HyCache: a User-Level Caching Middleware for Distributed File Systems

Dongfang Zhao*, Ioan Raicu*†
*Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616
†Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439
dzhao8@hawk.iit.edu, iraicu@cs.iit.edu

***Abstract*—One of the bottlenecks of distributed file systems deals with mechanical hard drives (HDD). Although solid-state drives (SSD) have been around since the 1990's, HDDs are still dominant due to large capacity and relatively low cost. Hybrid hard drives with a small built-in SSD cache does not meet the need of a large variety of workloads. This paper proposes a middleware that manages the underlying heterogeneous storage devices in order to allow distributed file systems to leverage the SSD performance while leveraging the capacity of HDD. We design and implement a user-level filesystem, HyCache, that can offer SSD-like performance at a cost similar to a HDD. We show how HyCache can be used to improve performance in distributed file systems, such as the Hadoop HDFS. Experiments show that HyCache achieves up to 7X higher throughput and 76X higher IOPS than Linux Ext4 file system, and can accelerate HDFS by 28% at 32-node scales (compared to vanilla HDFS).**

***Index Terms*—distributed file systems, user level file systems, hybrid file systems, heterogeneous storage, SSD**

I. Introduction

One of the bottlenecks of distributed file systems (DFS), e.g. Google File System [1] and Hadoop Distributed File System [2], is mechanical hard disk drives (HDD): their slow increase in bandwidth, slow decrease in latency, and exponential increase in capacity, have made modern storage devices quite unbalanced. Making things worse, the low bandwidth and high latency of HDD hinders the exploration of data locality, which is critical to distributed computing applications [3]. Even though non-volatile memory e.g. 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 [4] e.g. CPU still follows Moore's Law [5].

SSD could bridge the gap nicely between RAM and HDD for a distributed file system, except that replacing all HDDs with SSDs of the same capacity would be prohibitively expensive. In Table I we show the per-GB cost of the high-end SSD OCZ RevoDrive is 41X higher than HDD (Hitachi Deskstar). The industry also introduced hybrid hard drives (HHD) where an embedded SSD accelerates the mechanical hard drive. For example, Seagate has just released a product Momentus XT [6] which encapsulates both a 4GB SSD and a 500GB HDD into a single physical device. The advantage for such a HHD is that it is a drop-in replacement to HDD, however its small fixed SSD cache ($< 1\%$ capacity) limits its ability to accelerate large numbers of workloads. Furthermore, the small SSD cache typically has inexpensive and relatively slow controllers in order to keep the costs low. Compounding the limitations, often time these HHD only use the SSD cache to accelerate read operations, missing a significant opportunity to accelerate write operations as well. These drawbacks limit the applicability to fully leverage low cost HHD architectures for their potential higher performance.

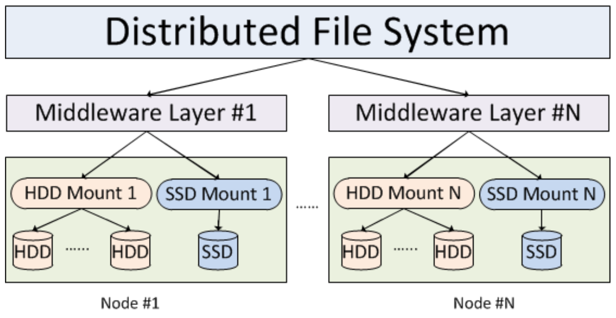


Fig. 1. The storage hierarchy with a middleware between distributed file systems and local file systems.

We propose a middleware called HyCache to manage heterogeneous storage devices for distributed file systems. HyCache provides standard POSIX interfaces through FUSE [7] and works completely in the user space. We show that in the context of file systems, the overhead of user-level APIs (i.e. *libfuse*) is negligible with multithread support on SSD, and with appropriate tuning can even outperform the kernel-level implementation (e.g. Seagate Momentus XT). The user-space feature of HyCache allows non-privileged users to specify the SSD cache size, an invaluable feature in making HyCache more versatile and flexible to support a much wider array of workloads. Furthermore, distributed or parallel file systems can leverage HyCache without any modifications through its POSIX interface. This is critical in many cases where the end users are not allowed to modify the kernel of HPC systems. Figure 1 shows the conceptual view of the storage hierarchy with HyCache. Instead of being mounted directly on the native file systems (e.g. Linux Ext4), distributed file systems are

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 X2	$2.81	960	1,500	1,300	230,000
OCZ Octane	$1.76	512	480	330	26,000
Seagate Momentus XT	$0.16	504	131	101	238
Hitachi Deskstar	$0.068	4,096	144	142	360

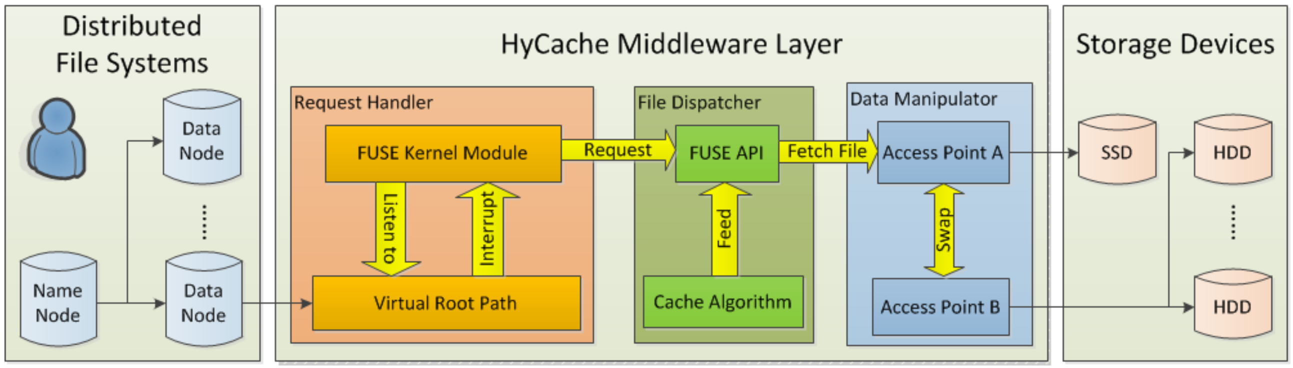


Fig. 2. Three major components in HyCache architecture: Request Handler, File Dispatcher and Data Manipulator.

deployed on top of HyCache on all data nodes.

The contribution of this work is threefold as follows:

1) *Designed and implemented HyCache, a versatile user-level POSIX-compliant file system with configurable caches that delivers high throughput, low latency, strong consistency, single namespace, and multithreaded support.*
2) *Developed a middleware layer between distributed file systems and large-capacity HDD with HyCache - delivered 28% improvement in HDFS performance.*
3) *Extensive performance evaluation showing that user-level file systems can be competitive with kernel-level file systems.*

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

II. DESIGN

In this section we will present HyCache architecture and discuss some pros and cons in our design. Figure 2 shows a bird's view of HyCache as a middleware between distributed file systems and local storages. At the highest level there are three logical components: request handler, file dispatcher and data manipulator. Request handler interacts with distributed file systems and passes the requests to the file dispatcher. File dispatcher takes file requests from request handler and decides where and how to fetch the data based on some replacement algorithm. Data manipulator manipulates data between two access points of fast- and regular-speed devices, respectively.

A. *Request Handler*

The request handler is the first component of the whole system that interacts with distributed file systems. HyCache virtual 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.

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)

[8] in HyCache and the users are free to plug in their own algorithms for swapping files.

C. Data Manipulator

Data manipulator manipulates data between two logical access points: one for fast speed access, i.e. SSDs and the other is for regular access e.g. HDDs. An access point is not necessarily a mount point of a device in the local operating system, but a logical view of any combination of these mount points. In the simplest case, Access point A could be the SSD mount point whereas access Point B is set to the HDD mount point. Access point A is always the preferred point for any data request as long as it has enough space based on some user defined criteria. Data need to be swapped back and forth between A and B once the space usage in A exceeds some threshold. For simplicity we only show Access point A and B in the figure, however there is nothing architecturally that prohibits us from leveraging more than two levels of access points.

III. Implementation

A. User Interface

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.

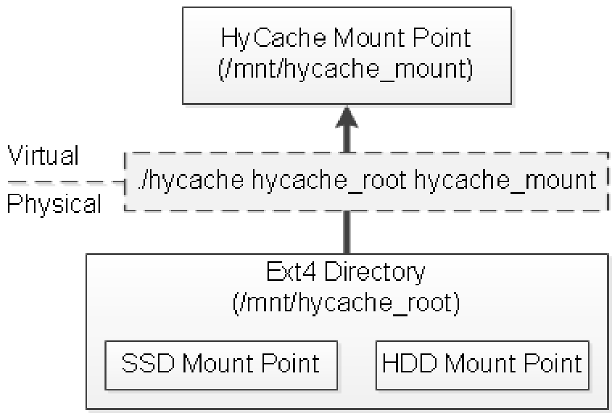


Fig. 3. How to mount HyCache in a UNIX-like machine.

B. Strong Consistency

We keep only one single copy of any file at any time to achieve strong consistency. 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. Second, symbolic link is natively supported by UNIX-like systems and FUSE framework.

HyCache is implemented for manipulating data at the file level rather than the block level because it is the job of the upper-level distributed file system to chop the big files (e.g. > 1TB). For example in Hadoop Distributed File System, an arbitrarily large file will typically be chunked up in 64MB chunks on each data node. Thus HyCache only needs to deal with these relatively small data blocks of 64MB that can be perfectly fit in a mainstream SSD device.

C. Single Namespace

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.

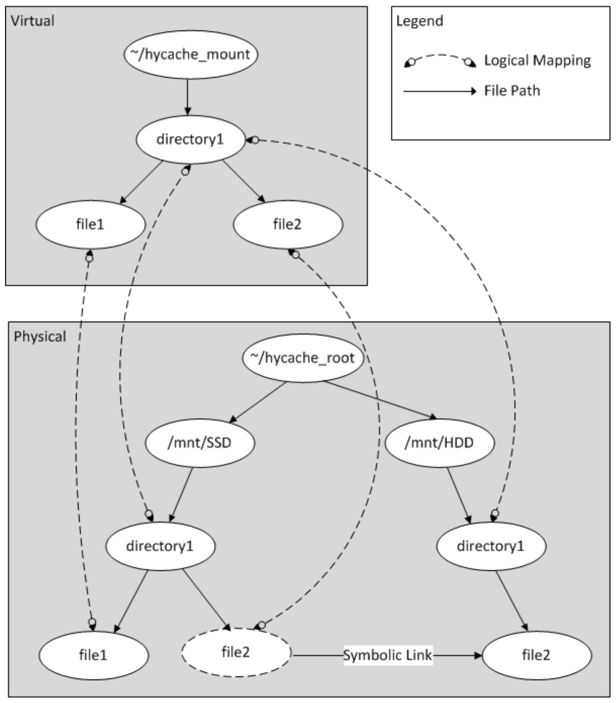


Fig. 4. File movement in HyCache. When free space of SSD cache is below some threshold, based on some caching algorithm file2 is evicted out of the SSD. The SSD cache still keeps a symbolic link of file2 which has been moved to the HDD drive.

We illustrate how a file is opened as an example. 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 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
```

Other POSIX implementations share the similar idea 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, e.g. *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
```

D. 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, 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.

Figure 5 illustrates how LRU is implemented for HyCache. 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.

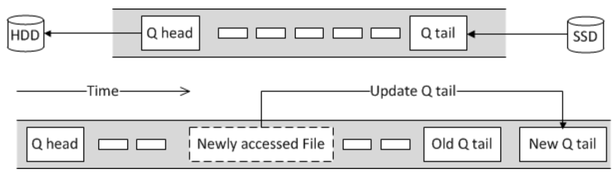


Fig. 5. LRU queue in HyCache.

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.

E. Multithread Support

HyCache fully supports multithreading to leverage the many-core architecture in most high performance computers. Users have the option to disable this feature to run applications in the single-thread mode. Even though there are cases where multithreading does not help and only introduces 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.

IV. Evaluation

Single-node 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 RevoDrive2 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*). For the experiments on Hadoop the testbed is a 32-node cluster, each of which has two Quad-Core AMD Opteron 2.3GHz processors with 8GB memory. The SSD and HDD are the same as in the single node workstation.

We have tested the functionality and performance of HyCache in four 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 [9] and PostMark [10]. The third and fourth experiments are to test the functionality of HyCache with a real application. We achieve this by deploying MySQL and HDFS on top of HyCache, and execute TPC-H queries [11] on MySQL and the built-in 'sort' application of Hadoop, respectively.

In the remainder 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. All the results are averages of at least 3 stable (i.e. within 5% difference) numbers.

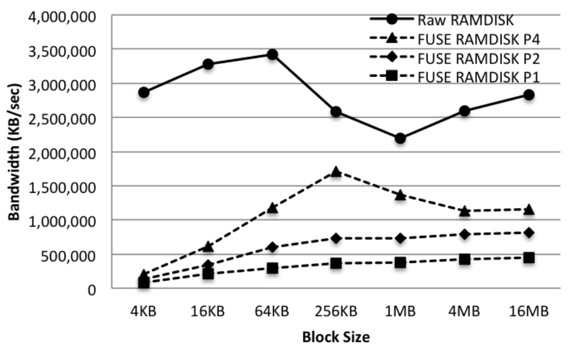
A. FUSE overhead

To understand the overhead introduced by FUSE in HyCache, we compare the I/O performance between raw RAMDISK (i.e. tmpfs [12]) 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.

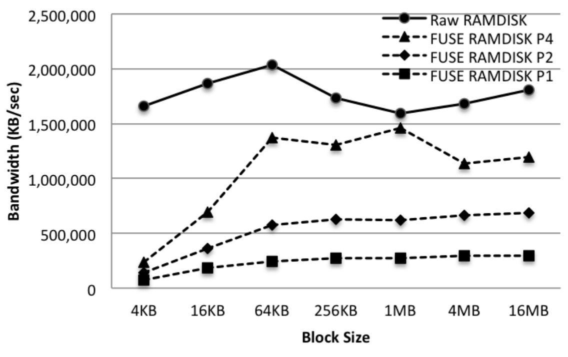
We mount FUSE on */dev/shm*, which is a built-in RAMDISK in UNIX-like systems. The read and write bandwidth on both raw RAMDISK and FUSE-based virtual file system are reported in Figure 6. Moreover, the performance of concurrent FUSE processes are also plotted which shows that FUSE has a good performance 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 bandwidth 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).

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.



(a) Read Bandwidth

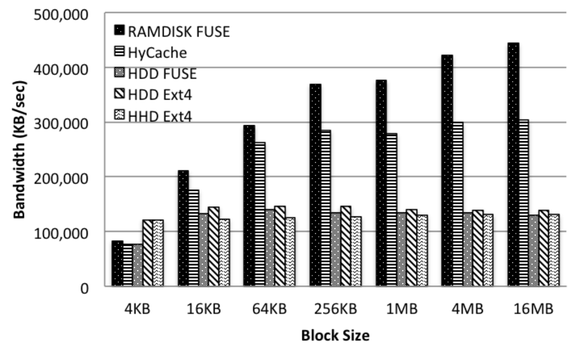


(b) Write Bandwidth

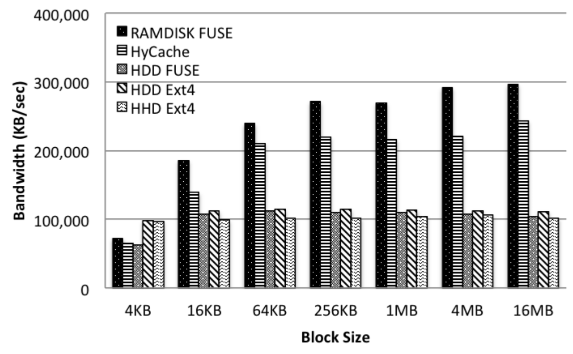
Fig. 6. Bandwidth of raw RAMDISK and a FUSE file system mounted on RAMDISK. Px means x number of concurrent processes, e.g. FUSE RAMDISK P2 stands for 2 concurrent FUSE processes on RAMDISK.

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 comparing 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 [13]. 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.



(a) Read Bandwidth

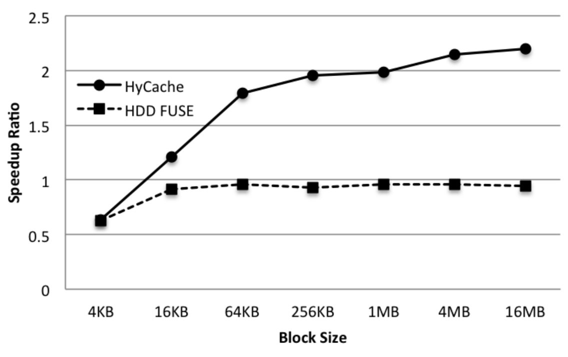


(b) Write Bandwidth

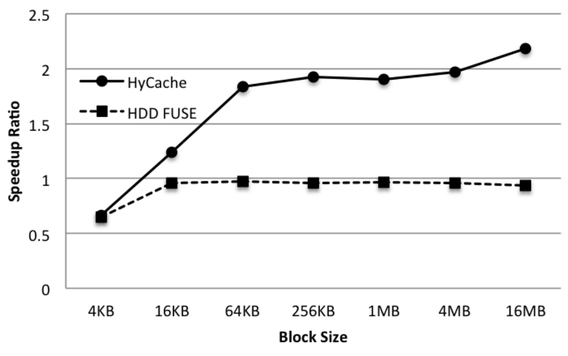
Fig. 7. IOzone bandwidth of 5 file systems.

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 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.

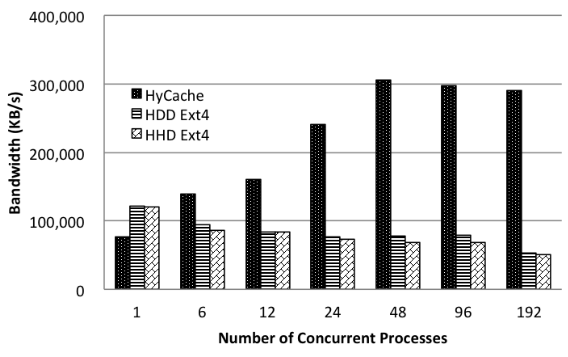


(a) Read Speedup

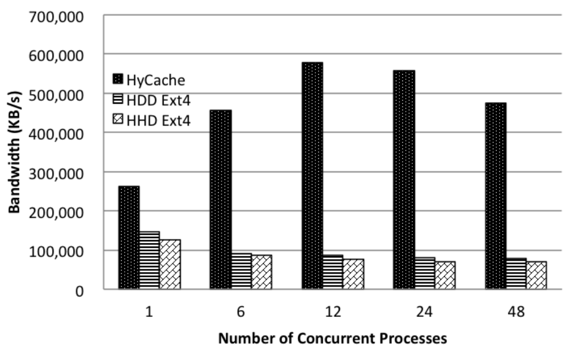


(b) Write Speedup

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



(a) 4KB Block



(b) 64KB Block

Fig. 9. Aggregate bandwidth of concurrent processes.

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.

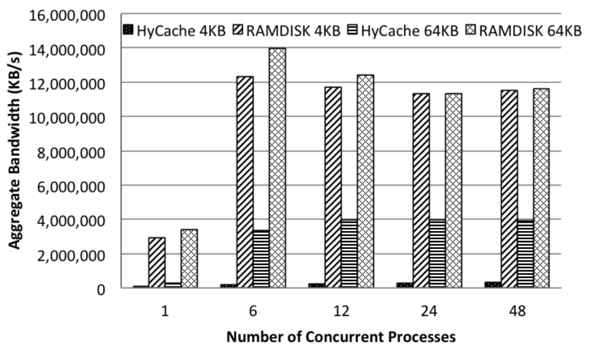
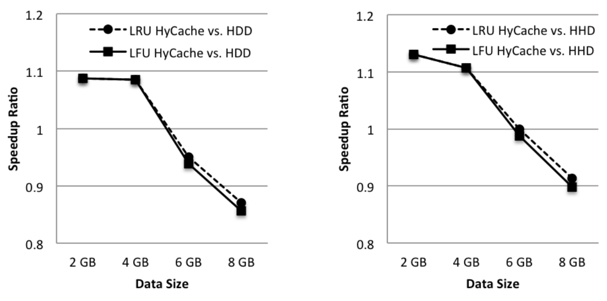


Fig. 10. Aggregate bandwidth of the FUSE implementation on RAMDISK.

C. Macro-benchmark

PostMark is 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 ($< 2\%$). 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

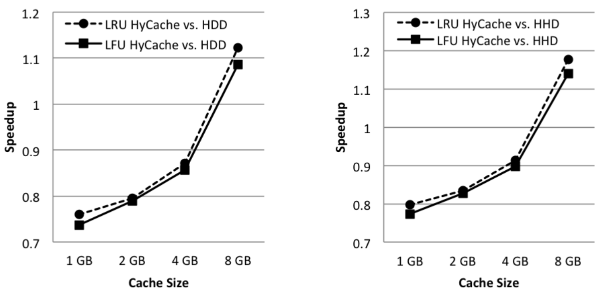
Fig. 11. PostMark: speedup of HyCache over Ext4 with 4 GB SSD cache.

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.

D. Application

We have run two real world applications on HyCache: MySQL and the Hadoop.

We install MySQL 5.5.21 with database engine MySIAM, and deploy TPC-H 2.14.3 databases. TPC-H is an industry standard benchmark for databases. By default it provides a



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

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

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.

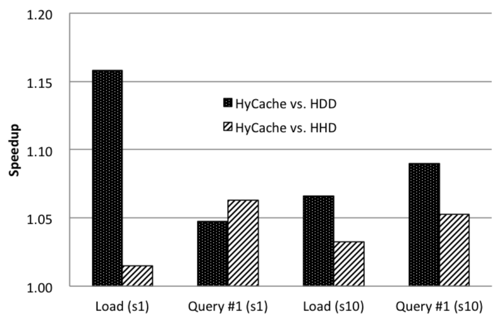


Fig. 14. TPC-H: speedup of HyCache over Ext4 on MySQL.

For HDFS we measure the bandwidth by concurrently copying a 1GB file per node from HDFS to the RAMDISK (i.e. $/dev/shm$). The results are reported in Table III, showing that HyCache helps improve HDFS performance by 28% at 32-node scales.

We also run the built-in ‘sort’ example as a real Hadoop application. The ‘sort’ application is to use map-reduce [14] to sort a 10GB file. We kept all the default settings in the Hadoop package except for the temporary directory which is specified as the HyCache mount point or a local Ext4 directory. The results are reported in Table III.

TABLE III
HDFS PERFORMANCE

	w/o HyCache	w/ HyCache	Improvement
bandwidth	114 MB/sec	146 MB/sec	28%
sort	2087 sec	1729 sec	16%

V. RELATED WORK

To the best of our knowledge, HyCache is the first user-level POSIX-compliant hybrid caching for distributed file systems. Some of our previous work [15–17] proposed data caching to accelerate applications by modifying the applications and/or their workflow, rather than the at the filesystem level. Other 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 ([18, 19]). Hystor [20] aims to optimize of the hybrid storage of SSDs and HDDs. However it requires to modify the kernel which might cause some issues. A more general multi-tiering scheme was proposed in [21] which helps decide the needed numbers of SSD/HDDs and manage the data shift between SSDs and HDDs by adding a ‘pseudo device driver’, again, in the kernel. iTransformer [22] considers the SSD as a traditional transient cache in which case data needs to be written to the spinning hard disk at some point once the data is modified in the SSD. iBridge [23] leverages SSD to serve request fragments and bridge the performance gap between serving fragments and serving large sub-requests. HPDA [24] offers a mechanism to plug SSDs into RAID in order to improve the reliability of the disk array. SSD was also proposed to be integrated to the RAM level which makes SSD as the primary holder of virtual memory [25]. NVMalloc [26] provides a library to explicitly allow users to allocate virtual memory on SSD. Also for extending virtual memory with Storage Class Memory (SCM), SCMFS [27] concentrates more on the management of a single SCM device. FAST [28] proposed a caching system to pre-fetch data in order to quicken the application launch. [29] considers SSD as a read-only buffer and migrate those random-writes to HDD.

VI. CONCLUSION AND FUTURE WORK

In this paper we addressed the long-existing issue with the bottleneck of local spinning hard drives in distributed file systems and proposed a cost-effective solution to alleviate this bottleneck, aimed at delivering comparable performance of an all SSD solution at a fraction of the cost. We proposed to add a middleware layer between the distributed file system and the underlying local file systems. We designed and implemented a user-level POSIX-compliant file system, HyCache, with high throughput, low latency, strong consistency, single namespace and multithread support. Non-privileged users can specify the cache size for different workloads without modifying the applications or the kernel. Our extensive performance evaluation showed that HyCache can be competitive with kernel-level file systems, and significantly improves the performance of the upper-level distributed file systems.

HyCache will be integrated into FusionFS [30] which is a high-performance distributed file system aimed at exascale computing, currently being developed by the authors of this work. The FusionFS file system also uses FUSE to provide a POSIX interface. We expect FusionFS to achieve an even more significant improvement in performance with HyCache than HDFS obtained, due to the fact that the FUSE overheads are already accounted for in FusionFS. FusionFS has already scaled to 1K nodes, and we aim to scale up FusionFS+HyCache to 10K nodes. We will also apply HyCache to Many-Task Computing (MTC) [31–34], which has specific emphasis on data-intensive computing [35] and cloud computing [36].

ACKNOWLEDGEMENT

This work was supported by the National Science Foundation (NSF) under grant OCI-1054974. The authors thank Dr. Xian-He Sun (IIT) for providing access to the 32-node cluster on which this work was evaluated. We also thank Dr. Zhiling Lan (IIT) for valuable discussions and feedback.

REFERENCES

[1] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. SOSP ’03, pages 29–43, New York, NY, USA, 2003. ACM.
[2] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. MSST ’10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.
[3] Alex Szalay, Julian Bunn, Jim Gray, Ian Foster, and Ioan Raicu. The importance of data locality in distributed computing applications. NSF Workflow Workshop, 2006.
[4] Steven W. Schlosser, John Linwood Griffin, John Linwood Grin, David F. Nagle, and Gregory R. Ganger. Filling the memory access gap: A case for on-chip magnetic storage. Technical report, 1999.
[5] Gordon Moore. Moore’s Law. *Intel Cooperation*, 1965.
[6] Mementus XT. http://www.seagate.com/www/en-us/products/internal-storage/momentus-xt-kit.
[7] FUSE Project. http://fuse.sourceforge.net.
[8] Stefan Podlipnig and Laszlo Boszormenyi. A survey of Web cache replacement strategies. *ACM Computing Surveys (CSUR), Volume 35 Issue 4*, 2003.
[9] D. Capps. IOzone Filesystem Benchmark. *www.iozone.org*, 2008.

[10] Jeffrey Katcher. Postmark: A new file system benchmark. In *Network Appliance, Inc.*, volume 3022, 1997.

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

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

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

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. OSDI, 2004.

[15] Ioan Raicu, et al. The quest for scalable support of data intensive workloads in distributed systems. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 207–216, New York, NY, USA, 2009. ACM.

[16] Ioan Raicu, Yong Zhao, Ian T. Foster, and Alex Szalay. Accelerating large-scale data exploration through data diffusion. In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, DADC '08, pages 9–18, New York, NY, USA, 2008. ACM.

[17] Ioan Raicu, Ian Foster, Alex Szalay, and Gabriela Turcu. AstroPortal: A science gateway for large-scale astronomy data analysis. In *TeraGrid Conference*, June 2006.

[18] Shan Li and H.H. Huang. Black-box performance modeling for solid-state drive. MASCOTS, 2010.

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

[20] Feng Chen, David Koufaty, and Xiaodong Zhang. Hystor: Making the best use of solid state drives in high performance storage systems. ICS, 2011.

[21] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 20–20, Berkeley, CA, USA, 2011. USENIX Association.

[22] Xuechen Zhang, Kei Davis, and Song Jiang. iTransformer: Using SSD to improve disk scheduling for high-performance I/O. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 715–726, Washington, DC, USA, 2012. IEEE Computer Society.

[23] Xuechen Zhang, Liu Ke, Kei Davis, and Song Jiang. iBridge: Improving unaligned parallel file access with solid-state drives. In *Proceedings of the 2013 IEEE 27th International Parallel and Distributed Processing Symposium*, IPDPS '13, Boston, MA, USA, 2013. IEEE Computer Society.

[24] Bo Mao, Hong Jiang, Dan Feng, Suzhen Wu, Jianxi Chen, Lingfang Zeng, and Lei Tian. HPDA: A hybrid parity-based disk array for enhanced performance and reliability. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[25] Anirudh Badam and Vivek S. Pai. SSDAlloc: hybrid SSD/RAM memory management made easy. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 16–16, Berkeley, CA, USA, 2011. USENIX Association.

[26] Chao Wang, Sudharshan S. Vazhkudai, Xiaosong Ma, Fei Meng, Youngjae Kim, and Christian Engelmann. NVMalloc: Exposing an aggregate SSD store as a memory partition in extreme-scale machines. In *IPDPS'12*, pages 957–968, 2012.

[27] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[28] Yongsoo Joo, Junhee Ryu, Sangsoo Park, and Kang G. Shin. FAST: Quick Application Launch on Solid-State Drives. In *FAST*, pages 259–272, 2011.

[29] Qing Yang and Jin Ren. I-CASH: Intelligently coupled array of SSD and HDD. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 278–289, Washington, DC, USA, 2011. IEEE Computer Society.

[30] Dongfang Zhao and Ioan Raicu. Distributed file systems for exascale computing. In *Doctoral Research Showcase, Supercomputing'12*, Salt Lake City, UT, 2012.

[31] Michael Wilde, Ioan Raicu, Allan Espinosa, Zhao Zhang, Ben Clifford, Mihael Hategan, Kamil Iskra, Pete Beckman, and Ian Foster. Extreme-scale scripting: Opportunities for large task parallel applications on petascale computers. In *SCIDAC, Journal of Physics: Conference Series 180. DOI*, pages 10–1088, 2009.

[32] Ioan Raicu. Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing. Doctoral Dissertation, the University of Chicago, 2009.

[33] Yong Zhao, Ioan Raicu, Ian T. Foster, Mihael Hategan, Veronika Nefedova, and Michael Wilde. *Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments*. book chapter in Grid Computing Research Progress. Nova Publisher, 2008.

[34] Ioan Raicu and Ian Foster, et al. Middleware support for many-task computing. *Cluster Computing, The Journal of Networks, Software Tools and Applications*, 13(3):291–314, 2010.

[35] Ioan Raicu, et al. *Towards Data Intensive Many-Task Computing*. book chapter in Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management. IGI Global Publishers, 2011.

[36] Yong Zhao, Xubo Fei, Ioan Raicu, and Shiyong Lu. Opportunities and challenges in running scientific workflows on the cloud. In *Proceedings of the 2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, CYBERC '11, pages 455–462, Washington, DC, USA, 2011. IEEE Computer Society.