Accepted in International Journal of Big Data Intelligence

Towards Cost-Effective and High-Performance Caching Middleware for Distributed Systems

Dongfang Zhao*, Kan Qiao*, 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@iit.edu, kqiao@hawk.iit.edu, iraicu@cs.iit.edu

Abstract—One performance bottleneck of distributed systems lies on the hard disk drive (HDD) whose single read/write head has physical limitations to support concurrent I/Os. Although the solid-state drive (SSD) has been introduced for years, HDDs are still dominant storage due to large capacity and low cost. This paper proposes a caching middleware that manages the underlying heterogeneous storage devices in order to allow distributed file systems to achieve both high performance and low cost. Specifically, we design and implement a user-level caching system that offers SSD-like performance at a cost similar to a HDD. We demonstrate how such a middleware improves the performance of distributed file systems, such as the HDFS. Experimental results show that the caching system delivers up to 7X higher throughput and 76X higher IOPS than Linux Ext4 file system, and accelerates HDFS by 28% on 32 nodes.

Index Terms—Distributed File Systems; User Level File Systems; Hybrid File Systems; Heterogeneous Storage; SSD.

1 Introduction

In the era of Big Data, applications' performance, from a system's perspective, is largely throttled by the I/O bottleneck and the storage subsystem. Scalable distributed filesystems, such as HDFS [1], emerge to meet the increasing need of I/O bandwidth. Yet, these storage solutions still assume the underlying device to be spinning hard disk drive (HDD) to keep cost low, as it is cost-prohibitive to replace inexpensive HDDs by the memory-class storage such as solid-state drives (SSD).

Modern HDDs usually have their own high-speed cache built in the device. These on-board caches, however, have two limitations. First, their size is extremely small comparing with the disk capacity. Second, the caching logic is implemented in the device controller and can be hardly leveraged by the application developers.

This paper proposes to deploy a user-level caching middleware between the distributed filesystem and local HDDs to achieve both low cost and high performance. In particular, we envision a memory-class storage such as SSD to coexist with HDD on each node. Such a middleware achieves low cost in the sense that the SSD size is small yet effective enough to store the "hot" data, rather than replacing the large-capacity HDDs.

Besides the high throughput offered by the memory-class storage, the proposed middleware attains two main objectives. First, file metadata need to be efficiently tracked. Second, files should be placed in accordance with applications' I/O patterns.

The metadata performance is often overlooked in modern data-intensive applications. As a case in point, HDFS only optimizes large files as its underlying data parallelism stems from the default 64MB data chunks. That is, HDFS splits the supposedly large file into several 64MB chunks, each of which is processed by a dedicated node; if the file is smaller than 64MB, HDFS does nothing more than a local filesystem such as Ext4. Due to the design assumption on processing large files, HDFS' single metadata server meets the needs of infrequent metadata operations. On the other hand, applications in high-performance computing (HPC) comprise many small- and medium-sized files, as Welch and Noer [2] reported that 25% – 90% of all the 600 million files from 65 Panasas [3] installations are 64KB or smaller. A large volume of small- or medium-size files call for the storage support of intensive metadata operations. We thus argue that the conventional wisdom of centralized metadata management needs to be revisited.

LRU is widely used as the de facto caching algorithm without a priori knowledge. We believe that in HPC systems where applications are well understood, heuristic caching algorithms are superior to LRU if applications' I/O patterns are taken into account. That is, we presume an application's I/O pattern is known *before* its execution. While this assumption is far too realistic in general, it is a reasonable (if not ubiquitous) practice in HPC applications. This is because many HPC applications are not directly executed by end users, but through workflow systems [4–7] that specify the task dependency automatically based on the high-level workflow description including the I/O workload.

In order to justify the effectiveness of our proposed architecture and design principles, we implement a system prototype of the caching middleware, deploy it in between HDFS and a 32-node Linux cluster, and evaluate its performance with both benchmarks and applications. We also report the I/O performance of the caching middleware on a leadership-class supercomputer at Los Alamos National Laboratory. Some preliminary results were previously presented at [8, 9]; the success of this caching work leads to another project (FusionFS [10–13]) aiming to build a fully-fledged file system to be deployed on all the compute nodes in modern supercomputers.

To summarize, this paper makes the following contributions:

- *Propose a caching layer to improve the I/O performance of large-scale HPC applications*
- *Devise novel metadata management to achieve excellent scalability for metadata-intensive workloads*
- *Design a heuristic file-placement algorithm with consideration of workloads' I/O patterns*
- *Implement a user-level distributed caching middleware for manipulating frequently-accessed data in distributed filesystems*
- *Evaluate the caching system with both benchmarks and applications, and report its effectiveness when deployed with HDFS*

The remainder of this paper is structured as follows. Section 2 introduces more background and detailed motivation of this work. We describe the overall design of the caching middleware in Section 3. Section 4 presents the middleware's scalable metadata management subsystem. We discuss the pattern-aware file placement in Section 5. Section 6 details the implementation of the caching middleware. We report the experimental results in Section 7. Section 8 reviews related work on caching and storage systems. We conclude this paper in Section 9.

2 Background and Motivation

One performance bottleneck of distributed filesystems, such as the Google File System [14] and the Hadoop Distributed File System [1], is the underlying mechanical HDD. There are mainly two types of imbalance associated to HDDs: (1) the imbalance between HDD's capacity and its bandwidth, and (2) the imbalance between HDD's performance and other components' performance.

To see the first imbalance within HDD, we observe much faster growth in HDD's storage capacity than its I/O bandwidth. That is, the data to be stored keeps increasing rapidly while the data transfer rate has barely improved. The main reason of HDD's slow improvement on bandwidth is due to its single physical read/write head: there is a physical limit on the movement of this component. On the other hand, HDD's areal density follows Moore's Law—roughly doubling every 18 months—making HDD capacity to increase exponentially. Therefore unless the conventional HDD design is fundamentally changed, the chance for its bandwidth to catch up the capacity increase is vanishing small. The consequence of this imbalance is that applications spend more time on I/O: more data but same transfer rate. Recall that I/O had been the performance bottleneck of many applications before the Big Data era, and this imbalance caused by HDD itself—in spite of its increasing capacity—makes it even worse.

The increasing imbalance between HDD and other system components is as challenging as, if not more challenging than, the one within HDD. For instance, the performance gap between HDD and main memory is in multiple orders of magnitude. High-end HDDs have 100–200MB/s peak bandwidth, while memory bandwidth ranges in order of 10GB/s. Making things worse, there is a growing disparity of speed between the CPU and memory—the so-called memory wall. CPU speed improves exponentially, and memory speed has largely fallen behind. Thus CPU and HDD are even more imbalanced due to the memory wall.

SSD has the potential to bridge the performance gap between memory and HDD, except that replacing all HDDs with SSDs is cost-prohibitive. In Table 1 we list that the per-GB cost of the high-end SSD OCZ RevoDrive is 41X higher than HDD (Hitachi Deskstar).

As a compromised solution, the industry recently introduces hybrid hard drives (HHD) where a small embedded SSD transparently buffers the hot data stored in the mechanical hard drive. For example, Seagate releases Momentus XT [15] that encapsulates both a 4GB SSD and a 500GB HDD into a single physical device. The advantage for such a HHD is the drop-in replacement to HDD. But its small fixed SSD cache (e.g. less than 1% of the overall capacity) limits its ability to accelerate a large number 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.

So we ask: how to boost a large variety of HPC applications' I/O performance with affordable cost? In fact, this cost-performance dilemma has been long existing in computer systems; and the conventional wisdom proves to be a viable solution—caching. For example there is a small built-in cache in HDD hardware to buffer the data from/to the memory; there are also 2 to 3 levels of caches on the CPU chip or the motherboard to buffer the data between CPU and the memory. Because these caches are all built in the hardware and transparent to the applications, and because they are all extremely small compared to the base medium, developers can hardly manipulate these caches to optimize their applications. Therefore one objective of this work is to build a user-level and highly-programmable SSD caching middleware to bridge the performance gap between memory and HDDs in distributed systems.

3 Design Overview

Fig. 1 shows the architectural overview of the storage hierarchy with the proposed middleware. Instead of

Table 1
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

being mounted directly on the local file systems (HDD or SSD mount point), distributed file systems are deployed on top of the middleware.

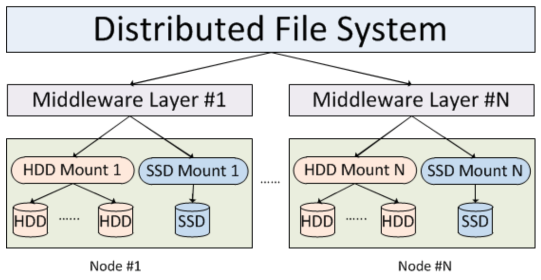


Figure 1. The storage hierarchy with a middleware between distributed file systems and local disks

One of our design principles is to make the middleware POSIX-compliant. This allows it to be integrated to other layers seamlessly, as most applications assume POSIX interfaces. The POSIX support, however, should not be confused with, or assumed in, its associated layers. For example, the native HDFS does not provide the POSIX interface to its applications, but it assumes the local filesystem on each node to comply with POSIX (e.g. Ext4). We leverage FUSE [16] to implement the POSIX interface, which will be discussed in Section 6.

Fig. 2 describes the middleware's three major 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 the replacement algorithm. Data manipulator manipulates data between two access points of fast- and regular-speed devices, respectively.

3.1 Request Handler

The request handler is the first component of the whole middleware system that interacts with distributed file systems. The `Virtual Root Path` can be any directory in a UNIX-like system as long as the end user has sufficient permissions on that directory. The virtual path is monitored by the FUSE kernel module, so any file operations on this mount point is passed to the FUSE kernel module. Then the FUSE kernel module imports the FUSE library and tries to transfer the request to FUSE API in the file dispatcher.

3.2 File Dispatcher

File dispatcher is the core component, as it redirects user-provided POSIX requests into customized handlers of file operations. FUSE only provides POSIX interfaces, and it is file dispatcher where these interfaces are implemented. Some of the most important file operations include `fopen()`, `fwrite()`, `fread()`, and `fclose()`. File dispatcher manages the file metadata, e.g. determining the file is manipulated on which physical node (inter-node metadata) and which disk (intra-node metadata). Metadata management detail will be discussed in Section 4.

Some replacement policies, i.e. cache algorithms, need to be provided to guide the file dispatcher for file placement. When the cache is full, the algorithm must choose which items to discard to make room for the new ones. For instance, 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. In addition to implementing the conventional LRU (Least Recently Used) and LFU (Least Frequently Used) [17], we design a heuristic caching mechanism crafted for HPC applications, which we will discuss in more detail in Section 5. It should be noted that the system is implemented in a loosely coupled fashion so that users are free to plug in their own favorable algorithms.

3.3 Data Manipulator

Data manipulator manipulates data between two logical access points: one for fast speed access (e.g. on SSD), and the other is for regular access (e.g. on HDD). 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. In this scenario, `A` is always the preferred point for any data request as long as it has enough space. For example, data need to be swapped back and forth between `A` and `B` once the space usage in `A` exceeds the threshold. Due to limited space, we only show two access points in the figure; there is nothing that architecturally prohibits us from leveraging more than two levels of access points.

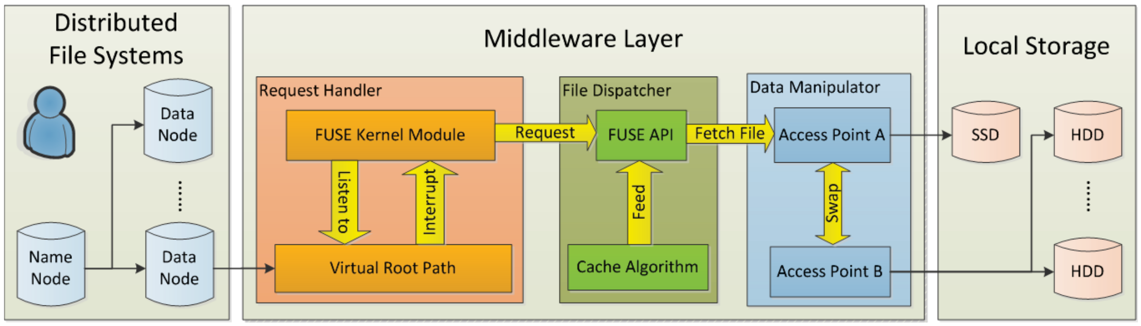


Figure 2. Three major components of the middleware: request handler, file dispatcher, and data manipulator

4 Two-Dimensional Metadata Management

There are two types of metadata communication in the distributed caching system. As shown in Fig. 3, a "horizontal" interaction is achieved by a distributed hash table, while a "vertical" file swap is mapped by a symbolic link. The hash table deals with the global namespace of all the cached data, and is agnostic about the internal file swap within the local node. Similarly, the symbolic link manages the local file placement according to the caching algorithm and has nothing to do with the hash table.

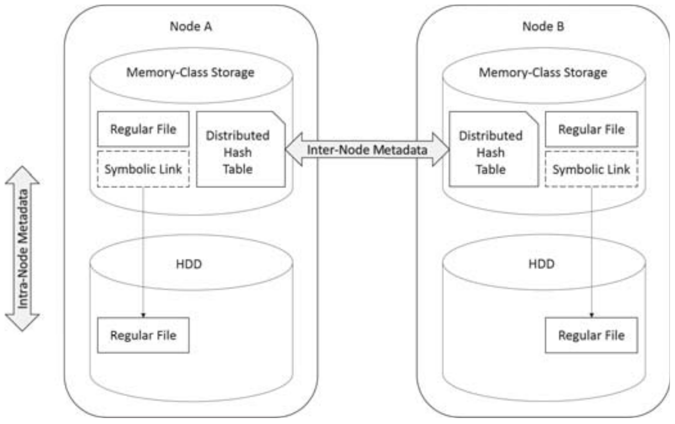


Figure 3. Two-dimensional metadata management

4.1 Inter-Node Metadata through Distributed Hash Table

Each participating node has a coherent view of all the files no matter if the file is stored in the local node or a remote node. The coherent view, or global namespace, is maintained by a distributed hash table (DHT [18, 19]), which disperses partial metadata on each node. As shown in Fig. 4, in this example `Node 1` and `Node 2` store two subgraphs (the top-left and top-right portions of the figure) of the entire metadata graph. The client could interact with the DHT to inquiry any file on any node, as shown in the bottom portion of the figure. Because the global namespace is just a logical view for clients, and it does not physically exist in any data structure, the global namespace does not need to be aggregated or flushed when changes occur to the subgraph on local compute nodes. The changes to the local metadata storage is exposed to the global namespace when the client queries the DHT.

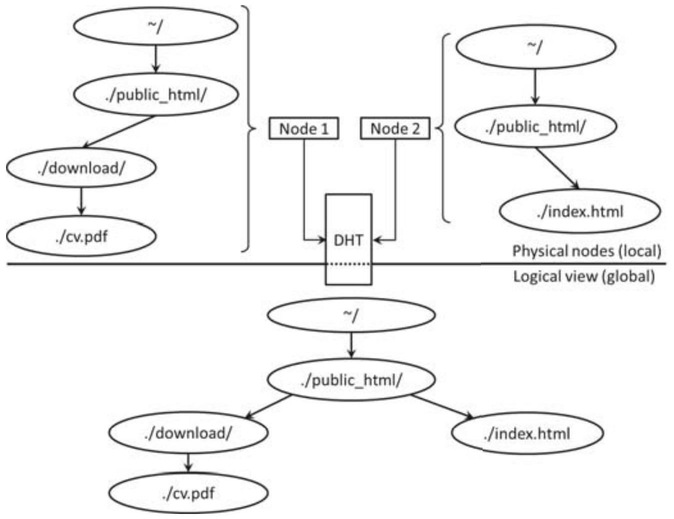


Figure 4. Metadata in the local nodes and the global namespace

Regular files and directories are managed by different data structures. For a regular file, the field `addr` stores the node where this file resides. For a directory, there is a field `filelist` to record all the entries under this directory. This `filelist` field is particularly useful for providing an in-memory speed for directory read, e.g. "ls /mnt/fusionfs". Nevertheless, both regular files and directories share some common fields, such as timestamps and permissions, which are commonly found in traditional i-nodes.

To make matters more concrete, Fig. 5 shows the distributed hash table according to the example metadata shown in Fig. 4. Note that the DHT is only a logical view of the aggregation of multiple partial metadata on local nodes (in this case, `Node 1` and `Node 2`). Five entries (three directories, two regular files) are stored in the DHT, with their file names as keys. The value is a list of properties delimited by semicolons. For example, the first and second portions of the values are permission flag and file size, respectively. The third portion for a directory value is a list of its entries delimited by commas, while for regular files it is just the physical location of the file, e.g. the IP address of the node on which the file is stored. Upon a client request, this value structure is serialized by Google Protocol Buffers [20] before sending over the network to the metadata server, which is just another compute node. Similarly, when the metadata blob is received by a node, we deserialize the blob back into the C structure with Google Protocol Buffers.

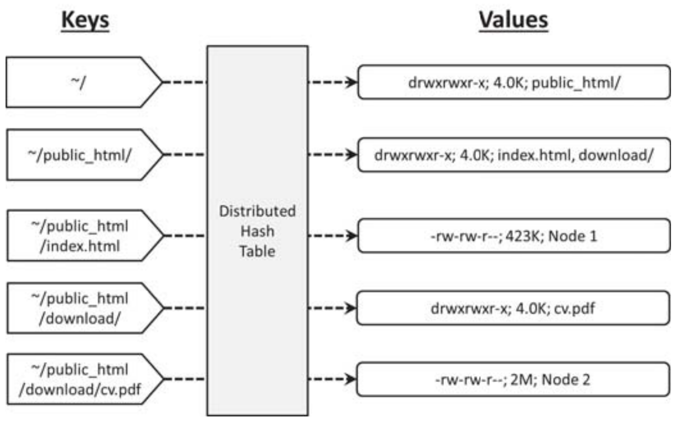


Figure 5. The global namespace abstracted by key-value pairs in a DHT

The metadata and data on a local node are completely decoupled: a regular file's location is independent of its metadata location. From Fig. 4, we know the `index.html` metadata is stored on `Node 2`, and the `cv.pdf` metadata is on `Node 1`. Nevertheless, it is perfectly fine for `index.html` to reside on `Node 1`, and for `cv.pdf` to reside on `Node 2`, as shown in Fig. 5.

Besides the conventional metadata information for regular files, there is a special flag (no shown in the figure) in the value indicating whether this file is being written. Specifically, any client who requests to write a file needs to sets this flag before opening the file, and will not reset it until the file is closed. The atomic compare-swap operation supported by DHT [18] guarantees the file consistency for concurrent writes.

Another challenge on inter-node metadata management is on the large directory. When a large number of clients write many small files on the same directory concurrently, the value of this directory in the key-value pair gets incredibly long, which causes its responsiveness extremely slow. This is because a client needs to update the entire old long string with the new one, even though the majority of the old string is unchanged. To address that, we employ an atomic append operation that asynchronously appends the incremental change to the value. This approach is similar to Google File System [14], where files are immutable and can only be appended. This gives us excellent concurrent metadata modification in large directories, at the expense of potentially slower directory metadata read operations.

4.2 Local Metadata through Symbolic Links

Fig. 6 shows a typical scenario of file mappings when the space of SSD cache is intensive so some files need to be swapped into the HDD. End users are only aware of the middleware's virtual root path and every single file in the virtual directory is mapped to the underlying SSD physical directory. SSD has a limited space so when the usage is beyond a threshold the middleware needs to move some files from SSD to HDD and keeps symbolic links to the migrated files.

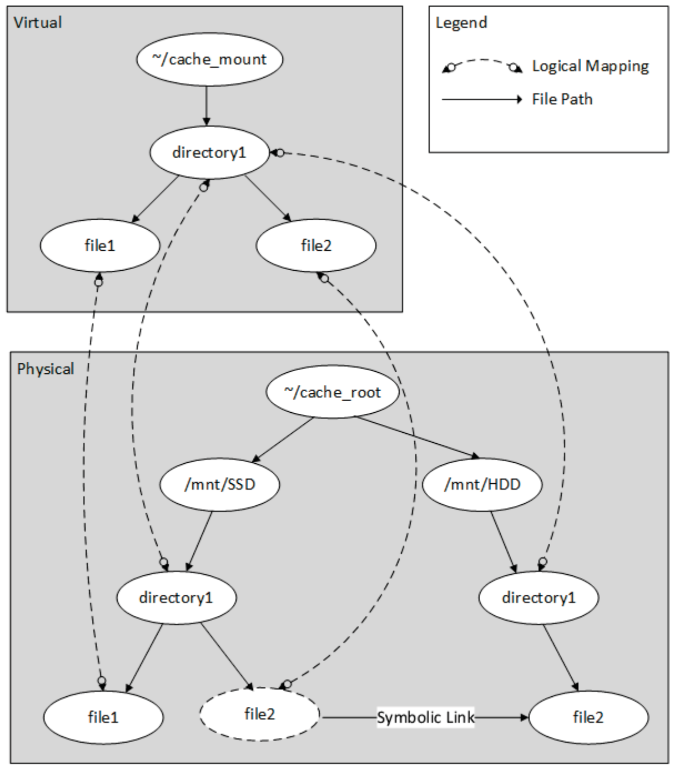


Figure 6. File movement within the local node: when free space of SSD cache is limited, based on the caching algorithm `file2` is evicted out of the SSD after which its symbolic link is created and kept in the SSD.

As an example, Algorithm 1 illustrates how a file is opened. The first thing is to check if the requested file is located 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 stale files are moved from SSD to HDD and the cache queue is updated accordingly. Then the symbolic link of the requested file is removed and the physical file 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.

Another important file operation that is worth mentioning is file removal. We explain how the middleware removes a file in Algorithm 2. Line 4 and Line 5 are

Algorithm 1 Open a file in the local cache

```
Input: 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
Output: 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
```

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 stored in HDD. If so, this regular file needs to be removed as well.

Algorithm 2 Remove a file in the local cache

```
Input: 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
Output: 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 a similar idea in Algorithm 1 and Algorithm 2: manipulate files in SSD and HDD back and forth to make users antagonistic about the underlying heterogeneous storage devices. Due to limited space, we will only present one more algorithm for file rename 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 the local cache

```
Input: 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
Output: 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
```

5 Pattern-Aware Heuristic File Placement

5.1 Problem Statement

The problem of finding optimal caching on multiple-disk is proved to be NP-hard [21]. A simpler problem on a single-disk setup has a polynomial solution [22], which is, unfortunately, too complex to be applied in real applications. An approximation algorithm was proposed in [23] with the restriction that each file size should be the same, which limits its use in practice.

In fact, at small scale (e.g. each node has $O(10)$ files to access), a brute-force solution with dynamic programming is viable, with the same idea of the classical problem of traveling salesman problem (TSP) [24] with exponential time complexity. Nevertheless, in real applications the number of accessed files could be 10,000 or more, which makes the dynamic programming approach unfeasible. Therefore we propose a heuristic algorithm of $O(n \lg n)$ (n is the number of distinct files on the local node) for each job, which is efficient enough for an arbitrarily large number of files in practice, especially when compared to the I/O time on the disk.

5.2 Assumptions and Notations

We assume a queue of jobs, and their requested files are known on each node in a given period of time, which could be derived from the job scheduler and the metadata information. This assumption is based on our observation of many workflow systems [4, 5], which implicitly make a similar assumption: users are familiar with the applications they are to run and they are able to specify the task dependency (often times automatically based on the high-level parallel workflow description). Note that the referenced files are only for the jobs deployed on the local node, because there is no need to cache the files that will be accessed by the jobs deployed on remote nodes.

The access pattern of a job is represented by a sequence $R = (r_1, r_2, \ldots, r_m)$, where each r_i indicates one access to a particular file. Note that the files referenced by different r_i's are possibly the same, and could be on the cache, or the disk. We use $File(r_i)$ to indicate the file object which r_i references to. The size of the referenced file by r_i is denoted by $Size(File(r_i))$. The *cost* is defined as the to-be-evicted file size multiplied by its access frequency after the current processing position in the reference sequence. The *gain* is defined as the to-be-cached file size multiplied by its access frequency after the current fetch position in the reference sequence. Since cache bandwidth is significantly higher than disk bandwidth, in our analysis we ignore the time of transferring data between the processor and the cache. Similarly, when the file is swapped between cache and disks, only the disk throughput is counted. The cache size on the local node is denoted by C, and the current set of files in the cache is denoted by S. Our goal is to minimize the total I/O cost of the disk by determining whether the accessed files should be placed in the cache.

5.3 Methodology

There are 3 rules to be followed in the proposed caching algorithms.

1) Every fetch should bring into the cache the very *next* file in the reference sequence if it is not yet in the cache.
2) Never fetch a file to the cache if the total cost of the to-be-evicted files is greater than the gain of fetching this file.
3) Every fetch should discard the files in the increasing order of their cost until there is enough space for the newly fetched file. If the cache has enough space for the new file, no eviction is needed.

We elucidate the above 3 rules with a concrete example. Assume we have a random reference sequence $R = (r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9)$. Let $File(r_1) = F_1$, $File(r_2) = F_2$, $File(r_3) = F_3$, $File(r_4) = F_4$, $File(r_5) = F_3$, $File(r_6) = F_1$, $File(r_7) = F_2$, $File(r_8) = F_4$, $File(r_9) = F_3$, and $Size(F_1) = 20$, $Size(F_2) = 40$, $Size(F_3) = 9$, $Size(F_4) = 40$. Let the cache capacity be 100. According to *Rule 1*, the first three files to be fetched to cache are (F_1, F_2, F_3). Then we need to decide if we want to fetch F_4. Let $Cost(F_i)$ be the cost of evicting F_i. Then we have $Cost(F_1) = 20 \times 1 = 20$, $Cost(F_2) = 40 \times 1 = 40$, and $Cost(F_3) = 9 \times 2 = 18$. According to *Rule 3*, we sort the costs in the increasing order (F_3, F_1, F_2). Then we evict the files in the sorted list, until there is enough room for the newly fetched file F_4 of size 40. In this case, we only need to evict F_3, so that the free cache space is $100 - 20 - 40 = 40$, just big enough for F_4. Before replacing F_3 by F_4, *Rule 2* is referred to ensure that the cost is smaller than the gain, which is true in this case by observing that the gain of prefetching F_4 is 40, larger than $Cost(F_3) = 18$.

5.4 Procedures

The caching procedure is presented in Algorithm 4, which is called when the i^{th} reference is accessed and $File(r_{i+1})$ is not in the cache. If $File(r_{i+1})$ is already in the cache, then it is trivial to keep processing the next reference, which is not explicitly mentioned in the algorithm. $File(r_{i+1})$ will not be cached if it is accessed only once (Line 2). Subroutine $GetFilesToDiscard()$ tries to find a set of files to be discarded in order to make more room to (possibly) accommodate the newly fetched file in the cache (Line 3). Based on the decision made by Algorithm 4, $File(r_{i+1})$ could possibly replace the files in D in the cache (Line 4 - 7). $File(r_{i+1})$ is finally read into the processor from the cache or from the disk, depending on whether $File(r_{i+1})$ is already fetched to the cache (Line 9).

Algorithm 4 Fetch a file to cache or processor

```
Input: $i$ is the reference index being processed
1:  procedure FETCH($i$)
2:      if $\{r_j | File(r_j) = File(r_{i+1}) \land j > i + 1\} \neq \emptyset$ then
3:          $flag, D \leftarrow GetFilesToDiscard(i, i + 1)$
4:          if $flag = successful$ then
5:              Evict $D$ out of the cache
6:              Fetch $File(r_{i+1})$ to the cache
7:          end if
8:      end if
9:      Access $File(r_{i+1})$ (either from the cache or the disk)
10: end procedure
```

The time complexity of Algorithm 4 is as follows. Line 2 takes $O(1)$ since it can be precomputed using dynamic programming in advance. $GetFilesToDiscard()$ takes $O(n \lg n)$ that will be explained when discussing Algorithm 5. Thus the overall time complexity of Algorithm 4 is $O(n \lg n)$.

The $GetFilesToDiscard()$ subroutine (Algorithm 5) first checks if the summation of current cache usage and the to-be-fetched file size is within the limit of cache. If so, then there is nothing to be discarded (Line 2 - 4). We sort the files by their increasing order of cost at Line 9, because we hope to evict out the file of the smallest cost. Then for each file in the cache, Lines 11 - 18 check if the gain of prefetching the file outweighs the associated cost. If the new cache usage is still within the limit, then we have successfully found the right swap (Lines 19 - 21).

We will show that the time complexity of Algorithm 5 is $O(n \lg n)$. Line 5 takes $O(1)$ to get the total number of occurrences of the referenced file. Line 9 takes $O(n \lg n)$ to sort, and Lines 10 - 22 take $O(n)$ because there would be no more than n files in the cache (Line 10) and Line 11 takes $O(1)$ to collect the file occurrences. Both Line 5 and Line 11 only need $O(1)$ because we can precompute those values by dynamic programming in advance. Thus the total time complexity is $O(n \lg n)$.

Algorithm 5 Get set of files to be discarded

Input: i is the reference index being processed; j is the reference index to be (possibly) fetched to cache

Output: $successful$ – $File(r_j)$ will be fetched to the cache and D will be evicted; $failed$ – $File(r_j)$ will not be fetched to the cache

```
1:  function GetFilesToDiscard($i, j$)
2:      if $Size(S) + Size(File(r_j)) \leq C$ then
3:          return $successful$, $\emptyset$
4:      end if
5:      $num \leftarrow$ Number of occurrences of $File(r_j)$ from $j + 1$
6:      $gain \leftarrow num \cdot Size(File(r_j))$
7:      $cost \leftarrow 0$
8:      $D \leftarrow \emptyset$
9:      Sort the files in $S$ in the increasing order of the cost
10:     for $F \in S$ do
11:         $tot \leftarrow$ Number of references of $F$ from $i + 1$
12:         $cost \leftarrow cost + tot \cdot Size(F)$
13:         if $cost < gain$ then
14:             $D \leftarrow D \cup \{F\}$
15:         else
16:             $D \leftarrow \emptyset$
17:             return $failed$,$D$
18:         end if
19:         if $Size(S \setminus D) + Size(File(r_j)) \leq C$ then
20:             break
21:         end if
22:     end for
23:     return $successful$, $D$
24: end function
```

6 Implementation

6.1 FUSE Framework

FUSE [16] is a framework to develop customized file system. FUSE module has been officially merged into the Linux kernel tree since kernel version 2.6.14 [25]. FUSE provides 35 interfaces to fully comply with POSIX file operations. The middleware thus implements each interface to support POSIX. Some of POSIX APIs are called more frequently e.g. those essential file operations such as `open()`, `read()`, `write()` and `unlink()`; others might be less popular or even remain optional in particular Linux distributions (e.g. `getxattr()` and `setxattr()` are to get and set extra file attributes, respectively).

6.2 User Interface

The local mount point of the middleware is not only a single local directory but a virtual entry point of two mount points for the SSD and HDD partitions, respectively. Fig. 7 shows how to mount the middleware in a UNIX-like system.

Suppose the middleware was mounted on a local directory called `cache_mount`, and another local directory (e.g. `cache_root`) was created and had two subdirectories: the mount point of the SSD partition and the mount point of the HDD partition. Users would execute `./cache_bin <root> <mount>` where `cache_bin` is the binary executable of the middleware, `root` is the physical directory, and `mount` is the virtual directory.

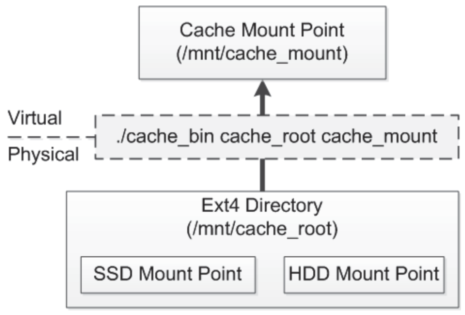


Figure 7. How to mount the caching middleware in a UNIX-like machine

6.3 Strong Consistency

We keep 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. There are two advantages to choose symbolic links for the local file swapping. First, symbolic is persistent, which means we need not spend extra cost to flush the files from memory to the hard disk. Second, symbolic link is natively supported by both the Linux kernel and the FUSE framework.

6.4 Data Granularity

The caching middleware manipulates 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 into smaller chunks. For example in HDFS, an arbitrarily large file is often split into 64MB chunks. Thus what the middleware deals with on the local node is a few 64MB chunks, which can be perfectly fit in a mainstream SSD device.

6.5 Multithread Support

The middleware implementation supports multithreading to leverage the many-core architecture in most high performance computers. Users, however, have the option to disable this feature to run applications in the single-thread mode. Although there are cases where multithreading does not help and only introduces overheads from switching contexts, by default multithreading is enabled 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.

7 Evaluation

7.1 Experiment Setup

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 memory. The HDD is Seagate Barracuda 1 TB, the SSD is OCZ RevoDrive 100 GB (peak performance 676 MB/s), and 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.

Some of the large-scale experiments are conducted on Kodiak [26], which is a 1024-node cluster at Los Alamos National Laboratory. Each Kodiak node has an AMD Opteron 252 CPU (2.6GHz), 4GB memory, and two 7200rpm 1TB hard disk drives.

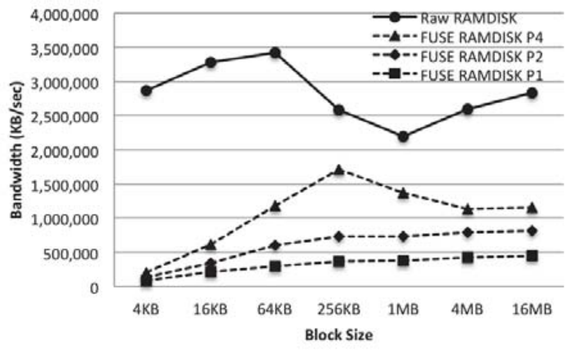
In the remainder of this paper we will use terms throughput and bandwidth interchangeably, which basically means the rate of data transferring. Unless otherwise specified all bandwidths are with respect to sequential read and write operations. All the results are averages of at least 3 stable (i.e. within 5% difference) numbers.

7.2 FUSE overhead

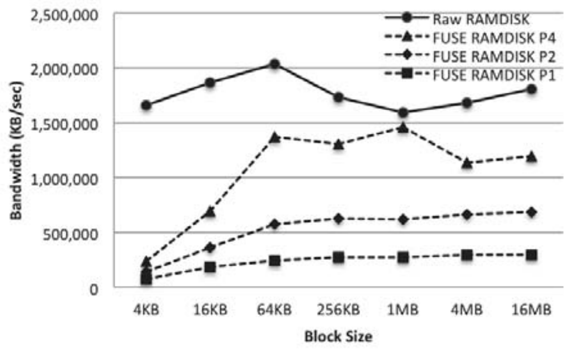
In order to quantify the overhead introduced by FUSE, we compare the I/O performance between raw RAMDISK (i.e. tmpfs [27]) and a simple FUSE file system mounted on RAMDISK. By experimenting on RAMDISK we completely eliminate all factors affecting performance particularly from the HDD and disk controller. Since all the I/O tests are essentially done in 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 Fig. 8. Moreover, the performance of concurrent FUSE processes are plotted, which shows that FUSE has a good scalability with respect to the number of concurrent processes.

In the case of single-process I/O, there is a significant performance gap. The read and write bandwidth on RAMDISK are in the order of gigabytes, whereas when mounting FUSE we could only achieve bandwidth below 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. Nevertheless, we will see in the following subsections that even with FUSE overhead on SSD, the caching middleware still significantly outperforms traditional HDD (Fig. 14).



(a) Read Bandwidth



(b) Write Bandwidth

Figure 8. 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.

7.3 Caching Efficiency

This experiment explores that the caching middleware achieves a high efficiency of the underlying SSD hardware. Meanwhile, we compare the caching throughput with a pure HDD solution and demonstrate the superior performance of the proposed caching mechanism.

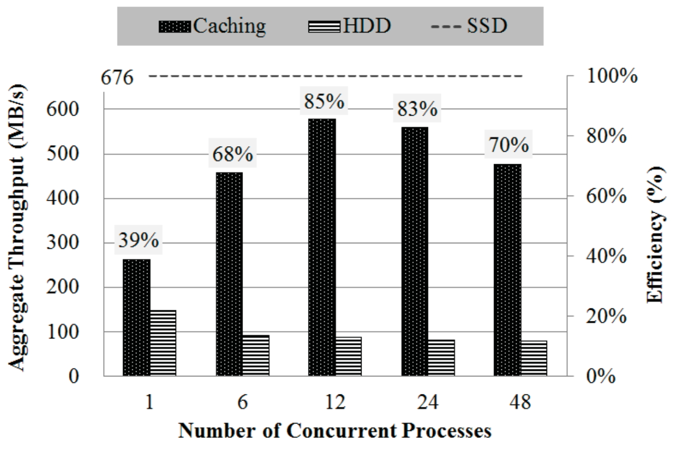


Figure 9. The I/O throughput and efficiency of the caching middleware comparing with pure SSD and pure HDD

Figure 9 shows that the caching middleware could achieve 580 MB/sec aggregate bandwidth (at 12 concurrent processes, the same number of hardware threads of the test bed) for concurrent data accesses, which is about 85% of the bandwidth of the raw SSD device (i.e. SSD Ext4). Therefore, the caching performance, even though limited by the FUSE overhead to some degree, is significantly higher than the conventional wisdom of a pure HDD solution. Note that, with more concurrent accesses the HDD performance actually degrades due to the single I/O head (for example, 85 MB/s on 12 processes).

7.4 Metadata

We show how the caching middleware improves HDFS metadata performance on Kodiak [26]. For both systems (caching middleware and vanilla HDFS), we have each node create (i.e. "touch") a large number of empty files (with unique names), and we measure the number of files created per second. In essence, each touched file incurs a metadata operation.

The aggregate metadata throughput of different scales is reported in Fig. 10. The performance gap is more than 3 orders of magnitude. Note that, HDFS starts to flatten out from 128 nodes, while the caching middleware keeps doubling the throughput all the way to 512 nodes, ending up with almost 4 orders of magnitude speedup (509022 vs. 57). HDFS performs poorly for metadata-intensive workloads mainly because of its centralized metadata server, whereas the caching middleware employs a distributed metadata management.

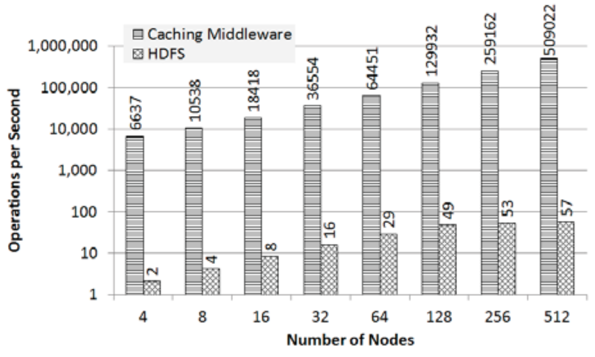


Figure 10. Metadata performance on Kodiak

7.5 Heuristic Caching

We plug the heuristic caching and LRU algorithms into the middleware, and evaluate their performance at 512-node scale. We create different sizes of files, randomly between 6MB and 250MB, and repeatedly read these data in a round-robin manner. The local cache size is set to 256MB.

The execution time of both algorithms is reported in Fig. 11. Heuristic caching clearly outperforms LRU at all scales, mainly because LRU does not consider the factors such as file size and cost-gain ratio, which are carefully taken into account in heuristic caching. In particular, heuristic caching outperforms LRU by 29X speedup at I/O size = 64,000GB (3,009 seconds vs. 86,232 seconds).

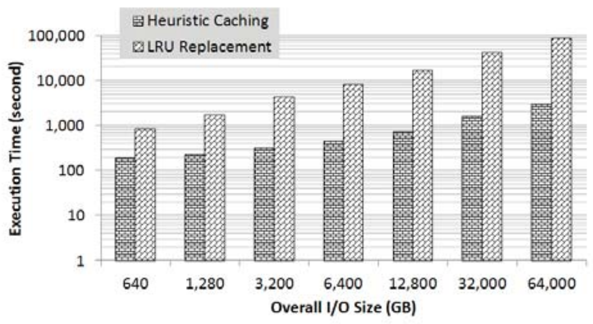


Figure 11. Comparison between Heuristic Caching and LRU

7.6 Benchmarks

IOzone [28] is a general filesystem 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, and so forth. 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.

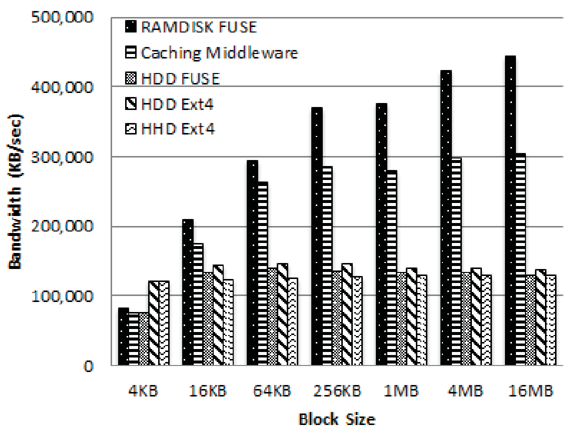
Fig. 12 shows the throughput with a variety of block sizes ranging from 4 KB to 16 MB. For each block size we show five bandwidths from the left to the right: 1) the theoretical bandwidth upper bound (obtained from RAMDISK), 2) caching middleware, 3) a simple FUSE file system accessing a HDD, 4) HDD Ext4 and 5) HHD Ext4.

Fig. 12(a) shows that the read throughput on the middleware is about doubled comparing with the native Ext4 file system for most block sizes. In particular, when block size is 16 MB the peak read throughput on the caching middleware is over 300 MB/s, which is 2.2X higher than Ext4 on HDD as shown in Fig. 13(a).

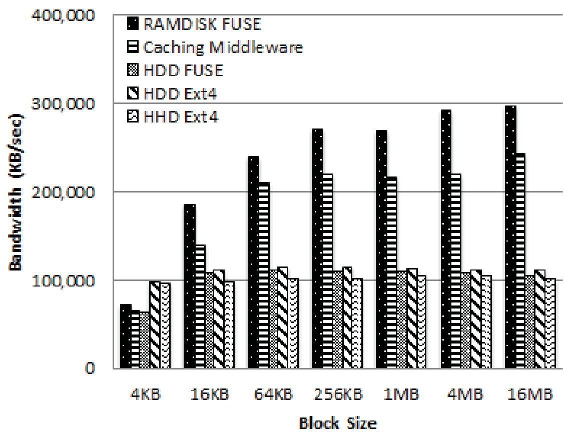
As for the overhead of FUSE framework compared to the native Ext4 file system on HDD we see FUSE only adds little overhead to file reads at all block sizes as shown in Fig. 13(a): for most block sizes FUSE achieves nearly 100% performance of the native Ext4. Similar results are also reported in a review of FUSE performance in [29]. This implies that even when the SSD cache usage is intensive and some files need to be swapped from SSD to HDD, the overall system performance keeps comparable to raw Ext4.

Fig. 12 also shows that the commercial HHD product performs at about the same level of the HDD. This is primarily due to a small and inexpensive SSD. It confirms our previous conjecture on the effectiveness of this approach.

We see a similar trend of file writes in Fig 12(b) as file reads. Again, the caching middleware is about twice

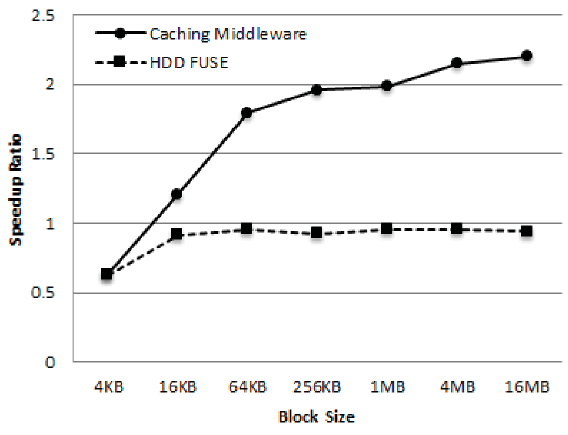


(a) Read Bandwidth

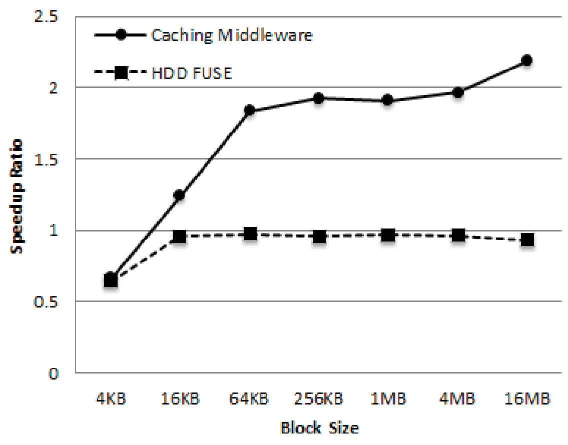


(b) Write Bandwidth

Figure 12. IOzone bandwidth of five file systems



(a) Read Speedup



(b) Write Speedup

Figure 13. Caching Middleware and FUSE speedup over HDD Ext4

as fast compared to Ext4 on HDDs for most block sizes. The peak write bandwidth (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 Fig. 13(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.

Fig. 13 shows that for small block size (e.g. 4 KB) the caching middleware 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, where the context switches of FUSE dominate the performance. Fortunately in most cases this small block size is more generally used for random read and write of small pieces of data (i.e. IOPS) rather than high-throughput applications.

Table 2 shows that the caching middleware has a far higher IOPS than other Ext4. In particular, it has about 76X IOPS as traditional HDD. The SSD portion of the HHD device (e.g. 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 also explains why the IOPS of the HHD lands in the same level of HDD rather than SSD.

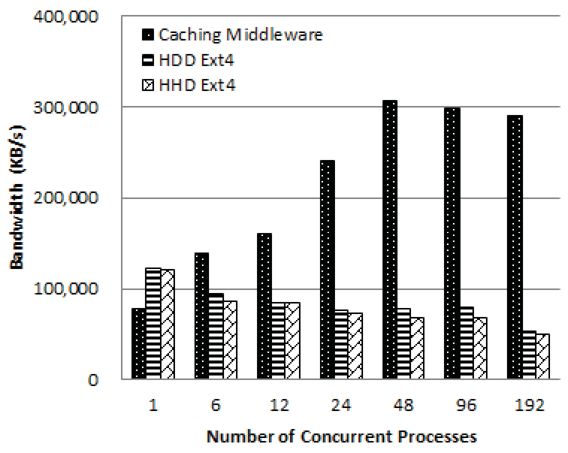
Table 2
IOPS of different file systems

Caching Middleware	HDD Ext4	HHD Ext4
14,878	195	61

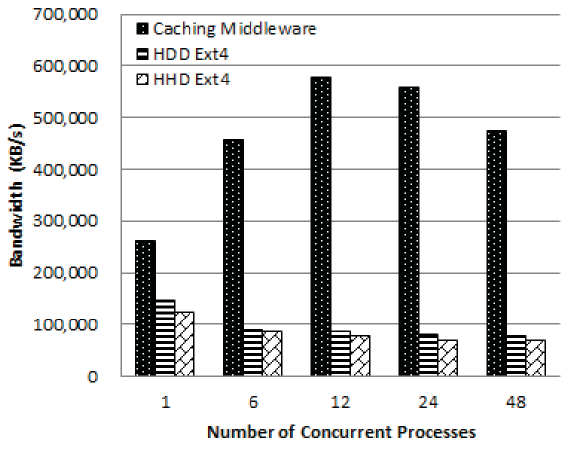
The caching middleware takes advantages of the multicore's concurrent tasking that delivers a significantly higher aggregate throughput. The point is that the caching middleware avoids reading or writing directly on the HDD so it handles multiple I/O requests concurrently. In contrast, traditional HDD only has a single number of heads for read and write operations.

Fig. 14 shows that the caching middleware has almost linear scalability with respect to 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), while the traditional Ext4 has degraded performance when dealing with concurrent I/O requests. The largest gap is when there are 12 concurrent processes for 64KB block (578

MB/s for the middleware and 86 MB/s for HDD): the caching middleware delivers 7X higher throughput than Ext4 on HDD.



(a) 4KB Block



(b) 64KB Block

Figure 14. Aggregate bandwidth of concurrent processes

The upper bound of aggregate throughput is limited by the SSD device rather than our middleware implementation. This can be justified by Fig 15 where the middleware is deployed on RAMDISK. The performance of raw RAMDISK are plotted as the baseline. We 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 bottleneck: it does not limit the I/O speed unless the device is slow. In other words, the middleware can be applied to any fast storage devices in future as long as the workloads have enough concurrency to allow FUSE to harness multiple computing cores.

7.7 Applications

We install MySQL 5.5.21 with database engine MySIAM, and deploy TPC-H 2.14.3 databases [30]. By default TPC-H provides a variety size of databases (e.g. scale 1 for 1 GB, scale 10 for 10 GB, scale 100 for 100GB) each of

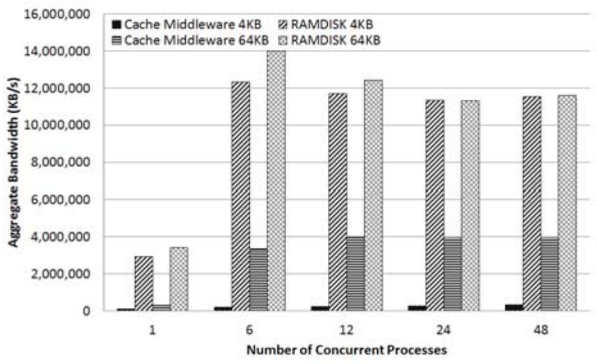
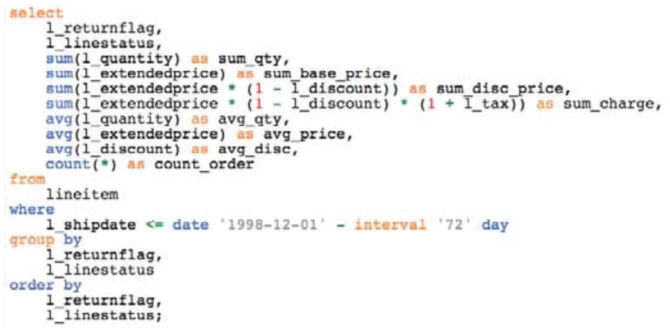


Figure 15. Aggregate bandwidth of the FUSE implementation on RAMDISK

which has eight tables. Furthermore, TPC-H provides 22 queries (i.e. Query #1 to Query #22) that are comparable to real-world business applications. Fig. 16 shows Query #1 that 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;
```

Figure 16. TPC-H: Query #1

To evaluate the write throughput of the middleware, we load table `lineitem` at scale 1 (600 MB) and scale 100 (6 GB) in the following three systems: caching middleware, HDD Ext4 and HHD Ext4. For read throughput, we run `Query #1` at scale 1 and scale 100. Experimental results are reported in Fig. 17, showing the middleware speedup the application by more than 15%.

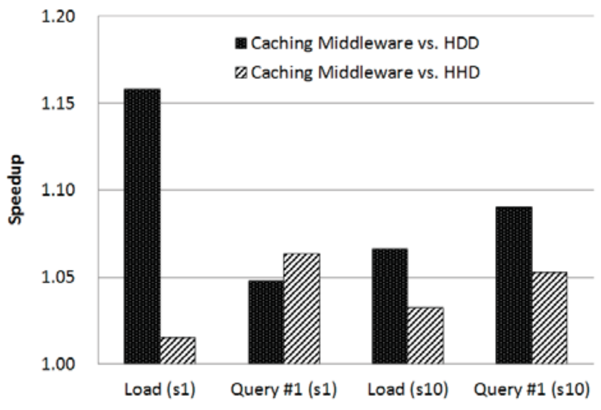


Figure 17. TPC-H: speedup of caching middleware over Ext4 for MySQL

For HDFS we measure the bandwidth by concurrently copying a 1GB file per node from HDFS to the RAMDISK (i.e. `/dev/shm`). We use the default replication number of HDFS since this paper is not focused on fault tolerance; our prior work [31] detailed a discussion on HDFS performance under failures though. The results are reported in Table 3, showing that the caching helps improve HDFS performance by 28% on 32 nodes.

We also run the built-in 'sort' utility in Hadoop. The 'sort' application uses MapReduce [32] to sort a 10GB file. We keep all the default settings in the Hadoop package except for the temporary directory that is specified as the caching mount point (and compare it with the local Ext4 directory). Results are reported in Table 3.

Table 3
HDFS Performance

Item	W/O Caching	W/ Caching	Improvement
Bandwidth	114 MB/sec	146 MB/sec	28%
Sort	2087 sec	1729 sec	16%

8 Related Work

Some recent work [33–35] proposed data caching to accelerate applications by modifying the applications or their workflows, rather than 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 these concerns would limit the system's applicability to end users. In contrast, this paper showcases a practical storage solution at user level.

There are a lot of work on the performance comparison between SSD and HDD in more perspectives such as [36, 37]. Of note, Hystor [38] aims to optimize hybrid storage of SSDs and HDDs. It, however, requires to modify the kernel that might not be desirable in many applications. A more general multi-tier scheme was proposed in [39] to decide the needed numbers of SSDs and HDDs, and to manage the data migration between SSDs and HDDs by adding a 'pseudo device driver', again, in the kernel. iTransformer [40] 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 [41] leverages SSD to serve request fragments and bridge the performance gap between serving fragments and serving large subrequests. HPDA [42] 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 [43]. NVMalloc [44] provides a library to explicitly allow users to allocate virtual memory on SSD. Also for extending virtual memory with Storage Class Memory (SCM), SCMFS [45] concentrates more on the management of a single SCM device. FAST [46] proposed a caching system to pre-fetch data in order to quicken the application launch. Yang et al. [47] consider SSD as a read-only buffer and directly write files on to HDD. None of the aforementioned work, however, considers SSD as a *persistent* caching middleware into the distributed filesystem, as demonstrated by this work.

Some prior work (e.g. [48], [33]) focused on job scheduling in order to improve applications' performance; this paper, however, achieves the same objective from the storage's perspective. That is, previous work was a top-down approach to manipulate jobs without much knowledge of the underlying storage, while this work shows a bottom-up approach to allow users to take advantage of storage's awareness of data locality by providing the portable POSIX interface. Moreover, in [33] it discussed different strategies broadly to showcase how to achieve different criteria such as data locality, load balance, or both, but this paper concentrates on detailing the scheduling and caching algorithms to (heuristically) minimize the overhead of the distributed storage.

There are extensive studies on leveraging data locality for effective caching. Block Locality Caching (BLC) [49] captures the backup and always uses the latest locality information to achieve better performance for data deduplication systems. The File Access corRelation Mining and Evaluation Reference model (FARMER) [50] optimizes the large scale file system by correlating access patterns and semantic attributes. Another research work proposes an Availability-aware DAta PlacemenT (ADAPT) [51] strategy to improve the application performance without extra storage cost, reducing network traffic, improving data locality, and optimizing application performance. In contrast, the caching middleware developed in this work achieves data locality with a unique mix of two principles: (1) write is always local, and (2) read locality depends on the job scheduler and a distributed hash table for load balance.

A thorough review of classical caching algorithms on large scale data-intensive applications is recently reported in [52]. The caching middleware is different from the classical cooperative caching [53] in that it assumes persistent underlying storage and manipulates data at the file-level. As an example of distributed caching for distributed file systems, Blue Whale Cooperative Caching (BWCC) [54] is a read-only caching system for cluster file systems. In contrast, the caching system this paper proposes is a POSIX-compliant I/O storage middleware that transparently interacts with the file systems.

Although the focus of this paper lies on the 2-layer hierarchy (i.e. SSD, HDD), the idea and methodology is applicable to multi-tier caching architecture as well. Multi-level caching gains much research interest, especially in the emerging age of cloud computing where the hierarchy of (distributed) storage is being redefined with more layers. For example Hint-K [55] caching is proposed to keep track of the last K steps across all the cache levels, which generalizes the conventional LRU-K algorithm concerned only on the single level information.

While this work represents a pure software solution for distributed cache, some orthogonal work focuses on improving caching from the hardware perspective. In [56], a hardware design is proposed with low overhead to support effective shared caches in multicore processors. For shared last-level caches, COOP [57] is proposed to only use one bit per cache line for re-reference prediction and optimize both locality and utilization. The REDCAP project [58] aims to logically enlarge the disk cache by using a small portion of main memory, so that the read time could be reduced. For SSD devices, a new algorithm called lazy adaptive replacement cache [59] is proposed to improve the cache hit and prolong the SSD lifetime.

Power-efficient caching has drawn a lot of research interests. It is worth mentioning that this work aims to better meet the need of high I/O performance for HPC systems, and power consumption is not a major design criterion at this point. Nevertheless, it should be noted that power consumption is indeed one of the toughest challenges to be overcome in future systems. One of the earliest work [60] tried to minimize the energy consumption by predicting the access mode and allowing cache accesses to switch between the prediction and the access modes. Recently, a new caching algorithm [61] is proposed to save up to 27% energy and reduce the memory temperature up to 5.45°C with negligible performance degradation. EEVFS [62] provides energy efficiency at the filesystem level with an energy-aware data layout and the prediction on disk idleness.

Caching has been extensively studied in different subjects and fields besides high-performance computing. In cloud storage, Update-batched Delayed Synchronization (UDS) [63] reduces the synchronization cost by buffering the frequent and short updates from the client and synchronizing with the underlying infrastructure in a batch fashion. For continuous data (e.g. online video), a new algorithm called Least Waiting Probability (LWP) [64] is proposed to optimize the newly defined metric called user waiting rate. In geoinformatics, the method proposed in [65] considers both global and local temporal-spatial changes to achieve high cache hit rate and short response time.

9 Conclusion

In this paper we address the long-existing issue with the I/O bottleneck on HDDs for distributed filesystems. We propose a cost-effective solution to alleviate this bottleneck, which delivers comparable performance of an all-SSD solution at a fraction of the cost. We design and implement a caching middleware between the distributed filesystem and the underlying local filesystems to demonstrate the validity of the proposed approach. Experimental results justify that the caching system delivers significantly higher throughput for a variety of benchmarks and applications running on distributed filesystems.

In future, we will integrate the proposed caching mechanism to other systems such as file compression [66, 67], data provenance [68, 69] and job scheduling [70]. We plan to further investigate the tradeoff between performance (for example, GPU acceleration [71]) and cost (for example, scientific applications on EC2 [72]) with the introduction of memory-class cache, and explore the viability to extend the current approach into incremental mechanisms [73–75].

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] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies*, 2010.

[2] Brent Welch and Geoffrey Noer. Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions. In *Mass Storage Systems and Technologies, 2013 IEEE 29th Symposium on*, 2013.

[3] David Nagle, Denis Serenyi, and Abbie Matthews. The Panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *Proceedings of ACM/IEEE Conference on Supercomputing*, 2004.

[4] Yong Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Services, IEEE Congress on*, 2007.

[5] Yong Zhao, Xubo Fei, I. Raicu, and Shiyong Lu. Opportunities and challenges in running scientific workflows on the cloud. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on*, 2011.

[6] Catalin Dumitrescu, Ioan Raicu, and Ian Foster. Experiences in running workloads over grid3. In *Grid and Cooperative Computing (GCC)*, volume 3795 of *Lecture Notes in Computer Science*, 2005.

[7] Michael Wilde, Ioan Raicu, Allan Espinosa, Zhao Zhang, Ben Clifford, Mihael Hategan, Sarah Kenny, Kamil Iskra, Pete Beckman, and Ian Foster. Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers. *Journal of Physics: Conference Series*, 180(1), 2009.

[8] Dongfang Zhao, Kan Qiao, and Ioan Raicu. Hycache+: Towards scalable high-performance caching middleware for parallel file systems. In *Proceedings of IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014.

[9] Dongfang Zhao and Ioan Raicu. HyCache: A user-level caching middleware for distributed file systems. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, 2013.

[10] Dongfang Zhao and Ioan Raicu. Storage support for data-intensive applications on extreme-scale hpc systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14), doctoral showcase*, 2012.

[11] Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, Dries Kimpe, Philip Carns, Robert Ross, and Ioan Raicu. FusionFS: Toward supporting data-intensive scientific applications on extreme-scale distributed systems. In *Proceedings of IEEE International Conference on Big Data*, 2014.

[12] Dongfang Zhao and Ioan Raicu. Distributed file systems for exascale computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12), doctoral showcase*, 2012.

[13] Dongfang Zhao, Da Zhang, Ke Wang, and Ioan Raicu. Exploring reliability of exascale systems through simulations. In *Proceedings of the 21st ACM/SCS High Performance Computing Symposium (HPC)*, 2013.

[14] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *ACM Symposium on Operating Systems Principles*, 2003.

[15] Mementus XT. http://www.seagate.com/www/en-us/products/internal-storage/momentus-xt-kit, 2014.

[16] FUSE. http://fuse.sourceforge.net, Accessed September 5, 2014.

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

[18] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, and Ioan Raicu. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing*, 2013.

[19] Tonglin Li, Raman Verma, Xi Duan, Hui Jin, and Ioan Raicu. Exploring distributed hash tables in highend computing. *SIGMETRICS Perform. Eval. Rev.*, 39(3), December 2011.

[20] Protocol Buffers. http://code.google.com/p/protobuf/, Accessed September 5, 2014.

[21] Christoph Ambühl and Birgitta Weber. Parallel prefetching and caching is hard. In *STACS*, 2004.

[22] Susanne Albers, Naveen Garg, and Stefano Leonardi. Minimizing stall time in single and parallel disk systems. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998.

[23] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. *SIGMETRICS Perform. Eval. Rev.*, 23(1), May 1995.

[24] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *J. ACM*, 9(1), January 1962.

[25] Linux.com. http://archive09.linux.com/feature/49757, 2014.

[26] Kodiak. https://www.nmc-probe.org/wiki/machines:kodiak, Accessed September 5, 2014.

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

[28] IOZone. http://www.iozone.org, 2014.

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

[30] TPCH Benchmark. http://www.tpc.org/tpch, 2014.

[31] Hui Jin, Kan Qiao, Xian-He Sun, and Ying Li. Performance under failures of mapreduce applications. In *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2011.

[32] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of USENIX Symposium on Opearting Systems Design & Implementation*, 2004.

[33] Ioan Raicu, Ian T. Foster, Yong Zhao, Philip Little, Christopher M. Moretti, Amitabh Chaudhary, and Douglas Thain. The quest for scalable support of data-intensive workloads in distributed systems. In *Proceedings of ACM International Symposium on High Performance Distributed Computing*, 2009.

[34] 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*, 2008.

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

[36] Shan Li and H.H. Huang. Black-box performance modeling for solid-state drives. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, 2010.

[37] Sanam 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.

[38] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the International Conference on Supercomputing*, 2011.

[39] 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*, 2011.

[40] 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*, 2012.

[41] 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*, 2013.

[42] 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*, 2010.

[43] 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*, 2011.

[44] 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 *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012.

[45] 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*, 2011.

[46] Yongsoo Joo, Junhee Ryu, Sangsoo Park, and Kang G. Shin. FAST: Quick application launch on solid-state drives. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, 2011.

[47] 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*, 2011.

[48] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde. Falkon: a fast and lightweight task execution framework. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[49] Dirk Meister, Jürgen Kaiser, and André Brinkmann. Block locality caching for data deduplication. In *Proceedings of the 6th International Systems and Storage Conference*, 2013.

[50] Peng Xia, Dan Feng, Hong Jiang, Lei Tian, and Fang Wang. Farmer: a novel approach to file access correlation mining and evaluation reference model for optimizing peta-scale file system performance. In *Proceedings of the 17th international symposium on High performance distributed computing*, 2008.

[51] Hui Jin, Xi Yang, Xian-He Sun, and Ioan Raicu. Adapt: Availability-aware mapreduce data placement for non-dedicated distributed computing. In *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*, ICDCS '12, 2012.

[52] R. Fares, B. Romoser, Ziliang Zong, M. Nijim, and Xiao Qin. Performance evaluation of traditional caching policies on a large system with petabytes of data. In *Networking, Architecture and Storage (NAS), 2012 IEEE 7th International Conference on*, 2012.

[53] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4), December 2003.

[54] Liu Shi, Zhenjun Liu, and Lu Xu. Bwcc: A fs-cache based cooperative caching system for network storage system. In *Proceedings of the 2012 IEEE International Conference on Cluster Computing*, 2012.

[55] Chentao Wu, Xubin He, Qiang Cao, Changsheng Xie, and Shenggang Wan. Hint-k: An efficient multi-level cache using k-step hints. *IEEE Transactions on Parallel and Distributed Systems*, 99, 2013.

[56] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Enabling software management for multicore caches with a lightweight hardware support. In *Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, 2009.

[57] Dongyuan Zhan, Hong Jiang, and Sharad C. Seth. Locality & utility co-optimization for practical capacity management of shared last level caches. In *Proceedings of the 26th ACM international conference on Supercomputing*, 2012.

[58] Pilar Gonzalez-Ferez, Juan Piernas, and Toni Cortes. The ram enhanced disk cache project (redcap). In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, 2007.

[59] Sai Huang, Qingsong Wei, Jianxi Chen, Cheng Chen, and Dan Feng. Improving flash-based disk cache with lazy adaptive replacement. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, 2013.

[60] Zhichun Zhu and Xiaodong Zhang. Access-mode predictions for low-power cache design. *IEEE Micro*, 22(2), March 2002.

[61] Jianhui Yue, Yifeng Zhu, Zhao Cai, and Lin Lin. Energy and thermal aware buffer cache replacement algorithm. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[62] Adam Manzanares, Xiaojun Ruan, Shu Yin, Jiong Xie, Zhiyang Ding, Yun Tian, James Majors, and Xiao Qin. Energy efficient prefetching with buffer disks for cluster file systems. In *Proceedings of the 2010 39th International Conference on Parallel Processing*, 2010.

[63] Zhenhua Li, Christo Wilson, Zhefu Jiang, Yao Liu, Ben Zhao, Cheng Jin, Zhi-Li Zhang, and Yafei Dai. Efficient batched synchronization in dropbox-like

cloud storage services. In *Proceedings of the 14th International Middleware Conference*, 2013.

[64] Yaoqiang Xu, Chunxiao Xing, and Lizhu Zhou. A cache replacement algorithm in hierarchical storage of continuous media object. In *Advances in Web-Age Information Management: 5th International Conference*, 2004.

[65] Rui Li, Rui Guo, Zhenquan Xu, and Wei Feng. A prefetching model based on access popularity for geospatial data in a cluster-based caching system. *Int. J. Geogr. Inf. Sci.*, 26(10), October 2012.

[66] Dongfang Zhao, Jian Yin, Kan Qiao, and Ioan Raicu. Virtual chunks: On supporting random accesses to scientific data in compressible storage systems. In *Proceedings of IEEE International Conference on Big Data*, 2014.

[67] Dongfang Zhao, Jian Yin, and Ioan Raicu. Improving the i/o throughput for data-intensive scientific applications with efficient compression mechanisms. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13), poster session*, 2013.

[68] Chen Shou, Dongfang Zhao, Tanu Malik, and Ioan Raicu. Towards a provenance-aware distributed filesystem. In *5th Workshop on the Theory and Practice of Provenance (TaPP)*, 2013.

[69] Dongfang Zhao, Chen Shou, Tanu Malik, and Ioan Raicu. Distributed data provenance for large-scale data-intensive computing. In *Cluster Computing, IEEE International Conference on*, 2013.

[70] Ke Wang, Xiaobing Zhou, Tonglin Li, Dongfang Zhao, Michael Lang, and Ioan Raicu. Optimizing load balancing and data-locality with data-aware scheduling. In *Proceedings of IEEE International Conference on Big Data*, 2014.

[71] Dongfang Zhao, Kent Burlingame, Corentin Debains, Pedro Alvarez-Tabio, and Ioan Raicu. Towards high-performance and cost-effective distributed storage systems with information dispersal algorithms. In *Cluster Computing, IEEE International Conference on*, 2013.

[72] Iman Sadooghi, Jesus Hernandez Martin, Tonglin Li, Kevin Brandstatter, Ketan Maheshwari, Tiago Pais Pitta de Lacerda Ruivo, Gabriele Garzoglio, Steven Timm, Yong Zhao, and Ioan Raicu. Understanding the performance and potential of cloud computing for scientific applications. *IEEE Transaction on Cloud Computing (TCC)*, 2015.

[73] Dongfang Zhao and Li Yang. Incremental isometric embedding of high-dimensional data using connected neighborhood graphs. *IEEE Trans. Pattern Anal. Mach. Intell. (PAMI)*, 31(1), January 2009.

[74] Robin Lohfert, James Lu, and Dongfang Zhao. Solving sql constraints by incremental translation to sat. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, 2008.

[75] Dongfang Zhao and Li Yang. Incremental construction of neighborhood graphs for nonlinear dimensionality reduction. In *Proceedings of International Conference on Pattern Recognition*, 2006.

Dongfang Zhao is working towards his PhD in computer science at Illinois Institute of Technology (IIT) in Chicago, IL. His research interests include data-intensive computing, cloud computing, and big data. Before joining IIT, he was a full time software developer at Epic Systems, Madison, WI until 2011. He obtained his Master's degree in computer science from Emory University (Atlanta, GA), dual degrees in statistics and artificial intelligence from Katholieke Universiteit Leuven (Belgium), and B.E. in computer science and technology from Northeastern University (China).

Kan Qiao is working towards his PhD in computer science at Illinois Institute of Technology in Chicago, IL. He obtained his bachelor's degree in Automation Science from Beijing University Of Aeronautics and Astronautics (China) in 2010. His research interests include combinatorial optimization and network algorithms. He has published more than 10 peer-reviewed papers. He was a world finalist of the 36th ACM International Collegiate Programming Contest in Warsaw, Poland. He worked for Google, Amazon and Baidu for multiple software engineering internships.

Dr. Ioan Raicu is an assistant professor in the Department of Computer Science (CS) at Illinois Institute of Technology (IIT), as well as a guest research faculty in the Math and Computer Science Division (MCS) at Argonne National Laboratory (ANL). He has received the prestigious NSF CAREER award (2011 - 2015) for his innovative work on distributed file systems for extreme-scales. He was a NSF/CRA Computation Innovation Fellow at Northwestern University in 2009 - 2010, and obtained his Ph.D. in Computer Science from University of Chicago under the guidance of Dr. Ian Foster in March 2009. He is a 3-year award winner of the GSRP Fellowship from NASA Ames Research Center. His research work and interests are in the general area of distributed systems. Over the past decade, he has co-authored over 100 peer reviewed articles, book chapters, books, theses, and dissertations, which received over 5569 citations, with a H-index of 29. His work has been funded by the NASA Ames Research Center, DOE Office of Advanced Scientific Computing Research, the NSF/CRA CIFellows program, and the NSF CAREER program. He is a member of the IEEE and ACM.