HyCache+: Towards Scalable High-Performance Caching Middleware for Parallel File Systems

Dongfang Zhao*, Kan Qiao*, Ioan Raicu*†
*Illinois Institute of Technology, USA
†Argonne National Laboratory, USA
{dzhao8, kqiao}@iit.edu, iraicu@cs.iit.edu

***Abstract*—The ever-growing gap between the computation and I/O is one of the fundamental challenges for future computing systems. This computation-I/O gap is even larger for modern large scale high-performance systems due to their state-of-the-art yet decades long architecture: the compute and storage resources form two cliques that are interconnected with shared networking infrastructure. This paper presents a distributed storage middleware, called HyCache+, right on the compute nodes, which allows I/O to effectively leverage the high bi-section bandwidth of the high-speed interconnect of massively parallel high-end computing systems. HyCache+ provides the POSIX interface to end users with the memory-class I/O throughput and latency, and transparently swap the cached data with the existing slow-speed but high-capacity networked attached storage. HyCache+ has the potential to achieve both high performance and low-cost large capacity, the best of both worlds. To further improve the caching performance from the perspective of the global storage system, we propose a 2-phase mechanism to cache the hot data for parallel applications, called 2-Layer Scheduling (2LS), which minimizes the file size to be transferred between compute nodes and heuristically replaces files in the cache. We deploy HyCache+ on the IBM BlueGene/P supercomputer, and observe two orders of magnitude faster I/O throughput than the default GPFS parallel file system. Furthermore, the proposed heuristic caching approach shows 29X speedup over the traditional LRU algorithm.**

***Index Terms*—distributed caching, parallel and distributed file systems, heterogeneous storage**

I. Introduction

The ever-growing gap between the computation and I/O is one of the fundamental challenges for today's large scale computing systems. In 2005, fusion science [12] output simulation data at 3.5TB/s, while in 2013 the fastest storage system [4] can only deliver 1.4TB/s throughput. To make it worse, the number of compute cores follows Moore's Law, meaning that the output data would keep doubling every 18 months. On the other hand, the storage systems have been improved at a much slower pace in the last decades, and are likely to keep this pace for the next decade if no architectural change is made.

The gap between I/O and computation is even larger for modern high-performance computing (HPC) systems. As shown in our previous study [53], current state-of-the-art yet decades long storage architecture of HPC systems would unlikely provide the support for the expected level of concurrent data access for future systems. The main critique comes from the topological allocation of compute and storage resources that are interconnected as two cliques. Even though the network between compute and storage has high bandwidth and is sufficient for compute-intensive petascale applications, it would not be adequate for data-intensive petascale computing or the emerging exascale computing (regardless if it is compute- or data-intensive). Future storage systems need to be re-architected to co-locate storage and compute resources in order to better support the extreme level of concurrency expected in future computing systems. These future storage systems should leverage the higher bi-section bandwidth of modern torus interconnects, and the abundance of computational resources of the entire system (generally 2 to 3 orders of magnitude larger than the resources found in a dedicated segregated distributed storage system).

This paper presents a distributed storage middleware, called HyCache+, right on the compute nodes, which allows I/O to effectively leverage the high bi-section bandwidth of the high-speed interconnect of massively parallel high-end computing systems (see Figure 1). HyCache+ acts as the primary place for holding hot data for the applications (e.g. metadata, intermediate results for large scale data analysis), and only asynchronously swaps with cold data on the remote parallel file system with a unique design on distributed metadata and data-locality. Therefore, HyCache+ opens the door to providing both high performance and cost-effective large capacity.

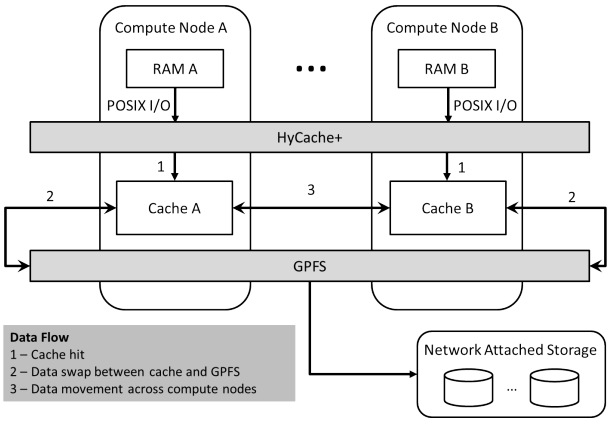

Figure 1. HyCache+ hierarchy

The name of HyCache+ originates from our previous work on a single-node hybrid caching system called HyCache [48]. HyCache+ significantly extends HyCache with the support of

networked storage and data reliability via replicas, and scales up to 4096 cores with a distributed hash table (DHT) for metadata management, as summarized in Table I.

Table I
SOME KEY HYCACHE+ IMPROVEMENTS OVER HYCACHE

Mechanism	HyCache+	HyCache
Network Storage	Yes	No
Data Movement	Local & Remote	Local Only
Replica	Arbitrary (e.g. 3)	1
Scalability	4096-cores	1-node
Metadata	DHT	Symbolic Link

To further improve the caching performance from the perspective of the global storage system, we propose a 2-phase mechanism, called 2-Layer Scheduling (2LS), to enhance the data locality of cached data enlightened by our previous work in the Falkon framework [34] on data diffusion [33]. The first layer schedules the jobs on (subsets of) available nodes in a manner of minimizing the total file size to be transferred across compute nodes. The second layer schedules the data locations across the local heterogeneous storage to heuristically maximize the total cached file size with consideration of both individual file size and its access frequency. Both layers make synergistic effort to achieve high global cache effectiveness.

We deploy HyCache+ on the IBM BlueGene/P supercomputer, and observed that the caching throughput is two orders of magnitude faster than the default parallel file system GPFS [35]. The proposed 2LS approach is evaluated and compared to the conventional Least Recently Used (LRU) replacement algorithm, showing up to 29X higher throughput.

In summary, this paper makes the following contributions:

1) ***Design and implement a scalable high-performance caching middleware, namely HyCache+, to improve the I/O performance of large scale distributed systems.***
2) ***Propose and analyze a novel caching approach —2-Layer Scheduling (2LS)— to optimize the network cost and heuristically reduce the disk I/O cost.***
3) ***Evaluate the HyCache+ storage system and the 2LS caching mechanism at large scale, and report their performance on a leadership class supercomputer.***

II. THE DESIGN AND IMPLEMENTATION OF HYCACHE+

Figure 2 shows the design overview of HyCache+, on an oversimplified 2-node cluster. A job scheduler deploys a job on a specific machine, *Node 1* in this example. To maximize the aggregate throughput, HyCache+ employs a unique approach on data movement: the write always occurs on a local node, and the read is located as close as possible to the application by some rules (elaborated in §III). If the file to be written happens to originate from a remote node, then the modified file will not be sent back to its original node. Rather, the metadata of the modified file is updated to reflect that its new primary location is the new node. This approach significantly reduces the network cost because updating metadata in most cases is cheaper than updating the (potentially much larger) file. The client (or application) is able to access the global namespace of the file system with a distributed metadata service. The hot files are accessed from the local cache if possible, and can potentially be replaced by the cold files in the remote parallel file systems according to the caching algorithm, e.g. LRU or Algorithm 2 that will be presented in §III. The hot files could be migrated between compute nodes with extremely high throughput and low latency, since most HPC systems are deployed with high-speed interconnect between compute nodes in order to meet the needs of large scale compute-intensive applications.

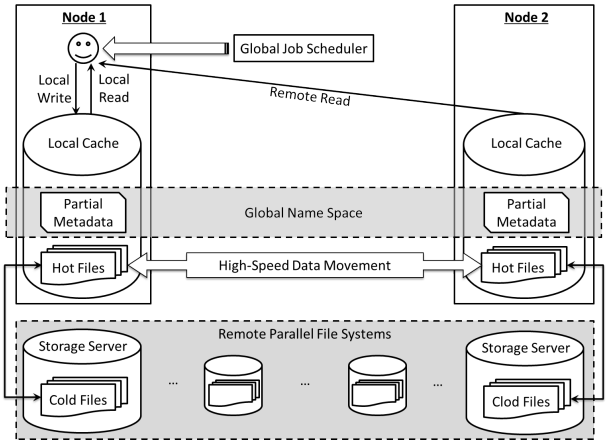

Figure 2. HyCache+ design overview

A. User Interface

One of our design goals is to provide complete transparency of the underlying storage heterogeneity to the users. By transparency, we mean that users are agnostic about which files are stored on which underlying storage types, or which physical nodes. This transparency is achieved by a global view of metadata of all the dispersed files.

In general, it is critical for a distributed/parallel file system to support POSIX for HPC applications, since POSIX is one of the most widely used standard. For legacy reasons, most HPC applications assume that the underlying file system supports POSIX. For the sake of backward compatibility, POSIX should be supported if at all possible. HyCache+ leverages the FUSE framework [1] to support POSIX. FUSE has been criticized for its efficiency on traditional HDD-based file systems. In native UNIX file systems (e.g. Ext4) there are only two context switches between the caller in user space and the system call in kernel spaces. However for a FUSE-based file system, context needs to be switched four times: two switches between the caller and the virtual file system; and another two between libfuse and FUSE. We will show that this overhead, at least in HPC systems when multi-threading is turned on, is insignificant (§IV-A).

HyCache+ is deployed as a user level mount point in accordance with other user level file systems. The mount point itself is a virtual entry point that talks to the local

cache and remote parallel file system. For example (see Figure 3), HyCache+ could be mounted on a local directory called */mnt/hycacheplus/*, while two other physical directories */dev/ssd/* and */mnt/gpfs/*) are for the local cache and the remote parallel file system, respectively.

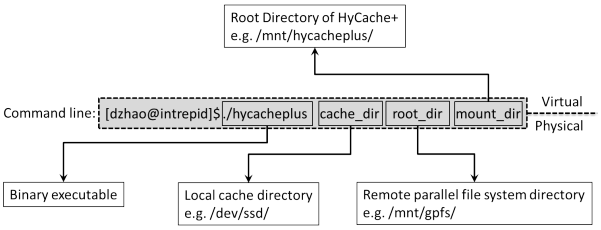

Figure 3. HyCache+ mountpoint

B. Network Protocols

We encapsulate a few different network protocols in an abstraction layer, called LibNap. Users only need to specify which network protocol to use in the deployment. Currently, LibNap supports four protocols: TCP, UDP, MPI, and UDT. UDT [15] is a user level protocol with high reliability built upon UDP.

To deal with the high concurrency on metadata servers, epoll is used instead of multithreading. The side effect of epoll is that the received message packets are not kept in the same order as on the sender. To address this, a header [message_id, packet_id] is added to the message at the sender, and the message is restored by sorting the packet_id for each message at the recipient. This is efficiently done by a sorted map with message_id as the key, mapping to a sorted set of the message's packets. The server also periodically triggers a garbage collection for the orphan packets due to crashed clients, unreliable network, and so on. Note that the above techniques work together with the network protocols such as TCP to for example deal with the lost packets between compute nodes.

C. Metadata Management

The global single namespace comprises partial metadata views on local nodes. Nevertheless, client could interact with the DHT for any key-value pair no matter if it is on the local storage or on some remote nodes. In some sense, DHT is the translator between local partial metadata and the global namespace. The global namespace does not need to be aggregated or flushed when local changes occur. Any changes in the local metadata storage are immediately visible to the global namespace without extra processing. It is an analogy that modifying a sub graph will automatically update the topology of the entire graph.

We implement the distributed metadata management by modifying our previous work on ZHT [19]. By selecting an appropriate hash function, the key-value pairs will be evenly distributed on the available nodes, which achieves load balance automatically.

We manage the tree-like directory hierarchy of HyCache+ with a (directional) adjacency list. The adjacency list data structure could then be implemented in a general key-value pair in the distributed hash table. To make matters more concrete, Figure 4 shows a segment of a DHT example. It should be noted this 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 (3 directories, 2 regular files) are stored in the DHT, with their file names as keys. The value is comprised of a list of properties delimited by semicolons. The first and second portions of the values are for permission flags and the file size, respectively. The third portion for a directory is a list of its children delimited by commas, while for regular files is the physical location of the file.

Key	Value
~/	drwxrwxr-x; 4.0K; ~/homedir/subdir
~/homedir/	drwxrwxr-x; 4.0K; ~/homedir/subdir, ~/homedir/homefile
~/homedir/subdir/	drwxrwxr-x; 4.0K; ~/homedir/subdir/subfile
~/homedir/homefile	-rw-rw-r--; 423M; Node 1
~/homedir/subdir/subfile	-rw-rw-r--; 133M; Node 2
⋮	⋮

Figure 4. HyCache+ metadata implementation with DHT

This example is only to illustrate how the DHT behaves in an oversimplified setup, and we should mention the following clarifications. (1) In real systems there is additional metadata information stored in the values, such as modification time, owner ID, etc, that are commonly seen in i-nodes. We do not list all of them here for the sake of limited space. (2) What is shown in the value is a simple string delimited by semicolons. This is only for clear presentation to explain what types of information is stored. In implementation, the value is in fact a serialization of the data structure for metadata. The structured metadata is serialized by Google Protocol Buffer [2] before being sent over the network to the metadata server, which is just a particular compute node and acts like a logical server to receive the metadata. Similarly, when the metadata is retrieved, we deserialize the blob back into the structure. All the regular i-node information is tracked, plus the list of children for a directory or the node location for a regular file.

DHT has strict consistency semantics, as only the primary copy is used for read and write operation, and replicas are only used to avoid data loss in case of node failures. We also use atomic append operations (supported by ZHT [19]) to implement lock-free concurrent directory modification, where only changes to a directory entries are transmitted (instead of the entire contents of the directory); this allows the creation or removal of files or directories in constant time with minimal network communication even in cases where the directories contain many entries.

D. Data Movement

Our strategy to achieve high and scalable write throughput is straightforward: a client only writes data to its local storage.

In other words, it is independent write across data nodes, in the sense that no interference exists on the layer of physical data servers. This local-only write is not really independent, in the sense that all these writes are under control of the coherent metadata on the same namespace.

The aggregate write throughput is obviously optimal: all writes are associated with local I/O throughput and avoids the following two potential overheads commonly seen in other systems: (1) the procedure to determine to which node the data will be written, normally accomplished by pinging the metadata nodes and/or some monitoring services, and (2) transferring the data to a remote node.

The potential issue with local writes is equally obvious: no guarantee for load balance is provided at all. The assumption that all workloads are uniformly deployed on all nodes is too strong, as we have observed hot spots in large scale systems. As a consequence, a periodical re-balance is invoked.

In some cases, the client is writing to a file that is originally stored in another node. Per our policy, the newly written file will not be sent back to the original node. Rather, the metadata of this file will be updated in the metadata. This saves the cost of transferring the file data over the network by a much faster operation on updating the metadata. The question arises though: what if two clients try to write to the same data? The answer is that a distributed lock service (coordination) is available built on top of the atomic compare-swap operation on the underlying distributed hash table.

Unlike data write, it is impossible to arbitrarily control where the requested data reside. The location of the requested data is highly dependent on the I/O pattern, and the probability of the requested data residing on the local storage is extremely low when assuming the I/O has a uniform distribution (i.e. $P = \frac{1}{n}$, where n denotes the number of nodes). In such cases, we could transfer the requested data from the remote node to the requesting node with some protocols that will discussed in §II-B.

For the default LRU replacement policy, we implement a priority queue to keep track of the hot files in the cache. Each element of the queue maintains key information such as file name, file size, and so on. The queue is updated in accordance with the cache content, which might need to import cold files or evict hot files. Details on the proposed 2LS mechanism will be discussed in §III.

E. Fault Tolerance

Failures of today's large scale systems are normality rather than exception. The conventional wisdom is to make a number of replicas to the primary copy [13, 17]. When the primary copy is failed, one of the replicas will be restored to replace the failed primary copy. This method has its advantages such as ease-of-use and low computational overhead, when compared to the emerging erasure-coding mechanisms [26] and its variants e.g. [25]. The main critique on replicas is, however, its low storage efficiency. For example, in Google file system [13] each primary copy has two replicas, which results in the storage utilization as $\frac{1}{1+2} \approx 33\%$.

We have investigated three semantics on data redundancy: synchronous replicas, asynchronous replicas, and erasure coding. In theory, asynchronous replicas would deliver the highest throughput, while being compromised on the possibility of failing to recover before the asynchronous update is completed. Synchronous update is a costly method, and satisfies the strong consistency requirement, if needed. Erasure coding trades off between performance and consistency, in that it needs to transfer less data than synchronous update, and takes more time than asynchronous update to guarantee the fault tolerance. In practice, the choice between synchronous and asynchronous replicas is highly dependent on the application requirements, while erasure coding is more related to the computing hardware and the chosen encoding algorithms. Our previous study [46] shows that a commodity GPU makes erasure coding outperform traditional replicas by 1.82X speedup.

III. 2-Layer Scheduling of Distributed Cache

A. Job Scheduling

The variables to be used in the discussion are summarized in Table II. If the application needs to access a file F_i on a remote machine, the overhead on transferring F_i is $Size(F_i)$.

Table II
VARIABLES OF GLOBAL SCHEDULING

Variable	Type	Meaning
M	Set	Machines of the cluster
A	Set	Applications to be run
F	Set	All files
F^k	Set	Files referenced by $A_k \in A$
$P_{i,j}$	Int	$F_i \in F$ placed on $M_j \in M$
$Q_{i,j}$	Int	$A_i \in A$ scheduled on $M_j \in M$

We formalize the problem as to find the matrix Q (i.e. scheduling which job on which machine) that minimizes the overall network cost of running all $|A|$ jobs on $|M|$ machines. That is to solve the objective function

$$\arg\min_{Q} \sum_{A_k \in A} \sum_{M_l \in M} \sum_{F_i \in F^k} \sum_{M_j \in M} Size(F_i) \cdot P_{i,j} \cdot Q_{k,l},$$

subject to

$$\sum_{M_j \in M} P_{i,j} = 1, \forall F_i \in F,$$

$$\sum_{M_j \in M} Q_{i,j} = 1, \forall A_i \in A,$$

$$P_{i,j}, Q_{i,j} \in \{0, 1\}, \forall i, j.$$

The first constraint guarantees that a file could be placed on exact one node. Similarly, the second constraints guarantees that a job could be scheduled on exact one node. Note that both constraints could be generalized by replacing 1 with other constants if needed, for example in distributed file systems [38] a file could have multiple replicas for high reliability. The last constraint says that both matrices should only store binary values to guarantee the first and the second constraints.

The algorithm to find the machine for a job to achieve the minimal network cost is given in Algorithm 1. The input is the job index x, and it returns the machine index y. It loops on each machine (Line 4), calculates the cost of moving all the referenced files to this machine (Lines 5 - 9), and updates the minimal cost if needed (Lines 10 - 13).

Algorithm 1 Global Schedule

Input: The x^{th} job to be scheduled
Output: The y^{th} machine where the x^{th} job should be scheduled

```
1:  function GLOBALSCHEDULE(x)
2:      MinCost ← ∞
3:      y ← NULL
4:      for M_i ∈ M do
5:          Cost ← 0
6:          for F_j ∈ F^x do
7:              Find M_k such that P_{j,k} = 1
8:              Cost ← Cost + Size(F_j)
9:          end for
10:         if Cost < MinCost then
11:             MinCost ← Cost
12:             y ← i
13:         end if
14:     end for
15:     return y
16: end function
```

The correctness of Algorithm 1 is due to the fact that the data locality is known as a priori input. That is, P is a given argument to the execution of all jobs. Otherwise Line 7 would not work appropriately. The per-job networking overhead is obviously minimal. Since jobs are assumed independent, the overall overhead of all jobs is also minimal.

The complexity of Algorithm 1 is $O(|M| \cdot |F_x|)$, by observing the two loops on Line 4 and Line 6, respectively. Note that we could achieve an $O(1)$ cost for Line 7 by retrieving the metadata of file j, as discussed in §II-C.

B. Heuristic Caching

Problem Statement. The problem of finding optimal caching on multiple-disk is proved to be NP-hard [6]. A simpler problem on a single-disk setup has a polynomial solution [5], which is, unfortunately, too complex to be applied in real applications. An approximation algorithm was proposed in [10] 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) [7] with exponential time complexity. However, in real applications the number of accessed files could be as large as 10,000, which makes the dynamic programming approach feasible. 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 arbitrarily large number of files in practice, especially when compared to the I/O time of the disk.

Assumptions. We assume a queue of jobs, and their requested files are known on each node in a given period of time, which could be calculated from the scheduling results of §III-A and the metadata information in §II-C. This assumption is based on our observation of many workflow systems [54, 55], 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 this node, because there is no need to cache the files that will be accessed by the jobs deployed on remote nodes.

Notations and Definitions 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 throughput is typically orders of magnitude higher than disks (i.e. O(10GB/s) vs. O(100MB/s)), 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.

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

Rule 1. Every fetch should bring into the cache the very *next* file in the reference sequence if it is not yet in the cache.

Rule 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.

The first 2 rules specify which file to be fetched and when to do the fetch, and say nothing about evicting files. Rule 3 speaks about what files to be evicted and when to do the eviction.

Rule 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 $Cost(F_3) = 18$.

The caching procedure is presented in Algorithm 2, 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 2, $File(r_{i+1})$ 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 6).

Algorithm 2 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}) ∧ j > i+1} ≠ ∅ then
3:          flag, D ← 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 is as follows. Line 2 only 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 3. Thus the overall time complexity of Algorithm 2 is $O(n \lg n)$.

The $GetFilesToDiscard()$ subroutine (Algorithm 3) 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).

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

We will show that the time complexity of Algorithm 3 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 $O(n \lg n)$.

IV. Evaluation

Most experiments are carried out on *Intrepid* [3], an IBM BlueGene/P supercomputer of 160K cores at Argonne National Laboratory. We use up to 1024 nodes (4096 cores) in the evaluation. Each node has a 4-core PowerPC 450 processor (850MHz) and 2GB of RAM. A 7.6PB GPFS [35] is deployed on 128 storage nodes. All experiments are repeated at least five times, or until results become stable (i.e. within 5% margin of error); the reported numbers are the average of all runs. Caching effect is carefully precluded by reading a file larger than the on-board memory before the measurement.

A. FUSE Overhead

We show that FUSE [1] overhead in the context of a HPC system, particularly with C/C++ bindings on memory-class storage, could be greatly compensated by the high concurrency supported by the storage. Our experiment shows that a FUSE+SSD file system could achieve 580 MB/sec aggregate bandwidth (at 12 concurrent processes, the same number of hardware threads of the test bed AMD Phenom II X6 1100T Processor) for concurrent data accesses, which is about 85% of the bandwidth of the raw SSD device (i.e. SSD Ext4).

B. I/O Throughput

We illustrate how HyCache+ significantly improves the I/O throughput of parallel file systems. The local cache size is set to 256MB (0.25GB) on each node. To measure local cache's stable throughput, each client repeatedly writes a 256MB file for 63 times (total 15.75GB). Then another 256MB file is written on each client to trigger the swapping between local cache and GPFS. Figure 5 reports (at 256-core scale) the real-time aggregate throughput, showing a significant performance drop at around 90-second timestamp, when the 15.75GB data are finished on the local cache.

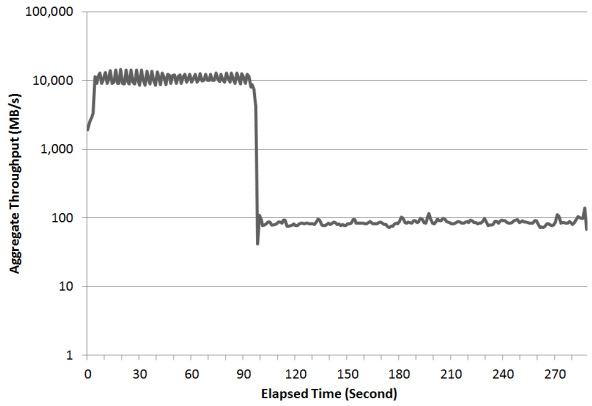

Figure 5. Throughput on BlueGene/P (256-cores)

C. Scalability

We demonstrate the scalability of HyCache+ by repeating the experiment of the same per-client workload in §IV-B on 512 nodes (2048 cores). The real-time throughput is reported in Figure 6. We see that HyCache+ shows an excellent scalability for both the cached data and the remote data: both the caching throughput and the disk throughput are about 8X faster than those numbers at 256-core scale in Figure 5.

D. Fault Tolerance

We are interested in the overhead introduced by different fault-tolerance semantics: strong consistency (i.e. synchronous replication) and weak consistency (i.e. asynchronous replication). To measure that, HyCache+ was deployed on 128 compute nodes, each of which writes independent files with 1MB block size, all within the cache. The baseline is the case when no replication is enabled, which is considered as

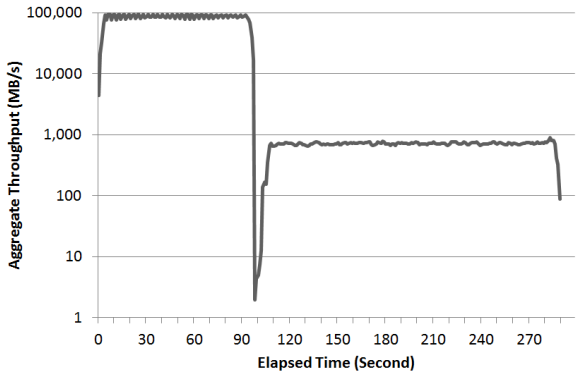

Figure 6. Throughput on BlueGene/P (2048-cores)

the upper bound of any replication strategies. Asynchronous replication achieves a high efficiency (90%) comparing to the no-replication strategy. To achieve strong consistency, synchronous replication incurs a high overhead, delivering 68% out of the no-replication throughput.

E. Scheduling Algorithms

We plug the heuristic caching and LRU algorithms into HyCache+, and simulate their performance at 512-node scale on Intrepid. We create different sizes of data, 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 Figure 7. 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).

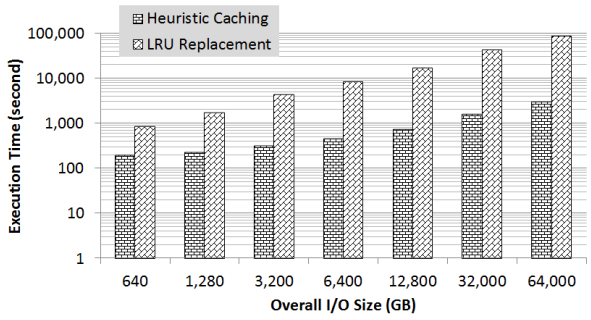

Figure 7. Comparison between Heuristic Caching and LRU

F. Broader Impact

Even though this work focuses on HyCache+ for parallel file systems, it also enlightens the design of future distributed file systems where each compute node is deployed with a local storage. We report some promising preliminary results of HyCache+ on a 64-node Linux cluster at Illinois Institute

of Technology. Each node has two Quad-Core AMD Opteron 2.3GHz processors and one 1TB Seagate Barracuda hard drive. All nodes are interconnected with 1Gbps Ethernet. The cache path is set to the local RAM disk (i.e. /dev/shm), and the local hard disk is considered as the media to hold the chunks of distributed file systems. Figure 8 shows the aggregated throughput with and without HyCache+. HyCache+ helps delivers about 2.2X improved I/O throughput.

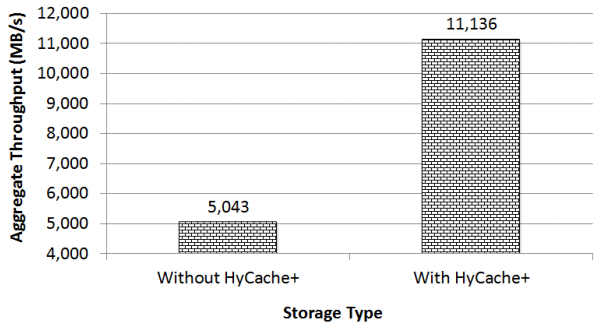

Figure 8. Aggregate throughput on a 64-nodes cluster

V. Related Work

While both our previous work [34] and [33] focused on the job scheduling in order to improve applications' performance, this paper achieves the same goal from the storage's perspective. That is, the previous work was a top-down mechanism to manipulate jobs without much knowledge of the underlying storage, but this work shows a bottom-up approach to allow users to take advantage on the storage's awareness of data locality by providing the convenient POSIX interface. Moreover, in [33] we discussed different strategies broadly to showcase how to achieve different criteria such as data locality, load balance, or a mix of both, while this paper concentrates on detailing the scheduling and caching algorithms to (heuristically) minimize the overhead of the distributed storage.

A thorough review of classical caching algorithms on large scale data-intensive applications is recently reported in [11]. HyCache+ is different from the classical cooperative caching [27] in that HyCache+ 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) [36] is a read-only caching system for cluster file systems. In contrast, HyCache+ is a POSIX-compliant I/O storage middleware that transparently interacts with the underlying parallel file systems. Even though the focus of this paper lies on the 2-layer hierarchy of a local cache (e.g. SSD) and a remote parallel file system (e.g. GPFS [35]), the approach presented in HyCache+ 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 [41] 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.

There are extensive studies on leveraging data locality for effective caching. Block Locality Caching (BLC) [24] 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) [42] optimizes the large scale file system by correlating access patterns and semantic attributes. In contrast, HyCache+ achieves data locality with a unique mix of two principles: (1) write is always local, and (2) read locality depends on the novel 2LS (§III) mechanism which schedules jobs in a deterministic manner followed by a local heuristic replacement policy.

While HyCache+ presents a pure software solution for distributed cache, some orthogonal work focuses on improving caching from the hardware perspective. In [21], a hardware design is proposed with low overhead to support effective shared caches in multicore processors. For shared last-level caches, COOP [45] is proposed to only use one bit per cache line for re-reference prediction and optimize both locality and utilization. The REDCAP project [14] aims to logically enlarge the disk cache by using a small portion of main memory, so that the read time could be reduced. For Solid-State Drive (SSD), a new algorithm called lazy adaptive replacement cache [16] 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 HyCache+ aims to better meet the need of high I/O performance for HPC systems, and power consumption is not the major consideration 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 [56] 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 [44] was proposed to save up to 27% energy and reduce the memory temperature up to 5.45°C with negligible performance degradation. EEVFS [23] provides energy efficiency at the file system level with an energy-aware data layout and the prediction on disk idleness.

While HyCache+ is architected for large scale HPC systems, caching has been extensively studied in different subjects and fields. In cloud storage, Update-batched Delayed Synchronization (UDS) [20] 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) [43] is proposed to optimize the newly defined metric called user waiting rate. In geoinformatics, the method proposed in [18] considers both global and local temporal-spatial changes to achieve high cache hit rate and short response time.

The job scheduler proposed in this work (§III-A) takes a greedy strategy to achieve the optimal solution for the

HyCache+ architecture. A more general, and more difficult, scheduling problem could be solved in a similar heuristic approach [28, 39]. For an even more general combinatorial optimization problem in a network, both precise and bound-proved low-degree polynomial approximation algorithms were reported in [8, 9]. Some incremental approaches [22, 50, 51] were proposed to efficiently retain the strong connectivity of a network and solve the satisfiability problem with constraints.

VI. Conclusion

This paper presents the design and implementation of HyCache+, a scalable high-performance caching middleware to improve the I/O performance of parallel file systems. It proposes and analyzes a novel 2-layer approach to minimize the network cost and heuristically optimize the caching effect. Large scale evaluation at up to 4096 cores shows that HyCache+ improves the I/O performance by up to two orders of magnitude, and the proposed caching approach could further elevate the performance by 29X.

As our future work, we plan to deploy HyCache+ on more parallel and distributed file systems and integrate the 2LS approach to our previous work on scheduling and launching workloads in Many-Task Computing (MTC) [29, 31, 32]. We believe HyCache+, together with other features such as data compression [52] and data provenance [37, 49], would make the next generation extreme-scale storage system (e.g. [47]) more practical for real applications [30]. We aim to have an entirely complete software stack from programming languages (e.g. Swift [54] and other scripts [40]), runtime systems (e.g. Falkon [34]), and storage systems (e.g. ZHT [19]) to address the avalanche of challenges brought on by Big Data applications.

Acknowledgement

This work was supported in part by the National Science Foundation under awards OCI-1054974 (CAREER). This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. The authors are grateful to Gruia Calinescu for the insightful discussion, and to Xian-He Sun for providing the access to the test bed in §IV-F. The authors are also thankful to the anonymous reviewers' comments to improve this paper.

References

[1] FUSE Project. http://fuse.sourceforge.net.

[2] Google protocol buffers. http://code.google.com/p/protobuf/.

[3] Intrepid. https://www.alcf.anl.gov/intrepid.

[4] Titan. http://phys.org/news/2013-04-supercomputer-titan-world-fastest-storage.html.

[5] S. Albers, N. Garg, and S. Leonardi. Minimizing stall time in single and parallel disk systems. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 454–462, 1998.

[6] C. Ambühl and B. Weber. Parallel prefetching and caching is hard. In *STACS*, pages 211–221, 2004.

[7] R. Bellman. Dynamic programming treatment of the travelling salesman problem. *J. ACM*, 9(1):61–63, Jan. 1962.

[8] G. Calinescu, S. Kapoor, K. Qiao, and J. Shin. Stochastic strategic routing reduces attack effects. In *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, pages 1–5, Dec 2011.

[9] G. Calinescu and K. Qiao. Asymmetric topology control: Exact solutions and fast approximations. In *IEEE International Conference on Computer Communications (INFOCOM '12)*, pages 783–791, March 2012.

[10] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. *SIGMETRICS Perform. Eval. Rev.*, 23(1):188–197, May 1995.

[11] R. Fares, B. Romoser, Z. Zong, M. Nijim, and X. 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*, pages 227–234, 2012.

[12] P. A. Freeman, D. L. Crawford, S. Kim, and J. L. Munoz. Cyberinfrastructure for science and engineering: Promises and challenges. *Proceedings of the IEEE*, 93(3):682–691, 2005.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, 2003.

[14] P. Gonzalez-Ferez, J. Piernas, and T. Cortes. The ram enhanced disk cache project (redcap). In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, MSST '07, pages 251–256, 2007.

[15] Y. Gu and R. L. Grossman. Udt: Udp-based data transfer for high-speed wide area networks. *Comput. Netw.*, 51(7):1777–1799, May 2007.

[16] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving flash-based disk cache with lazy adaptive replacement. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–10, 2013.

[17] H. Jin, K. Qiao, X.-H. Sun, and Y. Li. Performance under failures of mapreduce applications. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 608–609, 2011.

[18] R. Li, R. Guo, Z. Xu, and W. Feng. A prefetching model based on access popularity for geospatial data in a cluster-based caching system. *Int. J. Geogr. Inf. Sci.*, 26(10):1831–1844, Oct. 2012.

[19] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu. Zht: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 775–787, 2013.

[20] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Zhao, C. Jin, Z.-L. Zhang, and Y. Dai. Efficient batched synchronization in dropbox-like cloud storage services. In *Proceedings of the 14th International Middleware Conference*, Middleware '13, Beijing, China, 2013.

[21] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. 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*, SC '09, pages 14:1–14:12, 2009.

[22] R. Lohfert, J. Lu, and D. Zhao. Solving sql constraints by incremental translation to sat. In *21st International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, 2008.

[23] A. Manzanares, X. Ruan, S. Yin, J. Xie, Z. Ding, Y. Tian, J. Majors, and X. Qin. Energy efficient prefetching with buffer disks for cluster file systems. In *Proceedings of the 2010 39th International Conference on Parallel Processing*, ICPP '10, pages 404–413, 2010.

[24] D. Meister, J. Kaiser, and A. Brinkmann. Block locality caching for data deduplication. In *Proceedings of the 6th International*

Systems and Storage Conference, SYSTOR '13, pages 15:1–15:12, 2013.

[25] J. S. Plank, M. Blaum, and J. L. Hafner. Sd codes: Erasure codes designed for how storage systems really fail. In *Proceedings of the 11th USENIX conference on File and Storage Technologies*, FAST '13, 2013.

[26] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *Proccedings of the 7th conference on File and storage technologies*, FAST '09, pages 253–265, 2009.

[27] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, Dec. 2003.

[28] K. Qiao, F. Tao, L. Zhang, and Z. Li. A ga maintained by binary heap and transitive reduction for addressing psp. In *Intelligent Computing and Integrated Systems (ICISS), 2010 International Conference on*, pages 12–15, Oct 2010.

[29] I. Raicu. *Many-task computing: bridging the gap between high-throughput computing and high-performance computing*. PhD thesis, Chicago, IL, USA, 2009.

[30] I. Raicu, I. Foster, A. Szalay, and G. Turcu. Astroportal: A science gateway for large-scale astronomy data analysis. In *TeraGrid Conference*, 2006.

[31] I. Raicu, I. Foster, M. Wilde, Z. Zhang, K. Iskra, P. Beckman, Y. Zhao, A. Szalay, A. Choudhary, P. Little, C. Moretti, A. Chaudhary, and D. Thain. Middleware support for many-task computing. *Cluster Computing*, 13(3):291–314, Sept. 2010.

[32] I. Raicu, I. Foster, Y. Zhao, A. Szalay, P. Little, C. M. Moretti, A. Chaudhary, and D. Thain. *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.

[33] I. Raicu, I. T. Foster, Y. Zhao, P. Little, C. M. Moretti, A. Chaudhary, and D. Thain. 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, 2009.

[34] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a fast and light-weight task execution framework. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 43:1–43:12, 2007.

[35] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, 2002.

[36] L. Shi, Z. Liu, and L. Xu. Bwcc: A fs-cache based cooperative caching system for network storage system. In *Proceedings of the 2012 IEEE International Conference on Cluster Computing*, CLUSTER '12, pages 546–550, 2012.

[37] C. Shou, D. Zhao, T. Malik, and I. Raicu. Towards a provenance-aware distributed filesystem. In *5th USENIX Workshop on the Theory and Practice of Provenance (TaPP '13)*, 2013.

[38] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, 2010.

[39] F. Tao, K. Qiao, L. Zhang, Z. Li, and A. Nee. GA-BHTR: an improved genetic algorithm for partner selection in virtual manufacturing. *International Journal of Production Research*, 50(8):2079–2100, 2012.

[40] M. Wilde, I. Raicu, A. Espinosa, Z. Zhang, B. Clifford, M. Hategan, K. Iskra, P. Beckman, and I. Foster. Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers. In *Scientific Discovery through Advanced Computing Conference*, SciDAC '09, 2009.

[41] C. Wu, X. He, Q. Cao, C. Xie, and S. Wan. Hint-k: An efficient multi-level cache using k-step hints. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints):1, 2013.

[42] P. Xia, D. Feng, H. Jiang, L. Tian, and F. 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*, HPDC '08, pages 185–196, 2008.

[43] Y. Xu, C. Xing, and L. Zhou. A cache replacement algorithm in hierarchical storage of continuous media object. In *Advances in Web-Age Information Management: 5th International Conference*, pages 157–166, 2004.

[44] J. Yue, Y. Zhu, Z. Cai, and L. Lin. Energy and thermal aware buffer cache replacement algorithm. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, 2010.

[45] D. Zhan, H. Jiang, and S. 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*, ICS '12, pages 279–290, 2012.

[46] D. Zhao, K. Burlingame, C. Debains, P. Alvarez-Tabio, and I. Raicu. Towards high-performance and cost-effective distributed storage systems with information dispersal algorithms. In *Proceedings of the 2013 IEEE International Conference on Cluster Computing*, CLUSTER '13, 2013.

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

[48] D. Zhao and I. 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*, IPDPSW '13, pages 1997–2006, 2013.

[49] D. Zhao, C. Shou, T. Malik, and I. Raicu. Distributed data provenance for large-scale data-intensive computing. In *Proceedings of the 2013 IEEE International Conference on Cluster Computing*, CLUSTER '13, 2013.

[50] D. Zhao and L. Yang. Incremental construction of neighborhood graphs for nonlinear dimensionality reduction. In *Proceedings of the 18th International Conference on Pattern Recognition - Volume 03*, ICPR '06, pages 177–180, 2006.

[51] D. Zhao and L. Yang. Incremental isometric embedding of high-dimensional data using connected neighborhood graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 31(1):86–98, Jan. 2009.

[52] D. Zhao, J. Yin, and I. 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)*, 2013.

[53] D. Zhao, D. Zhang, K. Wang, and I. Raicu. Exploring reliability of exascale systems through simulations. In *Proceedings of the High Performance Computing Symposium*, HPC '13, pages 1:1–1:9, 2013.

[54] Y. 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, 2007 IEEE Congress on*, pages 199–206, 2007.

[55] Y. Zhao, I. Raicu, S. Lu, and X. Fei. Opportunities and challenges in running scientific workflows on the cloud. In *IEEE International Conference on Network-based Distributed Computing and Knowledge Discovery*, CyberC '11, 2011.

[56] Z. Zhu and X. Zhang. Access-mode predictions for low-power cache design. *IEEE Micro*, 22(2):58–71, Mar. 2002.