in Figure 6. Moreover, the performance of concurrent FUSE processes are also plotted which shows that FUSE has a good performance scalability with respect to the number of concurrent jobs. In the case of single-process I/O, there is a significant performance gap between Ext4 and FUSE on RAMDISK. The read and write bandwidths of Ext4 on RAMDISK are in the order of gigabytes, whereas when mounting FUSE we could only get a bandwidth in the range of 500 MB/s. These results suggest that FUSE could not compete with the kernel-level file systems in raw bandwidth, primarily due to the overheads incurred by having the file system in user-space, the extra memory copies, and the additional context switching. However, we will see in the following subsections that even with FUSE overhead on SSD, HyCache still outperforms traditional spinning disks significantly, and that concurrency can be used to scale up FUSE performance close to the theoretical hardware performance (see Figure 9 and Figure 10).



(a) Read Bandwidth



(b) Write Bandwidth

Fig. 6. Bandwidth of raw RAMDISK and a FUSE file system mounted on RAMDISK. Px means x number of concurrent processes, e.g. FUSE RAMDISK P2 stands for 2 concurrent FUSE processes on RAMDISK.

B. Micro-benchmark

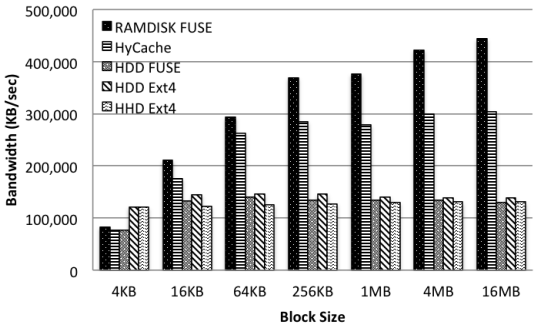
IOzone is a general file system benchmark utility. It creates a temporary file with arbitrary size provided by the end user and then conducts a bunch of file operations like re-write, read, re-read, etc. In this paper we use IOzone to test the read and write bandwidths as well as IOPS (input/output per second) on the different file systems.

We show the throughput with a variety of block sizes ranging from 4 KB to 16 MB. For each block size we show five bandwidths from the left to the right: 1) the theoretical bandwidth upper bound (obtained from RAMDISK), 2) HyCache, 3) a simple FUSE file system accessing a HDD, 4) HDD Ext4 and 5) HHD Ext4.

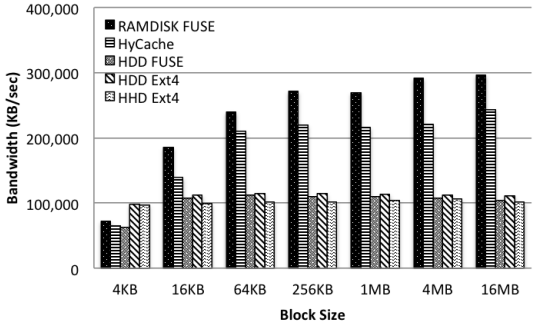
Figure 7(a) shows HyCache read speed is about doubled compared to the native Ext4 file system for most block sizes. In particular when block size is 16 MB the peak read speed for HyCache is over 300 MB/s. It is 2.2X speedup with respect to the underlying Ext4 for HDD as shown in Figure 8(a). As for the overhead of FUSE framework compared to the native Ext4 file system on spinning disks we see FUSE only adds little overhead to read files at all block sizes as shown in Figure 8(a): for most block sizes FUSE achieves nearly 100% performance of the native Ext4. Similar results are also reported in a review of FUSE performance in [30]. This fact indicates that even when the SSD cache is intensive and some files need to be swapped between SSD and HDD, HyCache can still outperform Ext4 since the slower media of HyCache (HDD FUSE in Figure 7), are comparable to Ext4. We will present the application-level experimental results in the macro-benchmark subsection where we discuss the performance when files are frequently swapped between SSD and HDD. We can also see that the commercial HHD product performs at about the same level of the HDD, likely primarily due to a small and inexpensive SSD.

We see a similar result of file writes in Figure 7(b) as file reads. Again HyCache is about twice as fast when compared to Ext4 on spinning disks for most block sizes. The peak write bandwidth which is almost 250 MB/s is also obtained when block size is 16 MB, and it achieves 2.18x speedup for this block size compared to Ext4 as shown in Figure 8(b). Also in this figure, just like the case of file reads we see little overhead of FUSE framework for the write operation on HDD except for 4KB block.

Figure 8 shows that for small block size (i.e. 4 KB) HyCache only achieves about 50% throughput of the native file system. This is due to the extra context switches of FUSE between user level and kernel level, in which the context switches of FUSE dominate the performance. Fortunately in most cases this small block size (i.e. 4 KB) is more generally used for randomly read/write of small pieces of data (i.e. IOPS) rather than high-throughput applications. Table II shows HyCache has a far higher IOPS than other Ext4. In particular, HyCache has about 76X IOPS as traditional HDD. The SSD portion of the HHD device (i.e. Seagate Momentus XT) is a read-only cache, which means the SSD cache does not take

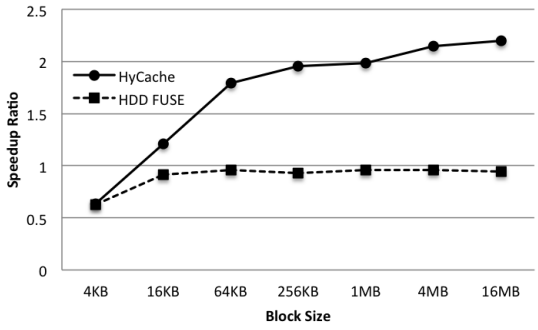


(a) Read Bandwidth

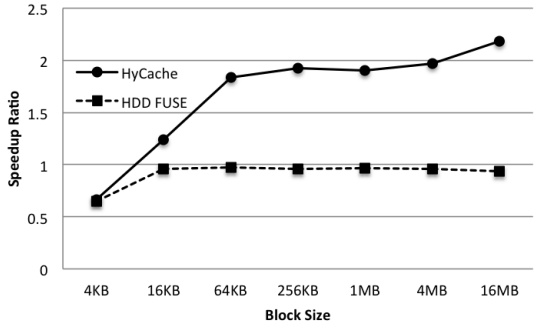


(b) Write Bandwidth

Fig. 7. IOzone bandwidth of 5 file systems.



(a) Read Speedup



(b) Write Speedup

Fig. 8. HyCache and FUSE speedup over HDD Ext4.

effect in this experiment because IOPS only involves random writes. This is why the IOPS of the HHD lands in the same level of HDD rather than SSD.

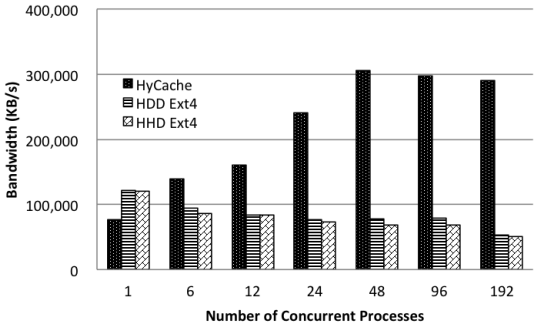
TABLE II
IOPS OF DIFFERENT FILE SYSTEMS

HyCache	HDD Ext4	HHD Ext4
14,878	195	61

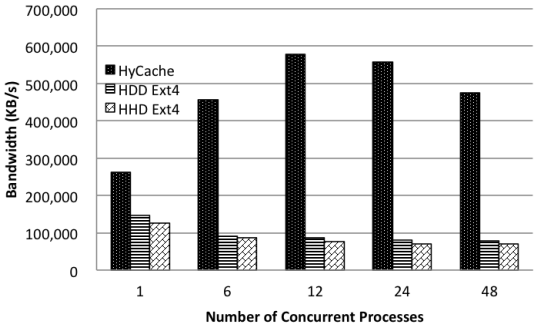
HyCache also takes advantages of the multicore's concurrent tasking which results in a much higher aggregate throughput. The point is that HyCache avoids reading/writing directly on the HDD so it handles multiple I/O requests concurrently. In contrast, traditional HDD only has a limited number of heads for read and write operations. Figure 9 shows that HyCache has almost linear scalability with the number of processes before hitting the physical limit (i.e. 306 MB/s for 4 KB block and 578 MB/s for 64 KB block) whereas the traditional Ext4 has degraded performance when handling concurrent I/O requests. The largest gap is when there are 12 concurrent processes for 64KB block (578 MB/s for HyCache and 86 MB/s for HDD): HyCache has 7X higher throughput than Ext4 on HDD.

The upper bound of aggregate throughput is limited by the SSD device rather than HyCache. This can be demonstrated in Figure 10 which shows how HyCache performs in RAMDISK. The performance of raw RAMDISK were also plotted. We can see that the bandwidth of 64KB block can be achieved at about 4 GB/s by concurrent processes. This indicates that FUSE itself is not a bottle neck in the last experiment: it will not limit the I/O speed unless the device is slow. This implies that HyCache can be applied to any faster storage devices in future as long as the workloads have enough concurrency to allow FUSE to harness multiple computing cores. Another observation is that HyCache can consume as much as 35% of raw memory bandwidth as shown in Figure 10 for 64KB block and 24 processes: 3.78 GB/s for HyCache and 10.80 GB/s for RAMDISK.

What FUSE overhead hurts HyCache most severely is the metadata management. Because of the aforementioned extra context switches between kernel and user spaces, the metadata operations in HyCache are slower than Ext4, as shown in Table III. The workaround of this is to plug-in an in-memory metadata management system for acceleration. We implemented an emulation feature in HyCache that allows us to study the effects of different metadata management techniques. When we tried incorporating NoVoHT (Non-Volatile Hash Table), which is essentially an in-memory hash table and uses a log-based persistence mechanism with periodic checkpointing, we observed a comparable performance of metadata operations to those of native kernel-level file systems such as Ext4. We will integrate NoVoHT into HyCache in the next release, together with some other optional choices.



(a) 4KB Block



(b) 64KB Block

Fig. 9. Aggregate bandwidth of concurrent processes.

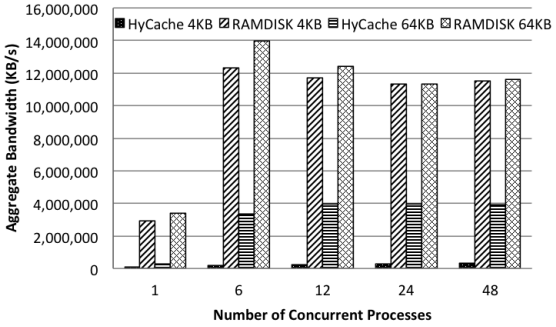


Fig. 10. Aggregate bandwidth of the FUSE implementation on RAMDISK.

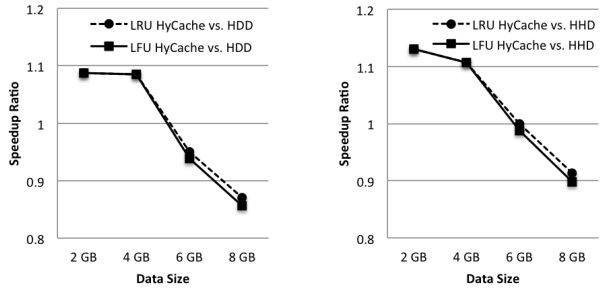
TABLE III
NUMBER OF METADATA OPERATIONS PER SECOND

HyCache	HyCache w/ NoVoHT	HDD	HHD
68	322	362	367

C. Macro-benchmark

PostMark is a one of the most popular benchmarks to simulate different workloads in file systems. It was originally developed to measure the performance of ephemeral small-file regime used by Internet software like Emails, netnews and web-based commerce, etc. A single PostMark instance carries out a number of file operations like read, write, append and delete, etc. In this paper we use PostMark to simulate a synthetic application that performs different number of file I/Os on HyCache with two cache algorithms LRU and LFU, and compare their performances to Ext4.

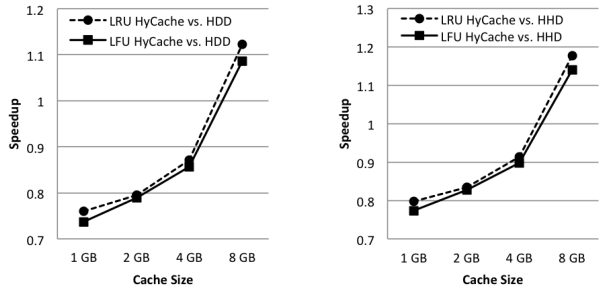
We show PostMark results of four file systems: HyCache with LRU, HyCache with LFU, Ext4 on HDD and Ext4 on HHD. And for each of them we carried out four different workloads: 2 GB, 4 GB, 6 GB and 8 GB. To make a fair comparison between HyCache and the HHD device (i.e. Momentous XT: 4 GB SSD and 500 GB HDD), we set the SSD cache of HyCache to 4 GB. Figure 11 shows the speedup of HyCache with LRU and LFU compared to Ext4 on HDD and HHD. The difference between LRU and LFU is almost negligible. The ratio starts to go down at 6 GB because HyCache only has 4 GB allocated SSD. Another reason is that PostMark only creates temporary files randomly without any repeated pattern. In other words it is a data stream making the SSD cache thrashes (this could be considered to be the worst case scenario).



(a) HyCache vs. HDD (b) HyCache vs. HHD

Fig. 11. PostMark: speedup of HyCache over Ext4 with 4 GB SSD cache.

A big advantage of HyCache is that users can freely allocate the size of the SSD cache. In the last experiment HyCache did not work well as HHD mainly because the data is too large to fit in the 4 GB cache. Here we show how increasing the cache size impacts the performance. Figure 12 shows that if a larger SSD cache (i.e. 1GB - 8GB) is offered then the performance is indeed better than others with as much as a 18% performance improvement: LRU HyCache with 8GB SSD cache vs. HHD.



(a) HyCache vs. HDD (b) HyCache vs. HHD

Fig. 12. PostMark: speedup of HyCache with varying sizes of cache.

D. Application

To further test HyCache with real applications, on HyCache we install MySQL 5.5.21 with database engine MySIAM, and deploy TPC-H 2.14.3 databases.

MySQL is open source and one of the most popular databases nowadays. Unlike other commercial databases like Oracle or Microsoft SQL Server which has a proprietary architecture, MySQL has a simple yet sufficient design: database is just a directory and tables within a database are the files under that directory. This feature makes MySQL a good fit for HyCache, as all we need is to mount the database as a directory in HyCache and manipulate files (i.e. tables, from the perspective of MySQL) within that directory. In essence, when loading data to the database we are performing file writes whereas executing a query is actually reading the files.

TPC-H is an industry standard benchmark for databases. By default it provides a variety size of databases (e.g. scale 1 for 1 GB, scale 10 for 10 GB, scale 100 for 100GB) each of which has eight tables. Further, TPC-H provides 22 complicated queries (i.e. Query #1 to Query #22) that are comparable to business applications in the real world. Figure 13 shows Query #1 which will be used in our experiments.

```
select
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
from
    lineitem
where
    l_shipdate <= date '1998-12-01' - interval '72' day
group by
    l_returnflag,
    l_linestatus
order by
    l_returnflag,
    l_linestatus;
```

Fig. 13. TPC-H: Query #1.

To test file writes in HyCache, we loaded table *lineitem* at scale 1 (which is about 600 MB) and scale 100 (which is about 6 GB) in these three file systems: LRU HyCache, HDD Ext4 and HHD Ext4. As for file reads we ran Query #1 at scale 1 and scale 100. HyCache has an overall of 9% and 4% improvement over Ext4 on HDD and HHD, respectively. The result details of these experiments are reported in Figure 14.

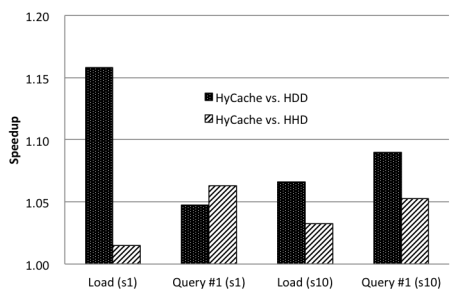


Fig. 14. TPC-H: speedup of HyCache over Ext4 on MySQL.

VI. Related Work

To the best of our knowledge HyCache is the first user-level POSIX-compliant file system for coordinating SSDs and HDDs. Some existing work requires modifying OS kernel, or lacks of a systematic caching mechanism for manipulating files across multiple storage devices, or does not support the POSIX interface. Any of the these concerns would limit the system's applicability to end users. We will give a brief review of previous studies on hybrid storage systems.

Some recent work reported the performance comparison between SSD and HDD in more perspectives ([33], [35]). SSD was also proposed to be integrated to the RAM level which makes SSD as the primary holder of virtual memory [2]. Hystor [5] aims to optimize of the hybrid storage of SSDs and HDDs. However it requires to modify the kernel which might cause some issues. HPDA [19] offers a mechanism to plug SSDs into RAID in order to improve the reliability of the disk array. ComboDrive [29] requires extra hardware support to statically allocate files between SSD and HDD based on file types. Jo et al [12] proposed another SSD+HDD architecture which supports a read-only image on SSD and isolates write operations only on HDD. A similar architecture was proposed in [40] which only considers SSD as a read-only buffer and migrate those random-writes to HDD. In database community, Khessib [15] proposed a similar architecture which leverages SSD as an intermediate persistent cache that sits between conventional HDD and memory. HeteroDrive [16] translates random writes via SSDs to sequential writes on HDD. Another hybrid architecture was proposed in [17] by combining two kinds of NAND flash - MLC flash and SLC flash. CacheFS [3] is an ongoing project from the open source community and still in an initial stage, e.g. statically allocating files based on file sizes.

VII. Conclusion and Future Work

In this paper we address one of the major challenges in computing systems and propose a cost-effective solution to alleviate the storage bottleneck. We believe that this work can be a solid building block for future distributed storage systems, aimed at delivering comparable performance of an all SSD solution at a fraction of the cost. We designed and implemented a user-level POSIX-compliant file system with high throughput, low latency, single name space, and strong consistency. Our extensive performance evaluation showed that user-level file systems can be competitive with kernel-level file systems as well as embedded hybrid hard drive technologies.

HyCache will adopt NoVoHT to improve the performance of metadata operations as the default metadata management. HyCache will be eventually integrated into FusionFS [7] which is a high-performance distributed file system aimed at exascale computing, currently being developed by the authors of this work.

Acknowledgement

This work was supported by the National Science Foundation (NSF) under grant OCI-1054974.

REFERENCES

[1] Alexander Galanin, Fuse-zip project, *http://code.google.com/p/fuse-zip/*.

[2] Anirudh Badam and Vivek S. Pai, *SSDAlloc: hybrid SSD/RAM memory management made easy*, Proceedings of the 8th USENIX conference on Networked systems design and implementation, 2011.

[3] CacheFS, CacheFS project, *http://code.google.com/p/cachefs/*.

[4] D. Capps, *IOzone Filesystem Benchmark*, www.iozone.org (2008).

[5] Feng Chen, David Koufaty, and Xiaodong Zhang, *Hystor: Making the best use of solid state drives in high performance storage systems*, Proceedings of the international conference on Supercomputing (ICS), 2011.

[6] FUSE Project, *http://fuse.sourceforge.net*.

[7] FusionFS: Fusion distributed File System, *http://datasys.cs.iit.edu/projects/FusionFS/*.

[8] Gordon Moore, *Moore's Law*, Intel Cooperation (1965).

[9] Hitachi Deskstar 7K4000, *http://www.hitachigst.com/deskstar-7k4000*.

[10] HyCache: a hybrid user-level file system with SSD caching, *http://datasys.cs.iit.edu/projects/HyCache/index.html*.

[11] Jan Kratochvil, Captive project, *http://www.jankratochvil.net/project/captive/*.

[12] Heeseung Jo, Youngjin Kwon, Hwanju Kim, Euiseong Seo, Joonwon Lee, and Seungryoul Maeng, *SSD-HDD-Hybrid Virtual Disk in Consolidated Environments*, Euro-Par 2009 – Parallel Processing Workshops, 2009.

[13] John Madden, Svnfs project, *http://www.jmadden.eu/index.php/svnfs/*.

[14] Jeffrey Katcher, *Postmark: A new file system benchmark*, Network Appliance, Inc., vol. 3022, 1997.

[15] Badriddine M. Khessib, Kushagra Vaid, Sriram Sankar, and Chengliang Zhang, *Using solid state drives as a mid-tier cache in enterprise database OLTP applications*, Proceedings of the second TPC technology conference on performance evaluation, measurement and characterization of complex systems, 2010.

[16] Sang-Hoon Kim, Dawoon Jung, Jin-Soo Kim, and Seungryoul Maeng, *HPDA: A hybrid parity-based disk array for enhanced performance and reliability*, International Workshop on Software Support for Portable Storage, 2009.

[17] Li-Pin Chang, *Hybrid solid-state disks: combining heterogeneous NAND flash in large SSDs*, Proceedings of the 2008 Asia and South Pacific Design Automation Conference, 2008.

[18] Linux.com, *http://archive09.linux.com/feature/49757*.

[19] Bo Mao, Hong Jiang, Dan Feng, Suzhen Wu, Jianxi Chen, Lingfang Zeng, and Lei Tian, *HeteroDrive: Reshaping the Storage Access Pattern of OLTP Workload Using SSD*, IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010.

[20] Mathieu Blondel, Wikipediafs project, *http://wikipediafs.sourceforge.net*.

[21] Mementus XT, *http://www.seagate.com/www/en-us/products/internal-storage/momentus-xt-kit*.

[22] Newegg, *http://www.newegg.com*.

[23] Nikolaus Rath, S3ql project, *http://code.google.com/p/s3ql/*.

[24] NoVoHT: Non-Volatile Hash Table, *http://datasys.cs.iit.edu/projects/index.html*.

[25] OCZ RevoDrive 1TB Hybrid Solid State Drive, *http://www.ocztechnology.com/ocz-revodrive-hybrid-pci-express-solid-state-drive.html*.

[26] OCZ RevoDrive 3 X2 PCI-Express SSD, *http://www.ocztechnology.com/ocz-revodrive-3-x2-pci-express-ssd.html*.

[27] OCZ Synapse, *http://www.ocztechnology.com/ocz-synapse-cache-sata-iii-2-5-ssd.html*.

[28] Patrick Frank, Cvsfs project, *http://sourceforge.net/projects/cvsfs*.

[29] Hannes Payer, Marco A.A. Sanvido, Zvonimir Z. Bandic, and Christoph M. Kirsch, *Combo Drive: Optimizing cost and performance in a heterogeneous storage device*, Proceedings of the 1st Workshop on integrating solid-state memory into the storage hierarchy (WISH'09), 2009.

[30] Aditya Rajgarhia and Ashish Gehani, *Performance and extension of user space file systems*, Proceedings of the 2010 ACM Symposium on Applied Computing (Sierre, Switzerland), March 2010.

[31] Ricardo Correia, Zfs-fuse project, *http://zfs-fuse.net/*.

[32] Steven W. Schlosser, John Linwood Griffin, , David F. Nagle, and Gregory R. Ganger, *Filling the Memory Access Gap: A Case for On-Chip Magnetic Storage*, Tech. report, School of Computer Science, Carnegie Mellon University, 1999.

[33] Shan Li and H.H. Huang, *Black-Box Performance Modeling for Solid-State Drive*, IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010.

[34] Peter Snyder, *tmpfs: A virtual memory file system*, Proceedings of the Autumn 1990 European UNIX Users' Group Conference, 1990.

[35] S.S. Rizvi and Tae-Sun Chung, *Flash SSD vs HDD: High performance oriented modern embedded and multimedia storage systems*, 2nd International Conference on Computer Engineering and Technology (ICCET), 2010.

[36] Stefan Podlipnig and Laszlo Boszormenyi, *A survey of Web cache replacement strategies*, ACM Computing Surveys (CSUR), Volume 35 Issue 4 (2003).

[37] Transaction Processing Performance Council, *TPC Benchmark H*, http://www.tpc.org/tpch, 2008.

[38] Tsukasa Hamano, Mysqlfs project, *http://sourceforge.net/projects/mysqlfs/*.

[39] Wikipedia, Filesystem in Userspace, *http://en.wikipedia.org/wiki/Filesystem_in_Userspace*.

[40] Qing Yang and Jin Ren, *I-CASH: Intelligently Coupled Array of SSD and HDD*, IEEE 17th International Symposium on High Performance Computer Architecture (HPCA), 2011.