J Supercomput (2019) 75:554–586
https://doi.org/10.1007/s11227-017-2167-7

WatCache: a workload-aware temporary cache on the compute side of HPC systems

Jie Yu[1] · Guangming Liu[2] · Wenrui Dong[1] · Xiaoyong Li[3]

Published online: 26 October 2017

Abstract As the computing power of high-performance computing (HPC) systems is developing to exascale, the storage systems are stretched to their limits to process the growing I/O traffic. Researchers are building storage systems on top of compute node-local fast storage devices (such as NVMe SSD) to alleviate the I/O bottleneck. However, user jobs have varying requirements of I/O bandwidth; therefore, it is a serious waste of expensive storage devices to have them on all compute nodes and build them into a global storage system. In addition, current node-local storage systems need to cope with the challenging small I/O and rank 0 I/O pattern from HPC workloads. In this paper, we presented a workload-aware temporary cache (WatCache) to meet above challenges. We designed a workload-aware node allocation method to allocate fast storage devices to jobs according to their I/O requirements and merged the devices of the jobs into separate temporary cache spaces. We implemented a metadata caching

This work was supported by the National Natural Science Foundation of China (Grant No. 61502511).

✉ Xiaoyong Li
sayingxmu@nudt.edu.cn

Jie Yu
yujie@nudt.edu.cn

Guangming Liu
liugm@nscc-tj.gov.cn

Wenrui Dong
dongwr@nscc-tj.gov.cn

1 College of Computer, National University of Defense Technology, Changsha, China

2 National Supercomputer Centre in Tianjin, Tianjin, China

3 Academy of Ocean Science and Engineering, National University of Defense Technology, Changsha, China

strategy that reduces the metadata overhead of I/O requests to improve the performance of small I/O. We designed a data layout strategy that distributes consecutive data that exceeds a threshold to multiple devices to achieve higher aggregate bandwidth for rank 0 I/O. Through extensive tests with several I/O benchmarks and applications, we have validated that WatCache offers linearly scalable performance, and brings significant performance promotions to small I/O and rank 0 I/O patterns.

Keywords High-performance parallel I/O · Burst buffer · Client-side temporary cache · Nonvolatile memory

1 Introduction

The sustained improvement of the computing capability of high-performance computing (HPC) systems has enabled scientific applications to perform more complex analysis and process larger datasets. For example, data-intensive analysis needs ingest massive amount of raw data to obtain insightful knowledge, and petascale-class simulations would constantly generate terabyte-sized data during each checkpoint [22]. These data-intensive applications require substantial I/O bandwidth to access the exponentially growing amount of data. However, most storage systems in HPC systems are built on conventional hard disk drives (HDDs) with several long-existing performance limitations, such as low random access performance and poor throughput under severe contention. Underlying storage systems are stretched to their limits to meet the overwhelming I/O demands of data-intensive applications. The increasing gap between computing and I/O capability has become a major performance drawback of HPC systems [48].

Nonvolatile memory (NVM) devices, particularly flash-based solid state drive (SSD), have drawn much attention in the community over the past few years owing to its high-performance and low power consumption [47]. Burst buffer [26] takes a step to integrate these fast storage devices into current HPC I/O stack. It has been employed on the I/O forwarding nodes of several leadership-class HPC systems [6,46,55] and has been proved to be a high-performance as well as cost-effective solution to buffer massive I/O traffic from compute nodes (CN) [33,39]. In order to further push burst buffer to a larger scale, researchers are trying to deploy burst buffer nearer to the I/O processes by attaching fast storage to the CNs. This approach arouses much interest in both academy and industry, and has been included in the blueprints of several next-generation HPC systems [1,8,43,44]. Nevertheless, there is no consensus on how to effectively manage these node-local fast storage devices for scientific applications.

There are tens of thousands of CNs in current leadership-class HPC machines [45]. Even considering the price-drop trend of NVM devices, there are still budget pressures to attach them to all the CNs. A more cost-effective approach is equipping a subset of CNs with fast storage. But then the limited fast storage devices are not enough to be allocated to all the applications simultaneously running on HPC machines. Fortunately, prior research [29] found that most of the I/O bandwidth of storage systems are consumed by only a small minority of applications. In addition, it is a serious waste of expensive storage resources to indiscriminately allocate the CNs with fast storage (burst nodes) to all applications regardless of their I/O demands. Therefore, proper strategy needs to be applied to assign the limited number of fast storage to applications that need them most.

There are many efforts that merge free memory or node-local fast storage to an additional storage tier between CNs and storage system [7,13,14,23,24,52]. However, HPC I/O has some unique characteristics that should be taken into full consideration. Firstly, rank 0 I/O is a common pattern that one process is in charge of all the I/O of application [22]. If store all data in local storage, the performance of accessing large amount of data in a single storage device is very poor compared to parallel file systems. On the other hand, if simply stripe data blocks across multiple clients, it loses the performance benefits of co-locating data with processes. Secondly, small I/O has become an I/O pattern that cannot be overlooked in HPC workloads [18]. However, the network overheads of querying metadata or retrieving data in current node-local storage systems still are the main constraint on small I/O performance. The network overheads need to be minimized to enhance small I/O performance.

In this paper, we present a workload-aware temporary cache (WatCache) to meet above design requirements. In order to ensure the effective use of expensive fast storage devices, WatCache only requires a part of the CNs in the HPC systems to be attached with local storage. We design a workload-aware node allocation method to allocate the appropriate numbers of fast storage devices to jobs according to their different I/O demands. WatCache merges the devices of the job into a job-wide global cache space, which is visible to all processes within this job. WatCache is established when a job starts running on its allocated CNs and is destroyed when the job ends. We further implement several techniques to enable WatCache to efficiently handle a variety of I/O patterns, including a data layout strategy that supports both rank 0 and all-rank I/O, and a metadata caching strategy which is designed to cope with large amount of small I/O in scientific applications. We use several I/O kernels and benchmarks to evaluate the performance of WatCache. The main contributions of this paper include following aspects.

- We design and implement a temporary cache system to manage the node-local burst buffers. It provides separate cache pools for different jobs and isolates the interference among them.
- We propose a workload-aware node allocation method to support cost-effective deployment of node-local burst buffers. It ensures that the limited fast storage devices are allocated to the applications that require the most I/O bandwidth.
- We implement several mechanisms to further promote WatCache performance to meet the requirements of different I/O patterns, including a data layout strategy to support both rank 0 and all-rank I/O, and a metadata caching mechanism to promote the performance of small I/O.
- We evaluate WatCache with several I/O kernels, benchmarks and realistic applications, which shows that WatCache is capable of bringing benefits to various I/O patterns as well as delivering linearly scalable performance.

2 Related work

There is a rapidly expanding amount of works on exploring how to integrate NVM devices to current HPC I/O stack, including attaching them to the main memory hierarchy as slower DRAM [21], and treating them as simple replacement of traditional HDD [31]. Burst buffer inserts a fast storage tier between compute resources and parallel file system to absorb I/O traffic that exceeds the capability of file systems. Owing to its high performance and cost-effectiveness, burst buffer has already been employed or is going to be employed in many HPC systems [1,6,8,43,44,46,55]. There are plenty of researches on evaluating or optimizing burst buffers of HPC systems [33,37,39,50]. Many vendors are already offering their burst buffer solutions, such as DataWarp [10] and IME [20]. However, those works and solutions are about shared burst buffers located in remote nodes. Our work is focused on node-local burst buffer, which offers linear performance scalability by co-locating storage and I/O processes.

A straightforward solution to manage the node-local fast storage devices is organizing them with file system. Many works extend conventional file systems or propose novel file systems to exploit the fast storage near CNs. Although their usage is different with that of cache systems, their strategies of managing metadata and data are worth comparison. Liu et al. present masFS [27] to exploit available memory and SSD resources on CNs. The architecture of masFS consists of many storage servers but only one metadata server, which can cause bottleneck when processing large amount of metadata queries. FusionFS [57] is a user-level file system that maintains its metadata with distributed hash table. Its directories are distributed across all nodes with subtree-based splitting. BurstFS [49] uses a distributed key-value store to manage its metadata. As the storage locations of key-value pairs in the store are determined with a hash function, the metadata are likely to be stored in remote nodes. Both FusionFS and BurstFS offer linearly scalable performance by co-locating data with compute processes. However, there are two common drawbacks in FusionFS and BurstFS. Firstly, their metadata are dispersed across all nodes; therefore, their performance of directory traversing (e.g., `ls` command) is very poor. Secondly, there is significant metadata overhead since the vast majority of I/O requests must retrieve the metadata remotely before accessing the data. Although BurstFS adopts lazy synchronization of metadata to provide efficient support for bursty writes, there are potential hazards of data coherence because the metadata are not consistent when clients are “lazy” to update them. Our work optimizes the metadata overhead by caching block metadata locally, which avoids many unnecessary metadata queries for the same data block and provides significant speedup to repetitive data access.

Caching is an alternative approach to manage the node-local fast storage. Congiu et al. [9] propose a set of MPI-IO hints extensions that enable users to cache data in fast, locally attached storage devices to boost collective I/O performance. Dong et al. [13] utilize flash-based SSDs as write back caches for checkpointing. Our work differs from these works in that it is designed to be a general-purpose cache. Holland et al. [19] conducted a detailed simulation to explore the design space of client-side flash cache. Their simulation results indicate that coherence maintenance of distributed flash cache would be the major holdback of its application. To avoid the costly maintenance of cache coherence, NetApp put forward their write-through distributed flash cache called

Mercury [7]. Mercury keeps no stale data in cache by writing dirty data to underlying file system and invalidating other copies in other clients immediately. Since all the data are written through to file system, Mercury degenerates to a read cache by forfeiting its advantage of quickly buffering massive amount of write out data.

Cooperative cache [11] is proposed to alleviate the costly coherence maintenance in distributed cache, in which clients cooperate together to cache each other's data and serve each other's misses. Memcached [15], a kind of cooperative cache, is a distributed key-value store that is often used to speed up dynamic database-driven websites by caching objects in RAM. BurstMem [52] and SFDC [14] extends the functionality of Memcached to construct general-purpose distributed caching systems in HPC environments. The locations of their cached data are determined by hashing the path of the data files, therefore the data are dispersed in all CNs. The performances of BurstMem and SFDC are limited since they introduce extra network overhead for processes to access recent data. Liao et al. [24,25] presented a application-level client-side file caching system called Collective Caching, which utilizes the memory buffers of all MPI processes to form a global cache pool. Our approach differs from Collective Caching in several facets. Firstly, the file metadata in Collective Caching are partitioned into small blocks and distributed in multiple nodes, which requires complex management and causes excessive metadata overhead. WatCache places the whole file metadata in just one client and adopts a metadata caching mechanism to minimize the metadata overhead. Secondly, we fully compare the respective bandwidth advantages of a single fast storage device and a parallel file system, and further optimize the data layout to boost the performance of parallel accessing in WatCache.

3 Background on HPC I/O patterns

3.1 Rank 0 I/O and all-rank I/O

Scientific applications often utilize Message Passing Interface (MPI) for parallel computing. In production HPC systems, there are hundreds to millions of MPI processes collaborating to solve a complex problem or performing an intricate simulation. Rank is another name of MPI process. Rank 0 I/O is a pattern that only the root process performs I/O. It is a common pattern observed in scientific applications [22]. In contrast with rank 0 I/O, all-rank I/O is a pattern that all processes perform I/O for their own needs. It is more intuitive for application developers because every process can get their required data directly.

Rank 0 I/O often exists in applications that need perform reduction operations, e.g., sorting or adding, to all the data distributed in the cluster. Thus, the data need to be collected and processed by a designated single process, and then written out by it. Applications that use non-parallel high-level I/O libraries, e.g., netCDF [36], also have to let rank 0 process in charge of all the I/O since those libraries do not support parallel accessing. An example of rank 0 I/O is illustrated in Fig. 1a. Rank 0 process reads in the whole input data and divides them into several parts. Then it distributes the data parts to other processes for calculating. Rank 0 process will collect the results and write them to file system when the calculation ends. Rank 0 I/O works well for

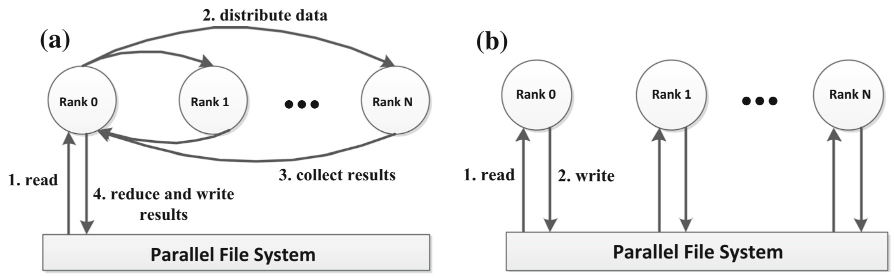

Fig. 1 A demonstration of rank 0 (**a**) and all-rank I/O (**b**) patterns

small files, such as input configuration and log files. But for larger amount of data, the performance is limited.

A common data layout strategy in cache systems is dividing data into blocks and striping them across all the CNs, just like the common data layout in parallel file systems. All the storage servers in parallel file systems are equally remote to processes in CNs, so that the performances of accessing data from they are about the same. Thus, striping data across multiple storage servers brings assured benefits. However, the bandwidth of accessing fast storage device in local CN is much better than accessing it from a remote CN. Caching data in local storage are more rational for client-side cache systems, because of the significant performance benefits of co-locating data and processes. Yet caching data locally only work well for all-rank I/O. For rank 0 I/O, the data are only stored in a single device. The performance of accessing large amount of locally cached data from a single storage device is poorer than accessing it from parallel file systems. Taken together, the data layout of cache systems should be carefully designed to adapt to both rank 0 I/O and all-rank I/O.

3.2 Small I/O

Large-scale multi-dimensional datasets are a common data type in scientific applications [28]. Because the storage space of file systems is one-dimensional, multi-dimensional data must be stored in row-major or column-major order. As a result, the logically contiguous data may be not contiguously stored in the file system. As applications access data based on their logical layout rather than their physical location, the logically contiguous I/O requests are in fact noncontiguous. Figure 2 shows an example of 4 processes accessing a two-dimensional matrix stored in row-major order. The matrix is partitioned into 4 blocks and each process handles one. When processes read in the first column of their blocks for analysis, the data they read are actually scattered throughout the whole physical space. Thus, they must issue many small noncontiguous requests to retrieve the required data.

In prior work [18], small I/O pattern has been observed in two leadership-class storage clusters of Oak Ridge Leadership Computing Facility (OLCF). We conduct an investigation in the 4 Lustre file systems in National Supercomputer Centre in Tinajin (NSCC-TJ). Figure 3 shows the I/O size distribution in 540 Lustre storage servers under production workloads, in a time period of 64 days (from June 27, 2016, to Aug

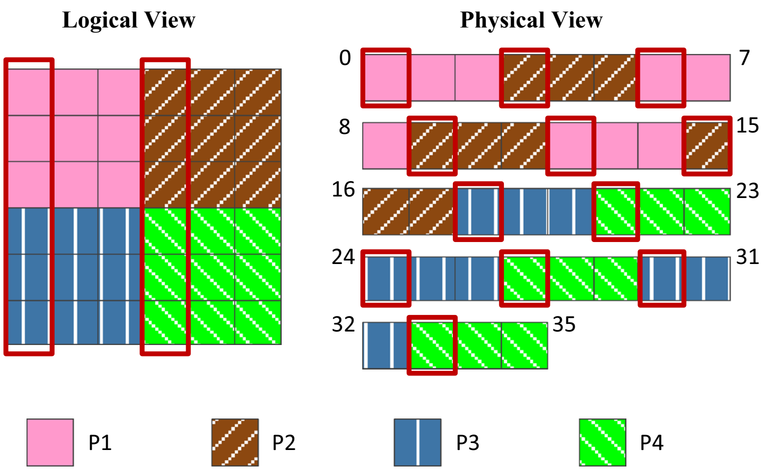

Fig. 2 An example of 4 processes accessing a 2-dimensional matrix stored in row-major order

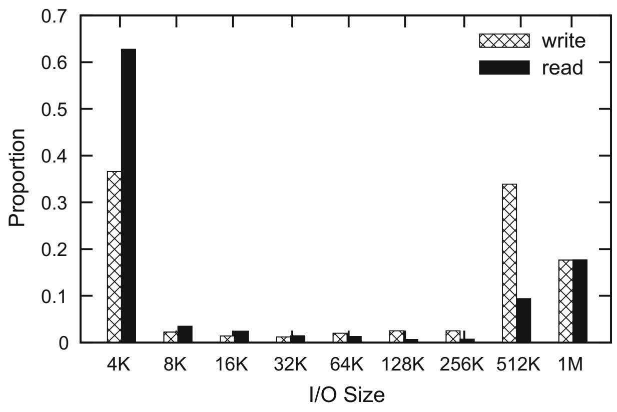

Fig. 3 The I/O size distribution on 4 Lustre file systems in NSCC-TJ

29, 2016). Since I/O requests larger than 1 MB are divided by Remote Process Call (RPC), the I/O sizes we captured range from 4 KB to 1 MB. From the results we find that 4 KB-sized I/O takes 62.7% of all read requests and 36.6% of all write requests. Small I/O has become an I/O pattern that cannot be overlooked in HPC workloads.

Parallel file systems like Lustre offer high-performance capabilities for large, consecutive parallel I/O; however, they are not capable of handling small, noncontiguous I/O. Fast storage tier on the CN side seems an optimal solution to absorb small I/O requests for its greater random access performance. However, in client-side cache systems, the overheads of querying metadata and retrieving data from remote nodes are still the main constraint on small I/O performance. Thus, the network overheads need to be minimized to face the challenge of small I/O in HPC workloads.

4 Workload-aware temporary cache system

4.1 Temporary cache system

HPC systems use job scheduling systems like Slurm [56] to manage the massive computing resources and schedule jobs submitted by users. Take Tianhe-1A in NSCC-TJ for example, whose architecture is shown in Fig. 4, users first login to the Login Nodes (LN), where they modify and compile their programs, prepare the configuration and input files, and at last submit jobs to Slurm. Slurm allocates the required number of CNs to users after arbitrating contention for resources. After acquiring permission from Slurm, user jobs can exclusively use the allocated CNs to run their applications.

The CNs in a HPC system are dynamically divided into many logical partitions by Slurm, each of which is then allocated to a user job. These partitions are running different applications, processing different datasets, exhibiting different I/O patterns and requiring different resources of compute, network and storage. There are no information exchanges or data sharing among logical partitions, so it is unnecessary to construct a global cache pool for all partitions. Instead, we create standalone temporary cache pools for different logical partitions. Before allocating CNs to a job, Slurm starts up the I/O service processes of WatCache on the CNs in advance. When the job ends and is about to exit, Slurm informs WatCache to write back all the dirty data. After that, the WatCache will be destroyed and its CNs will be available for allocation to other jobs again. The benefits of temporary cache system include the following.

(1) *Better locality* Users have no authority to access other users' data, so their jobs will never have the opportunity to access the same files. In this case, jobs of different logical partitions will not gain any benefits of data sharing from a global cache. On the contrary, in a global cache the competition for limit cache space would decrease

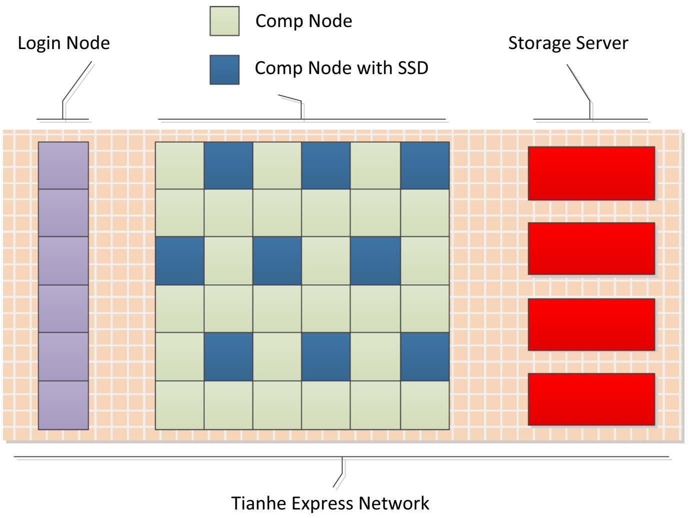

Fig. 4 The architecture of Tianhe-1A in NSCC-TJ

the hit rate. Our design separates different users from each other and isolates cache competition; therefore, it offers much better cache locality.

(2) *Easier to exploit I/O pattern* CNs of the same logical partition are always working together to solve the same problem; therefore, their I/O requests are highly related to each other. It provides an opportunity to discover possible I/O patterns of this application in the partition, which can be used to further enhance its I/O performance. On the contrary, in a global cache the I/O requests of different applications are interleaved with each other, which introduces difficulty to detect I/O patterns.

(3) *Isolate fault domains* In distributed storage systems, one node failure may cause global unavailability. The proposed temporary cache system possesses better reliability because it isolates the fault domains and stops failures spreading from one logical partition to the whole system. So that when a failure happens, the worst case is the downtime of a local temporary cache, while other temporary caches in the system remain functioning.

4.2 Workload-aware node allocation

There are tens of thousands of CNs in leadership-class HPC systems [45] currently, and the number of CN is still growing as the computing capacity is developing to exascale. Although the price per bit of PCIe SSD is declining over time, there still is high budget pressure to install fast storage devices for all the CNs. Besides, HPC facilities serve various kinds of applications every day, including data-intensive applications and compute-intensive applications. The I/O bandwidth of storage systems is mostly consumed by a small number of data-intensive applications [29]. It is a waste of expensive storage devices to allocate burst nodes to applications that require little I/O bandwidth. In order to ensure cost-effective use of fast storage devices, we only attach them to a limited number of CNs, as shown in Fig. 4. We propose a workload-aware node allocation method in Slurm to efficiently utilize the limited number of burst nodes.

A typical job submission command in Slurm is shown below. Slurm receives the job submissions from users and allocates the most suitable CNs to them in consideration of different priorities and requirements.

```
srun -N #nodes -n #procs ./app.exe
```

The workload-aware node allocation method adds a hint extension to current submission commands of Slurm, as illustrated in Table 1. Users can use the option `-b` to specify the degree of how much their jobs require I/O bandwidth. The enhanced job submission command is shown below.

```
srun -N #nodes -n #procs -b io_support ./app.exe
```

We add a field in the original node information array of Slurm to specify whether this is a burst node attached with node-local fast storage. Before Slurm node manager assigns CNs to a job, it first checks the `-b` hint. If the job requires high I/O support, node manager assigns as many burst nodes as possible to the job. If the job requires medium I/O support, node manager assigns 50% burst nodes among the allocated

Table 1 Workload-aware node allocation hints in job submission commands

Hint (`-b`)	Description
`Low`	Job requires little I/O bandwidth, disable burst nodes allocation
`Medium`	Job requires moderate I/O bandwidth, allocate 50% burst nodes
`High`	Job requires significant I/O bandwidth, allocate as many burst nodes as possible

CNs. If the job requires little I/O support, node manager will not assign burst nodes to it. Besides, the I/O service processes of WatCache will not be started in theses CNs, and the I/O calls are directly issued to Lustre clients. Usually, users are domain experts who cannot accurately know how much I/O dominant their applications are. Therefore finer-grained level of I/O workload is unnecessary, and we only provide three levels of I/O support so that users can select it easily.

The fast storage devices are uniformly distributed in all the CNs instead of being grouped in several cabinets. This approach avoids too much network overhead of transferring data between common CNs and burst nodes when not all allocated CNs are equipped with fast storage. For example, when available burst nodes are not enough to satisfy jobs that require high I/O support, or 50% burst nodes are allocated to jobs that require medium I/O support, the data accessed by a CN without local storage must be cached in a burst node. Node manager preferentially chooses common CNs that are nearer to burst nodes to assign, so that network overhead is minimized.

4.3 Support inter-job caching

As a temporary cache system whose lifetime is the running time of a job, WatCache only provides benefits of data reuse within a single job. However, in production HPC systems, there are many situations of inter-job data reuse. For example, users need to submit job, evaluate results, tune configuration and then submit the job again to iteratively optimize the experimental results [40]. As the consecutive jobs are processing the same dataset, the data cached by a former job can be completely reused by the latter job. Another example is the workflows, which consist of multiple coupling applications working together to complete a processing flow [4,42]. The latter application in the workflow needs to read and process the data that the former application wrote out. If the intermediate data between applications can be cached in a large cache system instead of stored in underlying file system, the I/O performance of the workflow can be boosted significantly.

In order to enable WatCache to support inter-job caching, we let WatCache starts along with another two job submission commands, `sbatch` and `salloc`. With `sbatch`, users can pack multiple applications into one single batch file and run them sequentially by submitting the batch file. As CNs are allocated to users when the batch job starts and will be recycled after the job ends, the lifetime of WatCache is extended to the running time of multiple applications within the batch job. So that the cached data can be shared among multiple applications. `sbatch` works well for workflow applications, however, the temporary cache still cannot persist across multiple batch jobs when users iteratively submit a same job. To fix that, users can use `salloc` command to get exclusive access to CNs for a duration of time, which Slurm will not recycle until users proactively relinquish them with `scancel`. WatCache starts along with `salloc` and will survive across multiple jobs until users proactively destroy it. As a result, WatCache can be shared among multiple jobs.

5 Design of WatCache

5.1 Overview

WatCache is a temporary cache system that resides on a number of CNs allocated for a job. In most cases, its lifetime is the running time of a job (the only exception is the inter-job caching described in Sect. 4.3). WatCache is deployed when the CNs are allocated to a job and will be destroyed when the job ends. Therefore, it is essential to make WatCache light-weighted so that the I/O services can be established and terminated quickly. We implement WatCache in user space to meet such requirement.

WatCache is implemented under server-client mode as shown in its architecture in Fig. 5. In each CN there is a server daemon process running on it. The servers are initiated when Slurm allocates CNs to jobs. Each server actually plays two different roles, metadata server which handles metadata queries, and data server which deals with data requests and manages the local storage. We adopt a distributed metadata management mechanism to avoid the metadata service becoming a bottleneck. Note that, in our design not all CNs are required to have a fast storage device attached; therefore, not all servers have to be data servers. However, all servers are metadata servers to maximize the load balance of metadata service. Before recycling CNs when job ends, Slurm will inform all the data servers to write back their dirty data and then terminates all their I/O service processes.

In the clients, we use wrapper functions to intercept the I/O calls of applications and redirect them to servers. The wrapping approach has been widely used in I/O tracing tools like Darshan [12], whose overhead is believed to be undetectable for jobs fewer than 10,000 processes [34]. There are two possible locations in the I/O stack to wrap I/O calls, MPI I/O and POSIX I/O. It is simpler to wrap MPI I/O because MPI I/O has fewer types of functions and more released semantics [25]. But MPI I/O is not as widely used in HPC applications as people generally believe. Recent research [29] on the I/O behaviors of thousands of HPC applications indicates that, only one quarter of applications use MPI I/O to access data, and POSIX is more popular among many-processor jobs. So we decide to intercept POSIX I/O calls to adapt to the widest spectrum of applications.

Currently, we wrap several main POSIX I/O functions to construct WatCache prototype system, including `(f)open`, `(f)read`, `(f)write`, `(f/l)seek`, `(f)close`, `(f/l)xstat`, etc. We insert the procedures of communicating and data accessing in these wrapper functions and pack them into a library, which can be linked to applications while compiling. When a process first opens a file, a client is initiated in the address space of the process through the intercepted `(f)open` call. The client will be terminated automatically when its host process exits. The I/O calls of applications are delivered to WatCache instead of VFS through these wrapper functions, where we can manipulate and rearrange the calls to make full use of local storage.

In the rest of this section, we will elaborate design details of the distributed metadata management and lock management mechanisms. To support the challenging I/O patterns described in Sect. 3, we designed a data layout strategy and a metadata caching mechanism. WatCache preferentially stores data in local storage to maximize the benefits of co-locating processes and data. Meanwhile, it supports data to be placed in multiple storage devices to promote the performance of parallel accessing when data are too large. To deal with small I/O, WatCache caches metadata locally to minimize the overheads of metadata queries of I/O requests.

5.2 Metadata management

We adopt a hash-based, distributed metadata management mechanism to handle metadata queries in WatCache, instead of a centralized one which stores all metadata in one server. Distributed mechanism has better performance and reliability because it prevents the single metadata server becoming a hot spot and causing single point of failure. Hash-based metadata management has been widely used in parallel file systems like Ceph [53] and Gluster [16]. However, it is widely criticized for its low scalability and poor performance of directory traversing. WatCache perfectly averts these two issues. Firstly, there are no user-initiated metadata traversing in cache at all. Users utilize WatCache transparently so that they cannot proactively inquire what's cached in WatCache. To be noted, their *ls* command for showing contents in directories is handled by underlying file system. Secondly, the lifetime of WatCache is about the same of the running time of a job. The scope of WatCache is determined when the

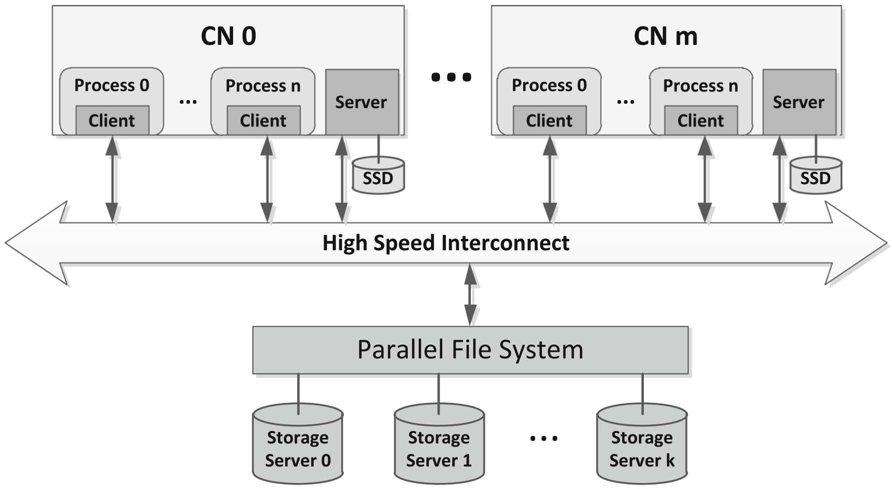

Fig. 5 Architecture of WatCache. Clients are established in the address space of user processes when they first access data, and servers are daemon processes in the CNs

job starts, and there is no need to scale out during the execution. Thus, the scalability problem of hash approach is avoided. Taken together, the hash approach is applicable for managing metadata in WatCache.

The metadata are distributed in all the metadata servers and are accessible to all processes with a hash algorithm, which is depicted in the Eq. 1. The hash function takes the absolute path of the requested file as input parameter. It determines the metadata server ID by performing modulus on the hash value with the total number of metadata servers. The hash value is uniformly distributed over the output range of the hash function, which grantees that all metadata servers can be chosen at roughly the same possibility. Since all the clients use the same hash function and the number of metadata servers remains the same throughout the lifetime of WatCache, the I/O requests for the same file can always find the same metadata server.

$$ID_{metadata\_server} = Hash(file\_path)\%Number_{metadata\_server} \tag{1}$$

In metadata servers, *radix trees* are used to hold the metadata of the files it manages. Each radix tree records the metadata of a file, which is organized with key-value pairs. The key is a data segment offset, which in WatCache is a data block ID. The value is the ID of the data server where the data block is stored. Radix tree is storage efficient because its required storage space grows cumulatively with the data range it handles. In the meanwhile, it supports range metadata querying of consecutive data blocks, so that the metadata query of a large data range can be served quickly.

Figure 6 shows an example of I/O procedure in WatCache. A process in CN 1 wants to read the first data block of `path1/file1`. The I/O call is intercepted by the client within the process. The client first determines the server which holds the metadata, then acquires the metadata from it. If the requested data block has already been cached in WatCache, metadata server searches the radix tree that holds the metadata of the requested file and returns the metadata. If the requested data block has not been cached, metadata server will create a new entry for it in the radix tree, and preferentially

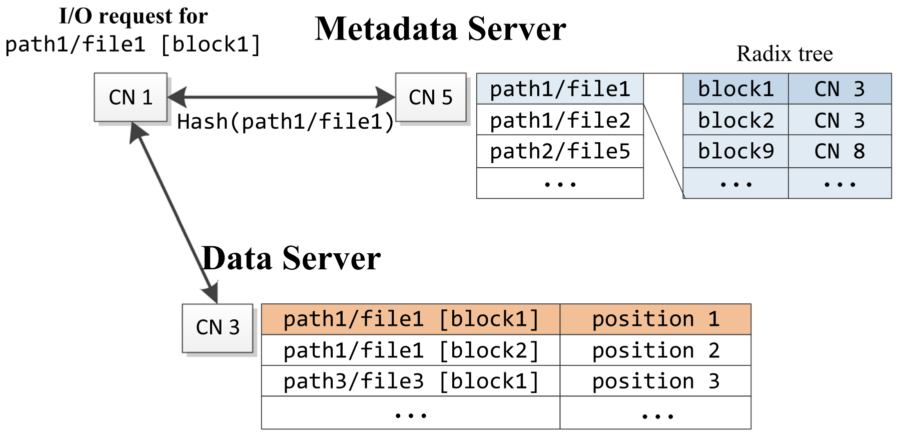

Fig. 6 An example of I/O procedure in WatCache

designate the requesting node CN 1 to store the data. After acquiring the metadata, client then retrieves the requested data from the data server that the metadata indicate. If the data have not been cached in the data server, data server will read the data from file system and cache it in local storage. In order to ensure data consistency, the data have to be written to the data server before being returned to user process.

If the data server happens to be on the same node with client, client directly accesses data in local storage after acquiring a lock of the data (which will be described in Sect. 5.3). Otherwise, the client will transmit data from data server through network. Our implementation of data transfer is based on GLEX [54], a high-performance communicating system on Tianhe Express Interconnect. For write I/O the procedure is similar. What is different is that the data will be buffered in flash cache immediately whether or not they are cached before. The dirty data will be written back to underlying file system with a write behind manner.

5.3 Lock management

In a distributed cache, multiple copies of a same piece of data may be cached in different clients at the same time. When a client modifies its copy, the other copies need to be invalidated or updated. When a client caches a dirty copy, the copy needs to be written back before any other client reads it. HPC applications use hundreds to thousands of processes to collaborate on the same datasets. Therefore, it is likely that different processes access the same data blocks, which results in data conflicts. The data coherence is usually assured by a complex distributed lock management in parallel file systems, e.g., LDLM [51] in Lustre. The distributed lock management is prerequisite for a functional distributed cache system. However, it introduces great complexity to system design, and the frequent operations of invalidation and writing back cause significant impact to performance.

WatCache provides transparent, shared cache space to the CNs by intercepting I/O calls before them are issued to VFS. There are no data residing in local storage that are not visible to other processes. Therefore, the coherency is maintained by the metadata management, without the need to invalidate or write back any data. As a result, the lock management in WatCache is much simpler than the client cache of a parallel file system.

We design a pipelined lock management mechanism to ensure safe access to cached data. POSIX atomicity requires that, all bytes written by a single write call need to be either completely visible or invisible to a following read call [35]. In WatCache, consecutive data blocks can be stored in multiple CNs, and accessing these blocks in different CNs without careful lock management would violate POSIX atomicity. We implement lock managers in data servers to ensure atomicity. After acquiring metadata, clients must seek grants from corresponding lock managers before accessing data. Lock requests are issued to lock managers in a strictly increasing order of block position in the file. To avoid possible deadlocks, the subsequent lock request must be issued after the previous lock request has been acquired. In order to accelerate the procedure of obtaining locks, the already-locked data blocks can be accessed by clients while waiting for subsequent locks. However, the locks must be released together to ensure atomicity when I/O request finishes.

Note that, the data server is a daemon process in user space. When it manages the data transfer between its local storage and underlying parallel file system, the I/O calls are passed through VFS. So it is possible that there are data residing in the kernel page cache. As mentioned above, when data cached in clients that are not visible to all other processes, there are potential hazards of data coherence. However, WatCache need not maintain the coherence of the data because data are in the client cache of parallel file system. It is the file system's responsibility to make sure the data in the kernel cache are up to date. In addition, WatCache stores only one copy of data in its cache space, so that a data block can only be cached in one data server. In this way WatCache ensures that all the I/O requests for one data block are served by just one data server. This approach avoids multiple copies residing in different client caches of file system. It further eliminates the possible contention in file system since a data block will be accessed by only one process.

5.4 Data layout strategy

5.4.1 Cache block size

In HPC workloads, it is common that many processes access one shared file in parallel [22]. If the entire file is stored in one single data server, the performance of parallel accessing the file will be limited. In WatCache we divide the files into blocks of fixed size, so that processes can accurately cache their requested data in local data server, and the performance of parallel accessing is improved.

The size of data blocks has a great influence on the performance of WatCache. Smaller block size leads to finer data granularity, which is conducive to place the cached data exactly where it is needed. But the network bandwidth is usually poorer when transferring data in smaller blocks. We test the bandwidth of our GLEX-based network transferring implementation. The results in Fig. 7 indicate that the peak bandwidth will be achieved only when the block size is larger than 256 KB. So the block size in WatCache is better to be larger than 256 KB. On the other hand, if the block size is too large, processes are more likely to cache data they do not need, which increases the delay of I/O requests. Considering that the stripe size of underlying file system is usually set to 1 MB, it is better to align cache block size to the stipe size. Otherwise, if cache block size is smaller than stripe size, a data stripe in file system can be cached in multiple cache blocks of different nodes, which leads to unnecessary contention when the data stripe is accessed. Based on above analysis, we set the cache block size to 1 MB.

5.4.2 Large and successive I/O detector

WatCache preferentially stores data blocks in local storage to avoid the extra network overheads when accessing data. But caching data locally does not apply to all scenarios. Let us assume the average bandwidth of a PCIe SSD is B_{SSD}, and its latency to respond

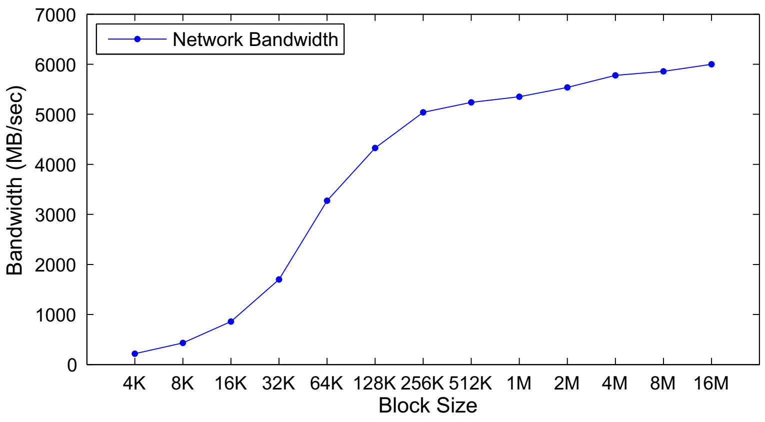

Fig. 7 The performance of transferring data over network with different block sizes

an I/O request is L_{SSD}. There is no network latency when all data are stored locally. When requesting a piece of data in the size of S_{data}, the time to complete this I/O in local SSD can be estimated by Eq. 2.

$$T_{\text{cache}} = \frac{S_{\text{data}}}{B_{\text{SSD}}} + L_{\text{SSD}} \tag{2}$$

T_{cache} determines the max bandwidth that processes can achieve in local SSD, which is the upper bound of performance when multiple nodes concurrently access the data stored in one SSD.

In parallel file systems files are striped over multiple storage servers. If this I/O request is issued to file system instead of WatCache, process can access the data in multiple storage servers in parallel and achieve a higher aggregated bandwidth. Let us assume the average bandwidth of each storage server is B_{disk}, the stripe size is S_{stripe}. Even though S_{data} may no align with S_{stripe}, and S_{data} may be so large that each storage server may hold more than one stripe, for ease of demonstration we can roughly estimate the number of storage servers involved in this I/O request with $S_{\text{data}}/S_{\text{stripe}}$. As the latency of a parallel file system is determined by its slowest storage server, we assume the maximum latency of these storage servers is L_{disk}. The network latency is represented by L_{network}. The time to complete this I/O in parallel file system can be calculated by Eq. 3.

$$T_{\text{pfs}} = \frac{S_{\text{data}}}{\frac{S_{\text{data}}}{S_{\text{stripe}}} \times B_{\text{disk}}} + L_{\text{disk}} + L_{\text{network}} \tag{3}$$

In order to determine at what data size WatCache outperforms parallel file system, we can combine Eqs. 2 and 3 by letting $T_{\text{cache}} < T_{\text{pfs}}$. Thus, Eq. 4 can be derived.

$$S_{\text{data}} < \left(\frac{S_{\text{stripe}}}{B_{\text{disk}}} + L_{\text{disk}} + L_{\text{network}} - L_{\text{SSD}} \right) \times B_{\text{SSD}} \tag{4}$$

After putting realistic parameters into Eq. 4, we conclude that when the data size is smaller than about 100 MB, WatCache performs better than the underlying file system. In other words, if a client contiguously accesses locally cached data more than 100 MB, WatCache is worse than none. Although the practical peak bandwidth of parallel file system is usually lower than the theoretic sum of peak bandwidths of all storage servers, the results are instructive for designing data layout policy.

Based on above analysis, the strategy of caching data locally works well for all-rank I/O (described in Sect. 3.1) because all processes access data in their local storage. However, rank 0 I/O cannot take advantage of the aggregate bandwidth of multiple storage devices if all the data are stored in a single node-local device. Such performance impact is common in workflows, too. The applications in the workflow are often developed by different communities. Besides, domain scientists who developed the applications are more focused on the scientific problem rather than delving deep into the data flow between applications. Therefore, data locality between applications can hardly be guaranteed, i.e., data written by a CN in the former application may be accessed by many other CNs in the latter application. It is better to avoid storing too much data on a single CN to improve the performance of parallel access.

We design a large and successive I/O detector (LASIOD) to take care of this problem. LASIOD adds an assessment process in the original caching strategy, and only the I/O that pass the assessment can cache locally. Since all the I/O calls for a file consult the same metadata server, it is applicable for us to put LASIOD in the metadata server side to monitor all requests for the file. As depicted in Algorithm 1, when detecting any I/O larger than a threshold size, LASIOD would divide the requested data into a certain number of parts and let the same number of fast storage devices to accommodate the data, including the device on the source CN. If there are a large number of sequential I/O requests, and their total requested data size exceeds the threshold size over time, LASIOD will stop choosing the source CN to cache data for latter requests, and designate another CN instead. By distributing consecutive data that exceed the threshold to multiple clients, it achieves higher aggregate bandwidth.

5.5 Metadata caching

Since the metadata of files are uniformly distributed in all the servers, it is highly possible that clients need to querying metadata remotely. Considering that the latency of a PCIe SSD and the network latency are approximately at the same order of magnitude, the remote metadata query has significant impacts on I/O performance, especially for small I/O.

We design a metadata caching mechanism to promote the metadata querying performance. The metadata of a file block are generated when the data are first cached in, and any client accesses the block would get a copy of the metadata and then stores it locally. Next time the client accesses the piece of data or any adjacent piece of data in the same file block, the metadata can be retrieved locally instead of being queried across the network. A record is inserted in the metadata server to bookkeep the clients that cache the metadata copies. The records will later be used for deleting all the metadata copies when data are about to be flushed out. Instead of treating the metadata

Algorithm 1 Decide cache location with LASIOD

```
Input: an I/O request $r$ from client $i$
Output: cache location of the requested data
1: if $r$ is not adjacent with last request then
2:     successive size = 0
3:     if requested size $<$ threshold size then
4:         cache location = client $i$
5:     else
6:         n = requested size / threshold size
7:         cache location = client $i$, $(i+1)$, …, $(i+n-1)$
8: else
9:     successive size += requested size
10:    if successive size $<$ threshold size then
11:        cache location = client $i$
12:    else
13:        $j$ = (successive size - requested size) / threshold size
14:        if requested size $<$ threshold size then
15:            cache location = client $(i+j)$
16:        else
17:            n = requested size / threshold size
18:            cache location = client $(i+j)$, $(i+j+1)$, …, $(i+j+n-1)$
19: return cache location % total client number
```

cache of different processes independently, we merge them to a CN-wide global cache with a lightweight NoSQL key-value store SHF [41]. All processes in the same CN can share their metadata information, and the cache hit rate is promoted.

With metadata caching mechanism, clients need not obtain metadata from remote servers when processes request for recent data. The required metadata have already been cached locally on the first access. When processes access new data, clients have to obtain metadata remotely. The network overhead of metadata queries has a certain impact on the I/O performance. However, the impact is not significant. The reasons are illustrated in following two cases. For read I/O, the missed data need to be retrieved from remote, disk-based storage system, whose overhead is about 2 orders of magnitude greater than network overhead. Therefore, the network overhead of obtaining metadata only accounts for a very small proportion of total overhead. For write I/O, if the size of requested data is large, e.g., larger than 1 MB cache block size, the network overhead is about 1 to 2 orders of magnitude smaller than the overhead of writing data in PCIe SSD. If the size of requested data is small, metadata caching mechanism has potential benefits of serving metadata queries locally when small pieces of data in the same block are accessed. Taken together, the metadata caching mechanism avoids many unnecessary metadata queries for the same data block and offers speedup to repetitive data access.

When designing distributed cache, what people concern most is the high cost of maintaining cache coherence, which causes dramatic loss of caching benefits. When caching file data in multiple clients, coherence needs to be maintained if any client tries to modify its locally cached data. The same is true for caching metadata in multiple clients. In order to prevent the costly coherence maintenance, WatCache forbids dynamically migrating cached data among clients, so that the metadata that record the location of the data remain invariant throughout its lifetime in WatCache.

Dynamically migrating data bring benefits when a client repeatedly accesses data stored remotely [24]; however, the complexity of management limits its effectiveness. Besides, the metadata caching is applicable more widely than data migration since it has the potential of reducing metadata overheads in all I/O requests. In WatCache, the cached data will never be relocated throughout their lifetime until being flushed out. Thus, there is no cache coherence problem when caching metadata in multiple clients in WatCache.

5.5.1 Compliant with LASIOD

There are potential hazards of LASIOD effectiveness when I/O calls bypass the metadata server to obtain metadata through metadata caching. For example, the LASIOD on the server cannot perceive the I/O in clients when the metadata queries are all satisfied in local metadata cache. We solve this problem by forcing I/O requests carry extra information in metadata queries.

For large I/O, if not all the metadata of requested data blocks are satisfied locally, it needs to consult the server to retrieve the remaining metadata. We let the client send the original range of this large I/O along with the query of remaining metadata to the server, so that LASIOD on the server can assess and determine its cache location. On the other hand, if all the requested metadata are satisfied locally, it means that the I/O has already been assessed by LASIOD before.

For small successive I/O requests, if all metadata queries hit the local metadata cache, it means that the I/O requests have been assessed by LASIOD before. If all metadata queries miss the local cache, LASIOD on the server will assess them when dealing with their metadata queries. If only a part of their metadata queries are satisfied locally, the LASIOD on the server cannot extract the whole successive pattern with the remaining metadata queries. To compensate for this shortcoming, we develop a successive I/O detector (SIOD) on the client side, whose assessing procedure is shown in Fig. 8. When dealing with an I/O request, if the request is believed to be a part of successive I/O series, the complete range of these successive I/O will be sent to server along with the original metadata request. Then the LASIOD on the server can decide whether to cache the I/O request locally with the extra information it brings. Note that, if small I/O calls from different clients that are not detected as successive I/O by their local SIOD add up together to be successive, they will be detected by LASIOD on server side.

6 Evaluation

6.1 Experimental setup

6.1.1 Comparing systems

Memcached [15] is a distributed in-memory key-value store that is intended for use in speeding up dynamic web applications by alleviating database load. Many efforts, e.g., BurstMem [52] and SFDC [14], have modified and extended the functionality

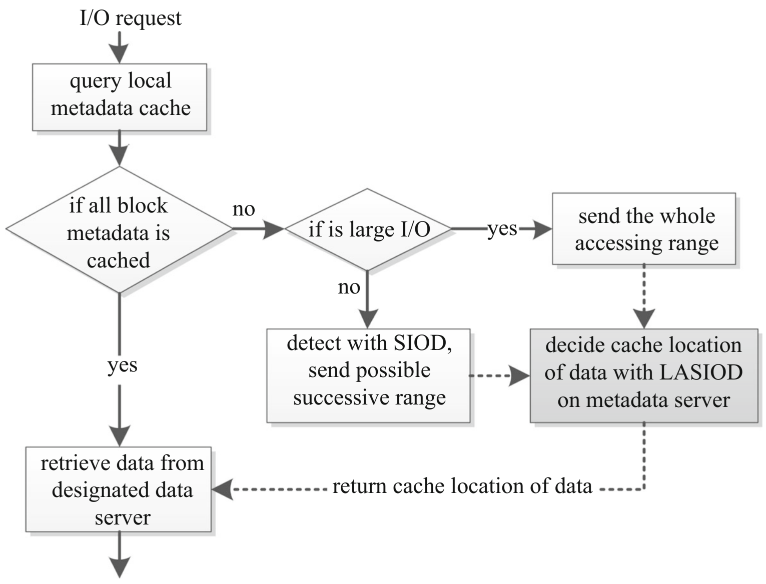
I/O request
query local metadata cache
if all block metadata is cached
no
yes
if is large I/O
yes
no
send the whole accessing range
detect with SIOD, send possible successive range
decide cache location of data with LASIOD on metadata server
retrieve data from designated data server
return cache location of data

Fig. 8 Make client-side metadata caching mechanism compliant with LASIOD, as clients may bypass metadata servers to obtain metadata. The dotted lines represent communication between client and metadata server

of Memcached to construct a distributed cache system to alleviate I/O bottleneck in HPC systems. In order to compare our proposed WatCache with such Memcached-like cache systems, we develop a cache system called McCache that emulates the architecture of Memcached. In McCache, the location of cached data is determined by the hash value of its file path, instead of by querying metadata servers. The simple design eliminates the costly metadata overhead in each data access. However, since the cached data must be stored in a node determined by the hash function, there is a sporting chance that the data will be stored in a remote node. As a result, McCache loses the performance advantage of serving I/O calls locally.

To the best of our knowledge, the most recent research on merging node-local fast storage into a global storage space is BurstFS [49]. BurstFS is a scratch file system that adopts a distributed key-value store to manage the metadata. In contrast, WatCache adopts a hash-based distributed metadata management, with a metadata caching strategy that optimizes the metadata overhead. Although BurstFS is a file system, its approach of managing data and metadata is worth comparison. Therefore, we develop a cache system that emulates the architecture of BurstFS, called KvCache. KvCache also adopts a distributed key-value store named MDHIM [17] to manage metadata. MDHIM stores the block metadata as key-value pairs across the cluster. The key is the unique ID of a data block, e.g., the combination of file path and the block offset. The value is the metadata, which records the stored location of the data. Since the keys are mapped to specific server by means of a hash algorithm, there is a high possibility that the value is stored in a remote node.

Table 2 Differences in metadata management and data placement among McCache, KvCache and WatCache

Systems	Metadata management		Data placement	
	Method	Overhead	Location	Overhead
McCache	Determined with a hash algorithm	Low	Mostly remote	High
KvCache	Stored in a distributed key-value store	High	Mostly local	Low
WatCache	Stored in metadata servers with metadata caching	Medium	Mostly local	Low

The main differences among McCache, KvCache and WatCache are illustrated in Table 2. McCache does not use metadata to record the location of cached data, instead it calculates the data location with a hash function. Without round-trip metadata queries in every I/O calls, McCache has the lowest latency of determining data location. However, the latency of accessing data in McCache is the highest since there is no guarantee that data can be stored in local storage. KvCache and WatCache use metadata to record the location of cached data; therefore, their data can be stored and accessed in local storage. However, the performance improvements of accessing data are at the expense of metadata efficiency, because the metadata must be retrieved mostly remotely before accessing locally cached data. WatCache optimizes the metadata overhead by storing metadata in servers with an extra metadata caching strategy; thus, its metadata overhead is lower than KvCache.

6.1.2 Storage media

WatCache supports various kinds of fast storage media to meet the requirements of hardware interface or budget, including SATA SSD, PCIe SSD, NVRAM, memory, etc. The downward trends in hardware costs and the developments in software stack like NVMe [32] have made PCIe SSD a cost-effective media for the node-local storage layer. However, the limited number of our PCIe SSD hampers our intention to deploy a large-scale WatCache. We decide to use RAM disk to construct WatCache for demonstration. The substitution has some rationality because their bandwidth is on the same order of magnitude and the gap is narrowing rapidly [38]. The RAM disks in WatCache are mounted with `tmpfs`, a file system that keeps all its files in the kernel page cache. The size of each RAM disk is 8 GB in our tests.

6.1.3 Testbed

Our evaluation was performed on Tianhe-1A in NSCC-TJ. We use 256 CNs, each of which has 2 Intel Xeon X5670 CPU (6 cores, 2.93 GHz) and 24 GB of RAM. Apart from the memory space used by the OS kernel and RAM disk, the remaining memory space can be utilized by kernel page cache is around 15 GB. Tianhe Express-1 interconnects the CNs with the a Lustre file system. The Lustre we used consists of 1 MDS and 128 object storage servers (OST). The peak performance of a single OST is about 200 MB/s. In Lustre the stripe size is set to 1 MB, and the stripe count is set to − 1. In this setting file data are divided into 1 MB-sized stripes and distributed in all 128 OSTs. This setting remains the same in all following tests unless otherwise specified. The threshold in LASIOD is set to 256 MB, which means that when data accessed by large I/O or successive I/O exceed 256 MB, the data will be partitioned and stored in multiple devices. All tests are conducted 5 times to make sure the results are stable enough. To avoid the caching benefits of OST cache and CN cache between tests, we use distinct datasets for each test.

6.2 Overall performance

We first evaluate the overall performance of WatCache. This test is conducted in 64 CNs, on each of which runs an IOR process, sequentially writing or reading a unique 256 MB file. All the files accessed by IOR processes have not been cached before the test. The sizes of the I/O calls vary from 4 KB to 4 MB.

Figure 9 shows the write bandwidth of McCache, KvCache and WatCache. Although processes in McCache directly access data without querying the metadata from a remote server or a distributed key-value store, its performance is the lowest at all I/O sizes. The reason is that the data are mostly stored in remote nodes and needs to be transferred from/to the requester over network. In contrast, data in KvCache and WatCache are stored in local storage and can be accessed much faster. The bandwidth of McCache increases along with the I/O size since bigger blocks are more efficient to transfer over network, and it reaches the peak at about 1 MB size.

In KvCache, processes query the metadata from a distributed key-value store during every I/O call. As the metadata are distributed in the cluster with a hash function, it is likely that the requested metadata are stored in a remote node. Thus, although processes in KvCache can access the data in local storage, they must query metadata remotely. For this reason, the bandwidths of McCache and KvCache are about the same at 4 KB I/O size. As the I/O size grows, the overhead of acquiring bigger data blocks remotely

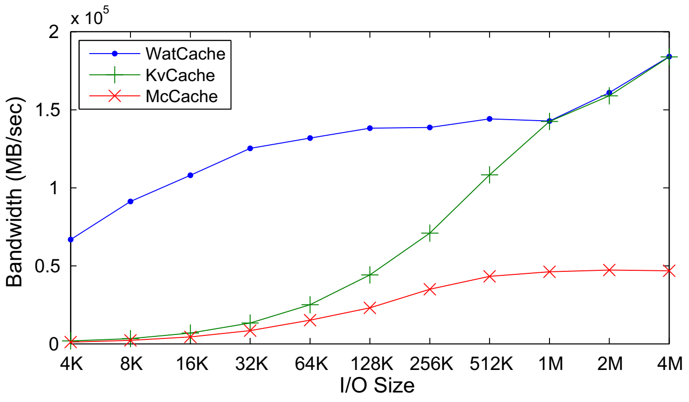

Fig. 9 Overall write performance of McCache, KvCache and WatCache

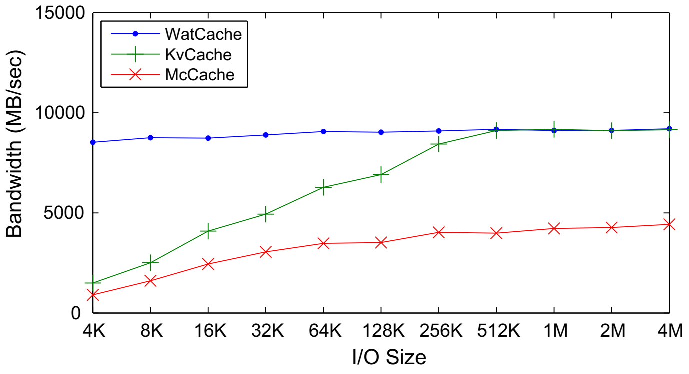

Fig. 10 Overall read performance of McCache, KvCache and WatCache

is far in excess of acquiring metadata remotely. So the performance of KvCache is faster than McCache at bigger I/O sizes.

WatCache outperforms the other two cache systems significantly at small I/O sizes. In WatCache, when accessing a data block with multiple small I/O requests, only the first request needs retrieve the block metadata from a remote metadata server. The block metadata are cached locally once the first request finishes, and the following requests can retrieve metadata quickly in local metadata cache. However, when I/O size grows larger than the cache block size, every I/O request has to retrieve its own metadata by itself, since no other requests share the same data block. As a result, its performance is about the same with KvCache at I/O sizes bigger than 1 MB.

Note that, the bandwidth of WatCache rarely grows at I/O sizes between 64 KB and 1 MB. For I/O sizes smaller than cache block size, the time of retrieving metadata is constant, because no matter how many I/O requests are accessing a data block, only one metadata query needs to be issued to remote metadata server. Therefore, the time of writing data has become the dominant factor that affects the overall bandwidth. The peak write bandwidth of tmpfs is reached at about 64 KB; therefore, the bandwidth of WatCache stops growing at 64 KB I/O size. As I/O size grows larger than cache block size, e.g., 4 MB, every 4 data blocks need only one metadata query since the 4 queries are packed together, which leads to 4$\times$ shorter time of retrieving block metadata in average. Thus, the aggregate bandwidth continues to grow a little with I/O size bigger than cache block size.

Figure 10 compares the overall read performance of McCache, KvCache and WatCache. The files read by processes are not cached in the fast storage tier before the test. The cache systems have to read the requested data from the Lustre before serving the read calls; therefore, the read bandwidths are limited to the peak bandwidth of Luster. At I/O sizes larger than 512 KB, the read bandwidths of WatCache and KvCache both reach the peak performance. As data cached in McCache are mostly stored in remote nodes, the read bandwidth is limited due to the extra network overhead. WatCache's performance is higher than KvCache at small I/O sizes owing to the optimization of metadata caching.

6.3 Overhead of metadata service

Since all CNs used by applications act as metadata servers regardless of whether they have burst buffers, it is necessary to evaluate the overhead of metadata service in CNs to make sure that it will not impact the performance. We distributed the metadata services across all the compute nodes used by the application, so that the workload of metadata querying on each compute node is not too heavy. Besides, we have adopted metadata caching mechanism to minimize the overhead of querying metadata remotely, which significantly reduces the number of metadata queries in metadata servers. According to our observation in the tests of WatCache, the CPU usage of metadata service process seldom exceeds 1%. In the worst case that all processes in different compute nodes access the same file, and all metadata queries are sent to only one metadata server, the CPU usage of metadata service process on the single metadata server is still within 3%.

We use radix trees to record the metadata of files in the metadata server. The required storage space of a radix tree grows cumulatively with the data range it holds. The memory usage of metadata server is mainly determined by the number of data blocks it records. We use 1 MB block in WatCache, and the metadata size of each data block is 16 bytes, including 8 bytes of block ID and 8 bytes of data location. When the metadata server records the metadata of 1 TB data, it will use 16 MB of memory. With 24 GB memory in each compute node, the memory used by metadata server is almost negligible.

Therefore, the overhead of metadata service has little impact to application performance.

6.4 Small I/O

As illustrated in Sect. 3.2, small I/O is a common I/O pattern in HPC workloads. However, common parallel file systems, e.g., Lustre, are designed to cope with large consecutive I/O requests rather than small I/O. Small I/O can cause significant performance impact if not dealt with properly. We conduct a test on the small I/O performance of McCache, KvCache and WatCache, and compare them with the performance of Lustre. We use only one IOR process, accessing a 128 MB file stored in Lustre with 4 KB I/O size. Two different I/O patterns, sequential I/O and random I/O, are both tested.

Figure 11a shows the write bandwidth of sequential I/O and random I/O. The write performance of WatCache is 7.1× higher than Lustre under sequential I/O and 5.6× higher under random I/O. The result is expected since write I/O requests in WatCache are buffered in local fast storage immediately. However, the bandwidths of McCache and KvCache are only about 25% of Lustre, which implies that, attaching a fast storage tier to HPC system with these two architectures gets worse performance under small I/O. The reason lies in the costly overhead of retrieving metadata remotely in KvCache or retrieving data remotely in McCache. Network latency is the major cause of poorer performance than Lustre. In WatCache, processes cache the block metadata locally so that any small I/O request fall into this block range can benefit from the cached metadata. The average metadata overhead of WatCache is much lower than KvCache.

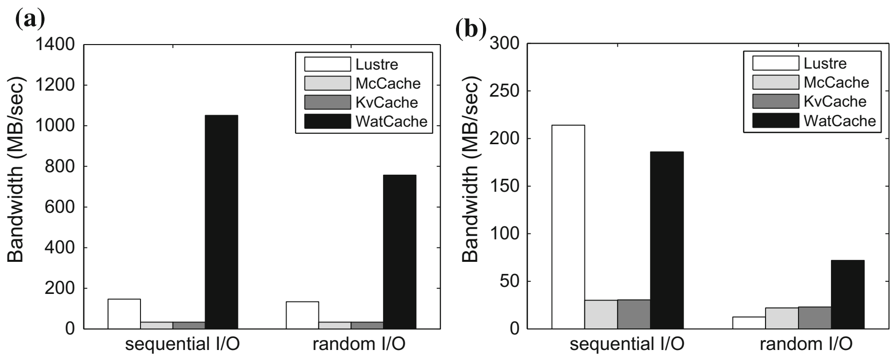

Fig. 11 The small I/O performance of Lustre, McCache, KvCache and WatCache. **a** Write performance of small I/O. **b** Read performance of small I/O

Figure 11b shows the read bandwidth of sequential I/O and random I/O. Since all the accessed data need to be read from Lustre, and WatCache introduces extra overhead of writing data into local storage to cache them, the read bandwidth of WatCache is 13% lower than Lustre under sequential I/O. As for random I/O, the random pattern disturbs the prefetching in Lustre, so that most I/O requests miss the page cache and are therefore issued to storage servers. The read bandwidth of Lustre dramatically drops to about 10 MB/s. Unlike Lustre, WatCache read in a whole block when handling read requests smaller than block size, that is why its random read performance is about 6× higher than Lustre. This strategy is not suitable for Lustre because Lustre client prevents reading in too much unwanted data and therefore causing unnecessary latency. However, WatCache is a distributed cache system whose cached data are visible to all processes in the cluster. The possibility of accessing the rest data of the block in WatCache is much higher than that in a single Lustre client. Therefore, the extra latency of reading the whole block is worthwhile.

6.5 Rank 0 I/O and all-rank I/O

In this section we conducted a test to evaluate the performance of WatCache under rank 0 I/O and all-rank I/O. We first let an IOR process write a 8 GB file, then let 64 IOR processes in 64 CNs read the file back. The write and read bandwidths are shown in Fig. 12.

The write bandwidths of McCache, KvCache and WatCache are about 4.1×, 7.1×, 8.4× better than Lustre respectively. The performance improvements are brought by buffering their written out data in the node-local fast storage tier. The bandwidth of McCache is lower than KvCache because its data are transferred to remote nodes. The bandwidth of WatCache is better than KvCache. The reason is that the written file is large enough to trigger LASIOD in WatCache; thus, only a portion of the data is written to local storage, and other portions of data are transferred to multiple remote nodes. Although writing data to remote RAM disks are slower than writing locally,

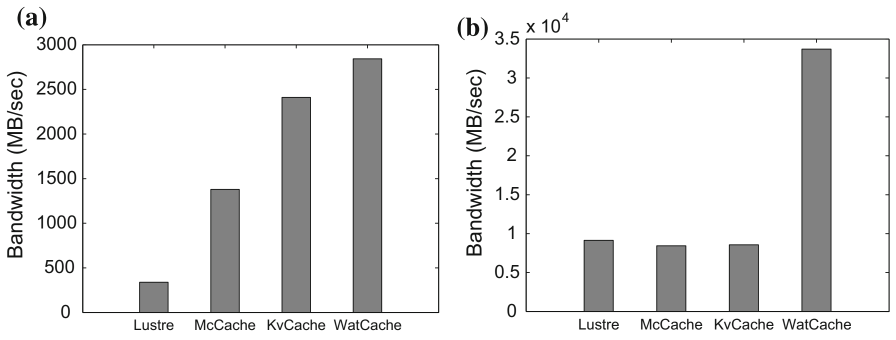

Fig. 12 Evaluate the performance of Lustre, McCache, KvCache and WatCache under rank 0 I/O and all-rank I/O patterns. **a** The performance of writing a file with an IOR process. **b** The performance of reading the file back with 64 IOR processes

the aggregate bandwidth is better since data are written concurrently to multiple RAM disks.

As shown in Fig. 12b, when reading the file back with 64 processes, WatCache has the highest bandwidth. The bandwidth of Lustre has reached the peak, which is about 9 GB/s. In McCache, the location of file data is determined with the hash value of the file path; thus, the file is stored in a single remote node. In KvCache, the file is stored in a single local device. Thus, the bandwidths of McCache and KvCache are about the same, which is the peak bandwidth of a RAM disk when accessed by multiple remote processes. However, in WatCache, the large file is striped over multiple nodes with LASIOD. Its read bandwidth is promoted significantly due to parallel accessing.

6.6 BTIO

BTIO, a part of NASA Advanced Supercomputing (NAS) parallel benchmark suit, is a I/O kernel widely used to evaluate the performance of storage systems. It is based on a computational fluid dynamics (CFD) application that solves 3D Navier–Stokes equations. BTIO partitions a 3D array across a square number of processes, on each of which runs a block-tridiagonal (BT) solver. Since the data accessed by each process are a subset of a global 3D array that stored in row-major order, it generates large amount of small noncontiguous I/O during execution.

Usually collective I/O is used to aggregate these small I/O requests into larger ones to achieve better I/O performance. However, most HPC applications do not use MPI I/O to access data [29], so that they have no chance to be optimized by collective I/O method provided by MPI I/O. It is essential to evaluate the WatCache's performance of serving original small I/O. Therefore, we evaluate both independent I/O method and collective I/O method in this test.

Figure 13a shows the test results of independent I/O. The performance of Lustre is extremely low (less than 10 MB/s) when the small I/O requests of BTIO are directly delivered to OSTs. Increasing the number of CNs cannot promote the performance since it is Lustre that causes the bottleneck. After attaching a fast storage tier to the

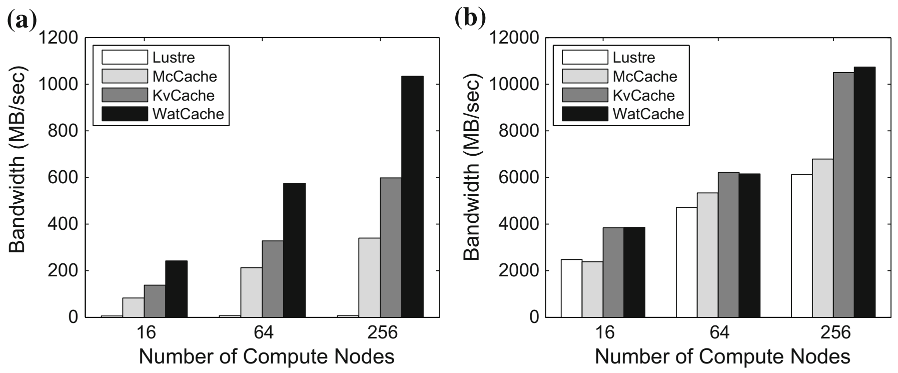

Fig. 13 Performance of BTIO on Lustre, McCache, KvCache and WatCache with independent and collective I/O methods. **a** Independent I/O. **b** Collective I/O

CN side, the performance is much better. The RAM disks are better at serving small, noncontiguous I/O than hard disk-based OSTs. As the number of CNs increases, the performance of three cache systems grows continually since more RAM disks are utilized. The performance of WatCache is better than McCache, since most of its data can be accessed by processes in local storage. Its performance is better than KvCache, too. Because its metadata caching strategy avoids lots of unnecessary remote metadata queries.

In the test presented in Fig. 13b, BTIO uses collective I/O to access the large dataset. A certain number of processes, which is often set to the number of OSTs, are chosen to act as aggregators that are responsible for performing all the I/O. Small I/O requests are sent to the aggregators first; then, the aggregators pack them into larger requests, which minimizes the total communications with Lustre and avoids lots of noncontiguous I/O. As shown in Fig. 13b, the performance is much better than independent I/O method. As small I/O requests are aggregated to larger requests, the metadata overhead difference between KvCache and WatCache is not so significant, that their bandwidths are about the same.

6.7 WRF workflow

Weather research and forecasting (WRF) is a numeric weather prediction model which has been widely used for both research and operational weather forecast purpose. A typical weather simulation at grid resolution less than 2 kilometers covering the entire globe would generate terabytes of data in a single forecast cycle [2]. Its I/O can be very inefficient when all data are gathered and accessed by rank 0 process. PnetCDF library is used in large-scale WRF simulation to support parallel I/O. When PnetCDF is enabled, MPI ranks are aggregated into groups, whose number is usually set to the stripe count of the output file, and then one aggregator in each group performs the I/O. WRF workflow is a typical job running in Tianhe-1A. The workflow consists of thousands of cycles, each of which contains two phases. In the first WRF phase, two different datasets are processed and simulated by WRF application in succession. Then

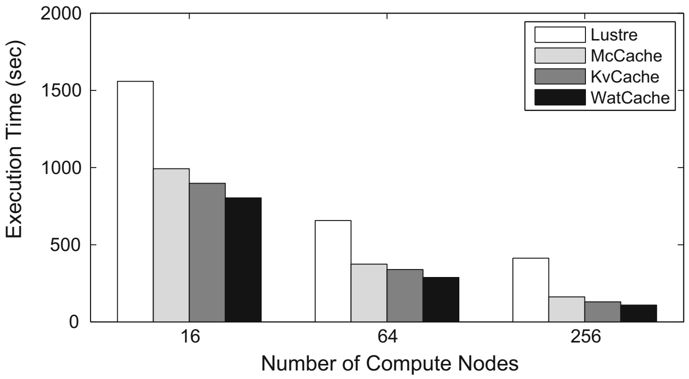

Fig. 14 The running time of WRF workflow on Lustre, McCache, KvCache and WatCache

in the second exchange phase, parts of the two output datasets from WRF phase are exchanged by an exchange application. In the next cycle, the two exchanged datasets are taken as input data of WRF application again.

We evaluate the performance of running WRF workflow in Lustre, McCache, KvCache and WatCache. The sizes of the two initial datasets are both 27 GB. In each forecast cycle of WRF application, two datasets are simulated for two time steps. The size of output datasets are 27 GB, too. The amount of data exchanged between two datasets is about 9 GB. In the test we run the workflow for 10 cycles and use PnetCDF library to perform I/O. The test results of their execution time are displayed in Fig. 14.

As shown in Fig. 14, the execution time of Lustre is much longer than the three cache systems. The output data written out by a node in WRF phase may be read in by another node for exchange purpose, and the data written out by the node in exchange phase may be read in by a third node in the WRF phase of the next forecast cycle. Since data are accessed by different nodes between phases, the data cached in the Lustre client cache have to be invalidated, and the requested data have to be accessed form Lustre OSTs. By caching the written out and exchanged data in node-local storage tier, the performance is boosted significantly, as revealed in the performance of three cache systems. All the data are visible to all processes in the different phases of the workflow and can be accessed much more efficient than data in OSTs. Above results reveal that the node-local fast storage tier is optimal for workflows since it caches massive amount of intermediate data. Moreover, WatCache has the best performance among the three cache systems, owing to its optimization of metadata caching and data layout.

6.8 Performance with different proportion of fast storage

In order to compare the performance of applications running on different proportion of fast storage, we test them with node allocation hints of `low`, `medium` and `high` respectively, which means that 0, 50 or 100% of their allocated CNs are attached with tmpfs. Four applications are tested, including WRF workflow, BTIO with independent

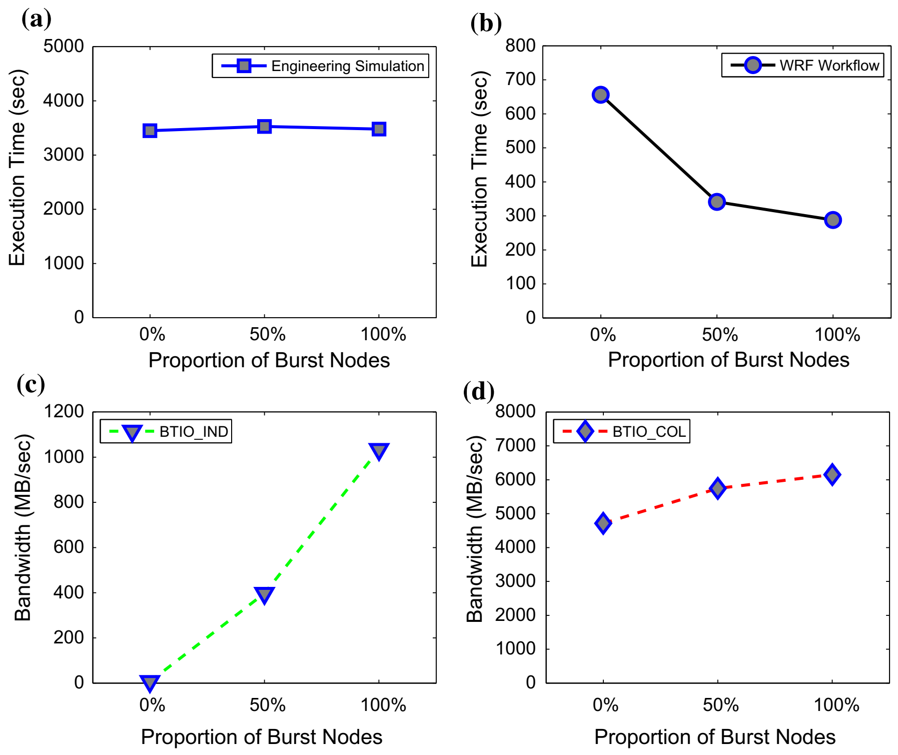

Fig. 15 Performance of 4 applications with different proportion of fast storage. **a** Engineering simulation. **b** WRF work ow. **c** BTIO with independent I/O. **d** BTIO with collective I/O

I/O method, BTIO with collective I/O method and an engineering simulation application that only reads and writes a small amount of data at the beginning and end of its execution. The experiments are conducted on 64 CNs, whose results are shown in Fig. 15.

As shown in the results, the performance of engineering simulation application is hardly affected by the proportion of its allocated fast storage, which is expected since the applications are mainly compute intensive. There are no benefits to allocate fast storage to this kind of applications. The performance of WRF workflow is greatly boosted by WatCache. Specifically, the performance of WRF workflow with 50% burst nodes is about 2$\times$ better than that with WatCache disabled. The performance increments are brought by caching large amount of intermediate data in a storage tier closer to CNs. The trend of performance increment slows down with the increase in the proportion of burst nodes. Even though, its performance with full support of WatCache is about 1.2 $\times$ faster than half support. Another application accelerated by WatCache significantly is BTIO with independent I/O method, whose bandwidth increases linearly with the number of burst nodes. There is still room for bandwidth growth when running it with 50% burst nodes, since there still are large amount of small I/O that are served in remote nodes, which introduces high network overhead. With more burst nodes utilized, more small I/O requests can be satisfied in local storage, which results in higher bandwidth. The bandwidth of BTIO with collective I/O method is boosted not so significantly since Lustre is expert in handling large, consecutive I/O. However, the bandwidth of Lustre is shared by a lot of applications and will reach its peak under higher concurrency. On the contrary, the bandwidth of WatCache is exclusively available to just one application and is linearly scalable. Thus, the bandwidth of BTIO with collective I/O method will be significantly better than Lustre when running on a larger scale.

To summarize, WatCache cannot offer benefits to compute-intensive applications and it is a waste to allocate burst nodes to them. The performance improvements are limited for applications which have been already optimized by MPI collective I/O. WatCache is suitable for caching large amount of intermediate data, handling small, noncontiguous I/O and serving extreme scale data-intensive applications. In order to achieve optimal cost-effectiveness, applications with large amount of intermediate data only need 50% burst nodes in their allocated CNs, and applications which issue large amount of small I/O have to acquire full WatCache support.

7 Conclusion and future work

As node-local burst buffers have been included in the blueprints of many next-generation HPC systems, in this paper, we proposed a workload-aware temporary cache system called WatCache to manage the new storage tier in HPC I/O hierarchy. WatCache merges the fast storage devices on the compute nodes that allocated to each user job to standalone cache pools. The cache pools are temporary since they are established along with the user jobs and will be recycled when jobs end. We presented a workload-aware node allocation method to make sure that the compute nodes with fast storage devices are allocated to jobs that require the most I/O support. Several techniques are implemented to further enhance the performance of WatCache, including a data layout strategy that improves the performance of parallel accessing, and a metadata caching mechanism that minimizes the metadata overhead. WatCache is tested through a series of I/O benchmarks and applications on hundreds of compute nodes, which clearly indicates that it brings significant performance improvements, especially on small I/O, as well as rank 0 and all-rank I/O. With high performance and flexibility, BrustCache is likely to be a practical solution to manage node-local burst buffers.

In the future work, we plan to employ mechanisms to predict the upcoming I/O workloads in HPC systems according the history submissions [5,30], so that the limited burst nodes can be utilized more efficiently. We also plan to adopt prefetching mechanism [3] to fully exploit the potential of multi-level storage hierarchy.

Acknowledgements We thank Jian Zhang and Fuxing Sun at NSCC-TJ for their generous help of setting up experimental environments on Tianhe-1A.

References

1. Aurora. http://aurora.alcf.anl.gov/
2. Balle T, Johnsen P Improving i/o performance of the weather research and forecast (wrf) model

3. Banu JS, Babu MR (2015) Exploring vectorization and prefetching techniques on scientific kernels and inferring the cache performance metrics. Int J Grid High Perform Comput 7(2):18–36
4. Bharathi S, Chervenak A, Deelman E, Mehta G, Su MH, Vahi K (2008) Characterization of scientific workflows. In: 2008 Third Workshop on Workflows in Support of Large-Scale Science. IEEE, pp 1–10
5. Brito JBDS (2016) Hcem model and a comparative workload analysis of hadoop cluster. Int J Big Data Intell 4(1):47
6. Burst buffer architecture and software roadmap. http://www.nersc.gov/users/computational-systems/cori/burst-buffer/burst-buffer/
7. Byan S, Lentini J, Madan A, Pabon L, Condict M, Kimmel J, Kleiman S, Small C, Storer M (2012) Mercury: host-side flash caching for the data center. In: IEEE Symposium on Mass Storage Systems and Technologies. https://doi.org/10.1109/MSST.2012.6232368
8. Catalyst. http://computation.llnl.gov/computers/catalyst
9. Congiu G, Narasimhamurthy S, Süß T, Brinkmann A (2016) Improving collective i/o performance using non-volatile memory devices. In: 2016 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, pp 120–129
10. Cray datawarp applications i/o accelerator. http://www.cray.com/products/storage/datawarp
11. Dahlin MD, Wang RY, Anderson TE, Patterson DA (1994) Cooperative caching: using remote client memory to improve file system performance. In: Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation. USENIX Association, p 19
12. Darshan. https://xgitlab.cels.anl.gov/darshan/darshan. Accessed August 13, 2016
13. Dong X, Xie Y, Muralimanohar N, Jouppi NP (2011) Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. ACM Trans Archit Code Optim 8(2):6
14. Dong W, Liu G, Yu J, Hu W, Liu X (2015) Sfdc: File access pattern aware cache framework for high-performance computer. In: High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conference on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on, pp 342–350
15. Fitzpatrick B (2004) Distributed caching with memcached. Linux J 2004(124):5
16. Gluster file system. http://www.gluster.org
17. Greenberg HN, Bent J, Grider G (2015) Mdhim: a parallel key/value framework for hpc. In: Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems, HotStorage'15. USENIX Association, Berkeley, CA, USA, pp 10–10. http://dl.acm.org/citation.cfm?id=2813749.2813759
18. Gunasekaran R, Oral S, Hill J, Miller R, Wang F, Leverman D (2015) Comparative i/o workload characterization of two leadership class storage clusters. In: Proceedings of the 10th Parallel Data Storage Workshop. ACM, pp 31–36
19. Holland DA, Angelino E, Wald G, Seltzer MI (2013) Flash caching on the storage client. In: Presented as Part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13). USENIX, San Jose, CA, pp 127–138. https://www.usenix.org/conference/atc13/technical-sessions/presentation/holland
20. Infinite memory engine. http://www.ddn.com/products/infinite-memory-engine-ime14k/
21. Jung M, Wilson III EH, Choi W, Shalf J, Aktulga HM, Yang C, Saule E, Catalyurek UV, Kandemir M(2013) Exploring the future of out-of-core computing with compute-local non-volatile memory. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. ACM, p 75
22. Koziol Q et al (2014) High performance parallel I/O. CRC Press, Boca Raton
23. Li X, Xiao L, Ke X, Dong B, Li R, Liu D (2014) Towards hybrid client-side cache management in network-based file systems. Comput Sci Inf Syst 11(1):271–289
24. Liao WK, Ching A, Coloma K, Nisar A, Choudhary A, Chen J, Sankaran Ry, Klasky S (2007) Using mpi file caching to improve parallel write performance for large-scale scientific applications. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing. ACM, p 8
25. Liao Wk, Coloma K, Choudhary A, Ward L, Russell E, Tideman S (2005) Collective caching: application-aware client-side file caching. In: High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE International Symposium on. IEEE, pp 81–90
26. Liu N, Cope J, Carns P, Carothers C, Ross R, Grider G, Crume A, Maltzahn C (2012) On the role of burst buffers in leadership-class storage systems. In: 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST). IEEE, pp 1–11

27. Liu X, Lu Y, Lu Y, Wu C, Wu J (2016) masfs: File system based on memory and ssd in compute nodes for high performance computers. In: 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS). IEEE, pp 569–576
28. Lofstead J, Polte M, Gibson G, Klasky S, Schwan K, Oldfield R, Wolf M, Liu Q (2011) Six degrees of scientific data: reading patterns for extreme scale science io. In: Proceedings of the 20th International Symposium on High Performance Distributed Computing. ACM, pp 49–60
29. Luu H, Winslett M, Gropp W, Ross R, Carns P, Harms K, Prabhat M, Byna S, Yao Y (2015) A multi-platform study of i/o behavior on petascale supercomputers. In: Proceedings of the 24th International Symposium on High-performance Parallel and Distributed Computing. ACM, pp 33–44
30. Mao L, Qi D, Lin W, Zhu C (2015) A self-adaptive prediction algorithm for cloud workloads. Int J Grid High Perform Comput 7(2):65–76
31. Mittal S, Vetter JS (2016) A survey of software techniques for using non-volatile memories for storage and main memory systems. IEEE Trans Parallel Distrib Syst 27(5):1537–1550
32. Nvm express. http://www.nvmexpress.org/
33. Ovsyannikov A, Romanus M, Van Straalen B, Weber GH, Trebotich D (2016) Scientific workflows at datawarp-speed: accelerated data-intensive science using nerscs burst buffer. In: Proceedings of the 1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems. IEEE Press, pp 1–6
34. Performance and debugging tools: Darshan. http://www.nersc.gov/users/software/performance-and-debugging-tools/darshan/. Accessed August 13, 2016
35. Pollak B. Portable operating system interface (posix)-part 1x: real-time distributed systems communication application program interface (api). IEEE Standard P 1003
36. Rew R, Davis G (1990) Netcdf: an interface for scientific data access. IEEE Comput Graph Appl 10(4):76–82
37. Romanus M, Ross RB, Parashar M (2015) Challenges and considerations for utilizing burst buffers in high-performance computing. arXiv preprint arXiv:1509.05492
38. Samsung enterprise ssd. http://www.samsung.com/semiconductor/products/flash-storage/enterprise-ssd/
39. Schenck W, El Sayed S, Foszczynski M, Homberg W, Pleiter D (2017) Evaluation and performance modeling of a burst buffer solution. ACM SIGOPS Oper Syst Rev 50(1):12–26
40. Schlagkamp S, Ferreira da Silva R, Allcock W, Deelman E, Schwiegelshohn U (2016) Consecutive job submission behavior at mira supercomputer. In: Proceedings of the 25th ACM International Symposium on High-performance Parallel and Distributed Computing. ACM, pp 93–96
41. Sharedhashfile. https://github.com/simonhf/sharedhashfile
42. Shibata T, Choi S, Taura K (2010) File-access patterns of data-intensive workflow applications and their implications to distributed filesystems. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. ACM, pp 746–755
43. Sierra. https://www.llnl.gov/news/next-generation-supercomputer-coming-lab
44. Summit fact sheet. https://www.olcf.ornl.gov/summit/
45. Top500 supercomputer sites. http://www.top500.org
46. Trinity. http://www.lanl.gov/projects/trinity/
47. Vetter JS, Mittal S (2015) Opportunities for nonvolatile memory systems in extreme-scale high-performance computing. Comput Sci Eng 17(2):73–82
48. W Hu, Liu Gm, Li Q, Jiang Yh (2016) Storage wall for exascale supercomputing. J Zhejiang Univ Sci 2016:10–25
49. Wang T, Mohror K, Moody A, Sato K, Yu W (2016) An ephemeral burst-buffer file system for scientific applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Press, p 69
50. Wang T, Oral S, Pritchard M, Wang B, Yu W (2015) Trio: burst buffer based i/o orchestration. In: 2015 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, pp 194–203
51. Wang F, Oral S, Shipman G, Drokin O, Wang T, Huang I (2009) Understanding lustre filesystem internals. Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep
52. Wang T, Oral S, Wang Y, Settlemyer B, Atchley S, Yu W (2014) Burstmem: a high-performance burst buffer system for scientific applications. In: 2014 IEEE International Conference on Big Data (Big Data). IEEE, pp 71–79

53. Weil SA, Brandt SA, Miller EL, Long DD, Maltzahn C (2006) Ceph: a scalable, high-performance distributed file system. In: Proceedings of the 7th Symposium on Operating Systems Design And Implementation. USENIX Association, pp 307–320
54. Xie M, Lu Y, Liu L, Cao H, Yang X (2011) Implementation and evaluation of network interface and message passing services for tianhe-1a supercomputer. In: 2011 IEEE 19th Annual Symposium on High Performance Interconnects. IEEE, pp 78–86
55. Xu W, Lu Y, Li Q, Zhou E, Song Z, Dong Y, Zhang W, Wei D, Zhang X, Chen H et al (2014) Hybrid hierarchy storage system in milkyway-2 supercomputer. Front Comput Sci 8(3):367–377
56. Yoo AB, Jette MA, Grondona M (2003) Slurm: simple linux utility for resource management. In: Workshop on Job Scheduling Strategies for Parallel Processing. Springer, pp 44–60
57. Zhao D, Zhang Z, Zhou X, Li T, Wang K, Kimpe D, Carns P, Ross R, Raicu I (2014) Fusionfs: toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In: 2014 IEEE International Conference on Big Data (Big Data). IEEE, pp 61–70