CARS: A contention-aware scheduler for efficient resource management of HPC storage systems

Weihao Liang\textsuperscript{a}, Yong Chen\textsuperscript{b}, Jialin Liu\textsuperscript{c}, Hong An\textsuperscript{a}

\textsuperscript{a}Department of Computer Science, University of Science and Technology of China, Hefei, China
\textsuperscript{b}Department of Computer Science, Texas Tech University, Lubbock, TX, USA
\textsuperscript{c}Lawrence Berkeley National Laboratory, Berkeley, CA, USA

\textbf{A B S T R A C T}

Many scientific applications are becoming more and more data intensive. As the data volume continues to grow, the data movement between storage and compute nodes has turned into a crucial performance bottleneck for many data-intensive applications. Burst buffer provides a promising solution for these applications by absorbing bursty I/O traffic. However, the resource allocation and management strategies for burst buffer are not well studied. The existing bandwidth based strategies may cause severe I/O contention when a large number of I/O-intensive jobs access the burst buffer system concurrently. In this study, we present a contention-aware resource scheduling (CARS) strategy to manage the burst buffer resources and coordinate concurrent data-intensive jobs. The experimental results show that the proposed CARS framework outperforms the existing allocation strategies and improves both the job performance and the system utilization.

\textsuperscript{©} 2019 Elsevier B.V. All rights reserved.

\textbf{1. Introduction}

More and more scientific applications in critical areas, such as climate science, astrophysics, combustion science, computational biology, and high-energy physics, tend to be highly data intensive and present significant data challenges to the research and development community \cite{x, y}. Large amounts of data are generated from scientific experiments, observations, and simulations. The size of these datasets can range from hundreds of gigabytes to hundreds of petabytes or even beyond. For instance, the FAST project (500-meter Aperture Spherical Telescope) \cite{z} in Australia uses the Next Generation Archive System (NGAS) to store and maintain a large amount of data which is to be collected. NGAS expects to handle about 3 petabytes of data from FAST every year, enough to fill 12,000 single-layer, 250 gigabyte Blu-ray disks. There are a number of reasons for this big data revolution: a rapid growth in computing capability (especially when compared with a much slower increase in I/O system bandwidth) has made data acquisition and generation much easier; high-resolution, multi-model scientific discovery will require and produce much more data; and the needs that insights can be mined out of large amounts of low-entropy data have substantially increased over years.

Meantime, with the increasing performance gap between the computing and I/O capability, data access has become the performance bottleneck of many data-intensive applications, usually containing a large number of I/O operations and large amounts of data are written to and read from the storage system. The newly emerged burst buffer \cite{a} is a promising solution to address this bottleneck issue. Given that the I/O patterns of many scientific applications are bursty \cite{b}, burst buffer is designed as a layer of hierarchical storage, which comprises of high-speed storage medium such as Solid State Drives (SSD), to let the applications write or read data quickly and return to the computation phase as soon as possible. The data temporarily stored in burst buffer can then be transferred to the remote storage system asynchronously without interrupting the applications. In recent years, several peta-scale HPC systems have been deployed with a burst buffer sub-system, such as Cori supercomputer \cite{c} at National Energy Research Scientific Computing Center (NERSC) and Trinity \cite{d} at Los Alamos National Laboratory (LANL). A number of large-scale data-intensive scientific applications have harnessed significant performance improvement from these burst buffer systems \cite{e}.

These burst buffer systems are designed as shared resources for hundreds or thousands of users and applications. Previous research efforts have mainly focused on studying how to leverage burst buffer to improve the applications performance by reducing the
I/O time [9–11] and overlapping between the computation and I/O phases of applications [12,13]. However, the resource management for burst buffer is still understudied and the existing scheduling strategies only consider the capacity request of users, which may cause I/O contention between multiple or many concurrent I/O intensive applications.

In this paper, we propose a contention-aware resource scheduling (CARS) design to reduce the I/O contention for multiple concurrent I/O intensive jobs and improve the performance of the burst buffer system. Following the study of different resource scheduling policies for shared burst buffer system and I/O modeling, CARS is designed as a new resource allocation and scheduling strategy for burst buffer. CARS characterizes the I/O activities of running applications on burst buffer system, and allocates the burst buffer resource for new jobs based on the analysis of the I/O load on the burst buffer system. Compared to the existing allocation strategies that only consider the capacity factor, CARS can significantly mitigate the I/O contention between data-intensive jobs, largely speeding up the I/O while improving the burst buffer system utilization.

The rest of the paper is organized as follows. Section 2 discusses the background and motivation of this research. Section 3 and Section 4 present the design of the contention-aware resource scheduling (CARS) and the I/O modeling respectively. The experimental and analytic results are presented in Section 5. Section 6 discusses the related work and compares this study with them, and Section 7 concludes this study and discusses future work.

2. Background and motivation

In this section, we first introduce the background about the burst buffer architecture and existing resource allocation policies of shared burst buffer systems. Then we illustrate the motivation of this research study and discuss the challenges in our work.

2.1. Overview of burst buffer

Fig. 1 illustrates an architecture overview of a shared burst buffer system. The burst buffer resides on dedicated nodes that are usually deployed with SSDs or storage-class memory (SCM) in general as a high-speed storage tier. As a shared resource, the burst buffer nodes are available for all compute nodes to access directly. While applications from compute nodes issue bursty I/O operations, the burst buffer nodes can quickly absorb I/O requests in their local SSDs/SCM, and let the applications continue to the next computing phase as soon as possible. Afterward, the burst buffer nodes will transfer the data stored in their local SSDs/SCM to a parallel file system (PFS) asynchronously. Existing large-scale HPC systems such as Cori [6] and Trinity [7] have deployed such shared burst buffer system architecture as production systems. Another alternative burst buffer architecture is that each compute node contains NVMe-device as high-speed local storage, as implemented on Summit [14]. Such an architecture can effectively reduce I/O time by allowing checkpoint files to be written locally and then flushed to external PFS asynchronously with specific hardware support [15]. This design is particularly suitable for applications that perform N-N (i.e. file per process) I/O operations. In this study, we focus on the shared burst buffer architecture and the optimization of data movement between compute nodes and the shared burst buffer nodes.

2.2. Resource management for burst buffer

Burst buffer serves the entire compute system as a shared storage layer. Thus, it is crucial for resource scheduler to manage the burst buffer nodes efficiently to be accessed by different users and applications. In the burst buffer system of Cori or Trinity supercomputer, the DataWarp [16] software integrated with SLURM [17] workload manager is responsible for the resource management. When a user submits jobs by indicating the options for burst buffer requirement in the SLURM batch script, the resource scheduler will allocate burst buffer nodes to meet users’ capacity need. The capacity request from users will be split up into multiple chunks and one chunk size is the minimum allocation size on one burst buffer node. For example, there are two choices of the chunk size on the Cori system, 20GB and 80GB, respectively.

Two allocation policies exist for how to assign burst buffer nodes for jobs on Cori, and users can specify the optimization_strategy option manually in SLURM batch script to choose different policies [16]. The first one is bandwidth policy, which instructs the resource scheduler to select as many burst buffer nodes as possible in a round-robin fashion to maximize the bandwidth for a job. In this case, users data will be striped across several burst buffer nodes and one node contains one chunk of the requested capacity size (i.e. 20GB or 80GB). So each node can be shared by several jobs under the bandwidth policy. The second one is interference policy, in which the system will assign as few burst buffer nodes as possible to minimize interference from other jobs. In this case, a burst buffer node will comprise as many chunks as possible and be occupied by only one job at a time. For example, assume there are four burst buffer nodes and three jobs are submitted in order. Each job requests the capacity of two chunks, e.g., 40GB for two 20GB chunks. Fig. 2 illustrates how burst buffer nodes will be assigned to these three jobs under these two policies, respectively. With the bandwidth policy, Job 1 is assigned to BB1 and BB2, Job 2 is assigned to BB3 and BB4, and Job 3 is assigned to BB1 and BB2 again in a round robin fashion. Every chunk of each job is stored in one burst buffer node. In this case, Job 1 and Job 2 will share the same burst buffer nodes. As for the interference policy, all chunks of each job will be allocated on one burst buffer node and each job occupies one burst buffer node exclusively. In other words, with the interference policy, burst buffer nodes will not be shared by multiple jobs. The bottom part of Fig. 2 demonstrates such policy. If we assume another two jobs, Job...
4 and Job 5, requests two chunks too, then BB4 will be allocated for one job, and the last job would be blocked till burst buffer node becomes available (i.e. completion of one job and release of one BB node).

2.3. Motivation and challenges

While the existing bandwidth and interference scheduling policies serve the purpose to some extent, they are rudimentary and underdeveloped to well leverage the valuable burst buffer resources. Particularly, the interference policy is designed to minimize the contention from multiple jobs, but sacrifices the utilization of burst buffer nodes due to an “exclusive” allocation policy. On the other hand, the bandwidth policy is designed to better utilize the burst buffer resources but do not consider the contention from highly concurrent jobs, which is the norm on large-scale HPC systems, especially for projected next-generation, exascale systems.

We take the example in Fig. 2 again to discuss the problem. It is easy to understand the limitation of the interference policy, as such policy prevents resource sharing among jobs on any one burst buffer node. Thus the interference policy will be limited by the scale of the burst buffer system, i.e. the number of burst buffer nodes available to jobs, instead of the total capacity available, or the total bandwidth available. Such scheduling policy can easily lead to under-utilization of resources, including the under-utilization of both bandwidth and capacity.

The bandwidth policy, however, takes the other extreme. It only focuses on resource sharing but ignores the critical contention issue. Consider the example in Fig. 2. Assume Job 1 has many I/O requests, keeps BB1 and BB2 busy and saturates the peak bandwidth of these two burst buffer nodes, whereas Job 2 may have light I/O load and the bandwidth of BB3 and BB4 is not fully utilized at all. When a new job (Job 3) arrives, the bandwidth policy will assign Job 3 to BB1 and BB2, in a round-robin fashion, as long as they have enough available space to meet the capacity request of Job 3. In such situation, Job 3 would have to compete for the limited available bandwidth of BB1 and BB2 with Job 1. Such a scenario creates a critical I/O contention issue on BB1 and BB2. On one hand, the performance of Job 1 will be significantly affected by sharing the limited bandwidth with Job 3. On the other hand, the light I/O load of Job 2 makes BB3 and BB4 underutilized. This scenario creates a critical performance imbalance issue too among different burst buffer nodes and largely limits the overall utilization of the entire burst buffer system. As a comparison, the interference policy can lead to a scenario that, even though there is no any contention from other jobs, one burst buffer node is only allocated to one job exclusively, largely restricting the utilization of burst buffer and more importantly, only delivering limited bandwidth to each job.

If there is a way to determine that there may be contention between Job1 and Job3, if they are allocated on same burst buffer nodes, then Job3 can be scheduled to BB3 and BB4 instead, which lead to higher utilization of BB3 and BB4 and also avoid the performance degradation of Job 1. Such a hypothesis and observation inspire our research motivation in this study that the burst buffer resource scheduling strategies should well consider the contention issue, while maximizing the benefits of sharing resources among jobs to best leverage the bandwidth and capacity available from burst buffer nodes.

From the previous discussion and the motivating examples, we have seen the potential of a new burst buffer resource scheduling strategy, which should meet two conflicting goals: maximizing resource utilization while minimizing the contention. Clearly there are challenges and we need to identify them to address them in designing a new burst buffer scheduling strategy. These challenges are three-fold, as discussed in detail below.

First, how to define and detect I/O contention? In general, I/O contention occurs when there are more than one job accessing the same burst buffer node. To understand the relationship between I/O load (i.e., the number of concurrent I/O processes) and performance, we used IOR [18] to conduct a series of tests on the Cori’s burst buffer system. The total data volume tested was 1TB. Each I/O request size was 2MB. The number of total processes varied from 1 to 16. All these processes were simultaneously accessing one burst buffer node. Fig. 3 shows the experimental results. It can be observed that the I/O throughput increased first and then reached the peak bandwidth with about 12 processes. The reason for the throughput increase is that jobs with few I/O processes cannot make burst buffer nodes fully utilized. After keeping increasing the number of processes, the I/O throughput began to saturate or drop slightly. The reason for the throughput degradation was caused by the I/O contention from too many I/O processes concurrently accessing the same burst buffer node. In this case, if more jobs or

![](image-url)
processes are scheduled to this burst buffer node, it will make the contention even worse. In Section 4, we present a quantitative analysis through a theoretical analysis of the I/O model in burst buffer systems, with parameters to describe the I/O activities of different jobs and how they lead to I/O contention.

Second, how to prevent or reduce I/O contention? As observed from the results in Fig. 3, too many concurrent I/O processes or jobs sharing same burst buffer nodes will make these nodes reach their peak bandwidth quickly and the overall performance can be degraded, especially if we keep assigning more jobs to access these burst buffer nodes. Intuitively, the basic idea to prevent or reduce I/O contention is not to make some burst buffer nodes overloaded. One straightforward way is to limit the number of running jobs or I/O processes on each burst buffer node. However, one main drawback of this method is that the threshold (i.e., the maximum number of concurrent jobs or processes) is difficult to be determined since the I/O behavior (e.g., transfer size of I/O request) can vary significantly among different applications. Therefore, we decide to exploit the idea of using a finer granularity approach, which will be described in Section 3 in details.

Third, how to measure the performance of jobs and system? For each individual job, its I/O throughput may be much lower when it shares the burst buffer nodes with other jobs. Different resource scheduling policies will determine how jobs are distributed among the burst buffer nodes and impact each job’s performance. Thus we need a uniform standard to measure the job performance under different resource allocation policies. As for the system performance, it is not practical to apply the system utilization concept of the compute system to the burst buffer system directly. To address this challenge, we present several performance metrics to measure the job performance and system utilization, from the perspective of users and system respectively, in Section 4.

3. Methodology

This section describes the proposed contention-aware resource scheduling solution for burst buffer systems including the framework and the scheduling algorithm.

3.1. Framework

3.1.1. Design overview

In order to manage burst buffer resource efficiently and to coordinate a large number of jobs from different users, we introduce a contention-aware resource scheduling (CARS) algorithm [19] for optimizing application and system performance. CARS is designed as a runtime middleware which is placed between applications and the underlying burst buffer as shown in Fig. 4. As a middleware layer, CARS dynamically captures the information of any incoming I/O-intensive jobs and monitors the status of the burst buffer system. Based on the data collected by the CARS over time, the burst buffer resources will be scheduled for each new incoming job. The CARS algorithm and methodology are designed to work with the existing HPC job scheduler (e.g., SLURM). Our objective is to improve both the applications’ performance and the system’s utilization by assigning appropriate burst buffer nodes to meet different applications’ request. Furthermore, CARS also provides options for users to select their preferred scheduling strategies and implements a user-guided scheduling scheme to better manage the resources.

3.1.2. Functional components

Fig. 4 shows a high-level view of the overall CARS framework with a typical HPC job scheduler (SLURM as an example) and the essential components. CARS provides a transparent and efficient resource management solution for the burst buffer system. We describe each component below.

Job tracer. Users submit their jobs to the job queue and then the job scheduler (e.g., SLURM) will arrange the execution order of the jobs in the job queue. Additional information of the jobs (e.g., the number of I/O processes) are provided by users as hints in the job scripts. When jobs are issued from the queue and accessing the burst buffer system, the job tracer will keep tracking the state of the current active jobs. For instance, suppose there are three jobs accessing the burst buffer nodes, and the job tracer will keep tracking the job’s I/O access information with pairs of (job name, the number of I/O processes), e.g., (Job1, 16), (Job2, 8), and (Job3, 8).

![Fig. 4. Overview of CARS framework.](image-url)
These information will be used as input in the CARS scheduling algorithm.

**Load monitor.** The load monitor will keep tracking how the current active jobs are distributed in the burst buffer system, such as on which burst buffer node each job is running on. Meantime, based on the status of active jobs monitored by the job tracer, the load monitor will calculate and record the I/O load (e.g., the number of concurrent I/O processes) of each burst buffer node. For example, suppose a job J consisting of 16 processes has been allocated to BB1 and BB2, by querying from the job tracer, the load monitor will be able to know the I/O load of BB1 and BB2, and generate real-time status in the format of (burst buffer node, job load), e.g., (BB1, 16), (BB2, 16), to maintain the load state of each burst buffer node.

**BB scheduler.** When a new job from the job queue is ready to execute, the burst buffer scheduler retrieves the information from the load monitor and determines which burst buffer nodes will be allocated to the new job by applying the scheduling algorithms. In the next subsection, we will describe a new contention-aware scheduling algorithm that can improve both job performance and the system utilization. After a new job has been allocated with the burst buffer nodes, the job tracer and the load monitor will update the corresponding state of current active jobs on the burst buffer nodes.

### 3.2. Contention-aware resource scheduling strategy

To maximize the resource sharing and utilization of burst buffer systems and minimize the I/O contention, we present a contention-aware scheduling strategy to allocate the burst buffer nodes for concurrent jobs. The scheduling strategy is shown in Algorithm 1. The fundamental idea is to choose the most under-utilized burst buffer node for a new job to minimize the I/O contention from other active jobs. We use the number of concurrent active I/O processes to assess the I/O load of each burst buffer node. For example, suppose a job A is assigned to m burst buffer nodes and the number of I/O processes of job A is pm. Among these m nodes, every pm/m processes of job A access one burst buffer node. When there are multiple jobs running on different burst buffer nodes, the resource scheduler can calculate the number of concurrent active I/O processes Ai for the ith burst buffer node. If a new job begins to execute, the scheduler will select the burst buffer node with minimum Ai (line 3) for this new job. If there are multiple burst buffer nodes with minAi concurrent I/O processes, the algorithm will choose the one with the minimum index (line 5) of burst buffer node. Then the algorithm will update the Ai of the allocated burst buffer node (line 11). If the job needs n burst buffer nodes, it needs to run n iterations to complete the allocation for this new job.

The proposed contention-aware scheduling is quite intuitive. It allocates burst buffer resources to meet users’ capacity requirement and also consider each burst buffer node’s I/O load. We use an example to illustrate the advantages of our proposed scheduling strategy. Let us assume there are four burst buffer nodes in total and four jobs coming in order. The number of I/O processes from Job 1 to Job 4 are 16, 8, 16, 8, respectively. Each job requests the capacity of two fix-sized chunks. As shown in the Fig. 5 (a), with the bandwidth policy, Job 1 and Job 3 share BB1 and BB2, but both of them have a large number of I/O processes, which can lead to I/O contention on these two burst buffer nodes. On the other hand, Job 2 and Job 4 with less I/O processes will share the other two nodes. With the interference policy, as shown in Fig. 5 (b), each job occupies one burst buffer node exclusively. Both the bandwidth policy and the interference policy cause uneven distribution of I/O processes, leading to load imbalance across different burst buffer nodes.

With the proposed contention-aware scheduling strategy, the resource scheduler chooses the burst buffer nodes with less I/O load for Job 3 and Job 4. In this case, each node has the same number of concurrent active I/O processes. On one hand, it prevents the possible I/O contention caused by jobs with high I/O load (e.g., Job 1 and Job 3) competing for the limited bandwidth of same burst buffer nodes. On the other hand, it also balances the I/O load across all burst buffer nodes in case of overloading or under-utilizing any nodes. We will present the modeling and analysis in the next section to further illustrate this new contention-aware resource scheduling strategy.

**Algorithm 1: Contention-Aware Resource Scheduling Algorithm.**

```
Input: A1, A2, A3, ..., Am
Output: allocatedlist
1 Set allocatedlist ← ∅;
2 for l to n do
3    minactive = minAi if multiple A_i equal to minactive then
4        newbb ← min i;
5    else
6        newbb ← i;
7    Add newbb to allocatedlist;
8    Update Ai[newbb];
9 end
```
4. Modeling and analysis

In this section, we discuss the proposed contention-aware resource scheduling (CARS) strategy through modeling and analysis. We also define evaluation metrics to understand the performance implications. We report detailed evaluation results in the next section. We first introduce the parameters to characterize users’ jobs and burst buffer systems, as shown in Table 1.

These parameters can be divided into two categories. The first category of parameters, denoted using capitalized letters, are constants that are related to the system specification. For example, the maximum bandwidth per burst buffer node in Cori system is roughly 6.5 GB/s. The current total number of BB nodes in Cori system is 288 as of April 2018, and the minimum allocation capacity for one BB node is either 20GB or 80GB. In general, these parameters describe a burst buffer system, and the actual values can be obtained from a particular deployment and used in scheduling. The other category of parameters, denoted with lowercase letters, represent variables specified in different jobs by the users. These parameters can be obtained either from users’ job script (e.g., a SLURM job script on Cori) or from the job queue. In general, the job scheduler has the knowledge of these information and can pass them to the burst buffer management software, i.e. the DataWarp in Cori system, for implementing our proposed contention-aware resource scheduling strategy.

Additionally, the proposed model describes the burst buffer as an independent I/O subsystem which is physically separated from compute nodes. The following subsections introduce the model of concurrent I/O behaviors on such storage architecture in detail. Therefore, it is ideal for shared burst buffer architectures such as those on Cori and Trinity. As for the node-local burst buffer storage on Summit, the proposed model does not apply to it because local NVMe-devices are distributed across compute nodes and cannot be viewed as an independent storage system.

4.1. Modeling and analyzing burst buffer allocations

Given the bandwidth strategy and the newly proposed CARS strategy (described in Section 3.2), a job’s request capacity will be striped across multiple burst buffer nodes, and each node contains at least one unit of minimum allocation capacity (i.e., G). Thus, for a particular capacity request (i.e., ri) of the ith job, it will be allocated with maximum rjG burst buffer nodes. In bandwidth allocation policy described in Section 2, these burst buffer nodes are selected in round-robin fashion. We assume each job’s I/O processes are evenly distributed among all their allocated burst buffer nodes. This is a reasonable assumption and can be easily achieved via the burst buffer management software (e.g., DataWarp) too. Therefore, when the ith job accesses the burst buffer system, there are pmGri processes concurrently running on each burst buffer node.

For the existing interference strategy, the behavior is rather simple, as each job gets allocated exclusive buffer nodes. For the CARS strategy, we further model the I/O contention and study its behaviors below.

4.2. Modeling and analyzing I/O contention

To study the impact of the I/O contention, we assess the bandwidth for each process when there are multiple jobs concurrently accessing the burst buffer system.

For simplicity, we assume one process from a job is corresponding to one CPU core on a compute node. Each process accesses the burst buffer and can transfer data across the network at maximum I/O rate bm gigabytes per second. Suppose there are N processes simultaneously accessing one burst buffer node, the aggregate bandwidth N × bm can be achieved, if it does not exceed the maximum bandwidth of one burst buffer node (i.e., BM). In many cases, N is pretty large and we define that the I/O contention happens when the aggregated bandwidth of concurrent I/O processes, i.e. N × bm exceeds one burst buffer node’s peak bandwidth BM. In such scenario, we assume the I/O bandwidth of this burst buffer node is equally shared by all concurrent processes from all jobs accessing it. Thus, the I/O bandwidth for each process can be computed as below. Note that the first case represents a normal case, while the second case represents the case when I/O contention happens.

\[
I = \begin{cases} 
\text{bm} & \text{if } N \times bm \leq BM \\
\frac{BM}{N} & \text{if } N \times bm > BM 
\end{cases}
\]  

4.3. Modeling and analyzing I/O time

In our modeling and analysis, we separate the I/O time from the first job starting accessing burst buffer to the last job finishing execution into multiple time steps. Each time step (ti) begins and ends with discrete events such as arrival of a new job on burst buffer system or completion of a running job. Each event can change the bandwidth distribution of concurrent processes among active jobs. During each time step, the average I/O bandwidth of each process can be considered as a constant. Therefore, suppose a job is accessing multiple burst buffer nodes and every process of this job accesses one node, the aggregated I/O bandwidth of this job during the ith time step (ti) can be calculated as:

\[
b_i = \sum_j l_j \times p
\]  

In Eq. (2), l_j (i.e. the bandwidth of a process) may vary among different burst buffer nodes, and each l_j can be calculated via Eq. (1). We assume that a job transfers d_j bytes data during the ith time step (t_i). Thus, the I/O time of this job can be computed as:

\[
T = \sum_i \frac{d_j}{b_i}
\]  

4.4. Evaluation metrics

To evaluate the job performance and the overall system utilization, we present three evaluation metrics below. These three evaluation metrics are orthogonal to each other and represent the most important three dimensions we focus on for evaluating burst buffer allocation strategies.

4.4.1. Average job efficiency

We define the efficiency of a job as the ratio of the I/O time when the job runs exclusively on one or more burst buffer nodes (denoted as TE) to the I/O time under I/O contention (denoted as TC). For example, the job efficiency of job A (Eff_A) is \(\frac{TE_A}{TC_A}\). Apparently the value of Eff_A is not greater than 1 and the case of Eff_A = 1 indicates that job A’s performance is not affected by any I/O contention. Suppose there are n jobs concurrently accessing the

Table 1
Parameters/variables used in modeling and analysis.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM</td>
<td>Maximum bandwidth of one BB node</td>
</tr>
<tr>
<td>CM</td>
<td>Maximum capacity of one BB node</td>
</tr>
<tr>
<td>N</td>
<td>Total number of BB nodes</td>
</tr>
<tr>
<td>G</td>
<td>Minimum allocation capacity for one BB node</td>
</tr>
<tr>
<td>pi</td>
<td>Number of I/O processes of the ith job</td>
</tr>
<tr>
<td>r_i</td>
<td>Total number of jobs</td>
</tr>
<tr>
<td>r_j</td>
<td>Request capacity size of the jth job</td>
</tr>
</tbody>
</table>
burst buffer system, the average job efficiency is defined as the geometric mean of all individual job efficiency, which can be calculated as:

\[ \text{Eff} = \left( \prod_{i=0}^{n-1} \frac{T_j}{T_i} \right)^{1/n} \]  

(4)

This evaluation metric is user-oriented and represents the fairness across jobs by evaluating the degree of slowdown due to I/O contention for each job in burst buffer scheduling. It is also obvious that \( \text{Eff} \) is not larger than 1, and the larger \( \text{Eff} \) means better performance for all jobs.

4.4.2. Average wait time

If the current number of burst buffer nodes, or the available space of the burst buffer system, cannot meet the next arriving job's capacity requirement, the following jobs will be blocked, waiting until there is enough available burst buffer nodes or capacity. The wait time of a job is defined as the interval between the time the job is submitted and the time the job starts to run. Suppose there are \( n \) jobs and the wait time of the \( i \)th job is \( w_i \), the average wait time of these \( n \) jobs can be calculated as below. This metric is also user-oriented but represents the perspective from job waiting time.

\[ \text{Wait} = \frac{1}{n} \sum_{i=0}^{n-1} w_i \]  

(5)

4.4.3. Average system utilization

Given \( N \) burst buffer nodes in total in the system and \( \text{BM} \) maximum I/O bandwidth for one node, during the \( i \)th time step \( (t_i) \), the aggregate I/O bandwidth of all current active jobs is \( \text{BW}_i \), which can be calculated by summing each active job's I/O bandwidth using Eq. (1) and Eq. (2). We define the average system utilization for the burst buffer system as:

\[ \text{Util} = \frac{\sum_{i=0}^{n-1} \text{BW}_i 	imes t_i}{N \times \text{BM} \times \sum_{i=0}^{n-1} t_i} \]  

(6)

In Eq. (6), the numerator can be described as the amount of data actually transferred during the entire I/O time of all jobs, and the denominator can be interpreted as the theoretical maximum amount of data transferred by using all burst buffer nodes. This metric \( \text{Util} \) is system-oriented and assesses the bandwidth utilization of the entire burst buffer system.

5. Evaluation

This section describes our evaluation methodology and results. We first introduce the emulation experiments at small to medium scale on a production HPC system (the Cori system at NERSC). These tests serve two purposes. First, these tests evaluate the proposed CARS strategy on a real system. Second, we leverage these tests to validate the model we have described in Section 4. We then present the simulation tests based on the validated model with large-scale I/O workloads to further understand the performance implications.

5.1. Emulation experiments

5.1.1. Evaluation setup

The emulation evaluations were conducted on the burst buffer system of Cori Phase II system at NERSC. Cori is a Cray XC40 supercomputer comprised of 9688 Intel Xeon Phi™ 7250 (Knights Landing) and 2388 Intel Xeon™ Processor E5-2698 (Haswell) compute nodes. We used 16 Haswell compute nodes to run the benchmark application, and each node has two 2.3 GHz 16-core Haswell processors and 128 GB of DRAM.

The Cori burst buffer system has 288 nodes. Each burst buffer node contains two Intel P3608 3.2 TB NAND flash SSDs which can provide approximately 6.4 TB of usable capacity and a peak of approximately 6.5 GB/s of read and write bandwidth. The Cray DataWarp software integrated with Slurm workload manager is responsible for the burst buffer resource management. Due to the limited capacity quota for a user account, we were allocated up to 8 burst buffer nodes for exclusive use in order to avoid interference from other users’ jobs.

5.1.2. Benchmark

We used IOR [18], a parallel file system benchmark which was developed by Lawrence Livermore National Laboratory as an evaluation benchmark. We configured the IOR parameters to build five synthetic I/O workloads, as shown in Table 2. All processes of a job are evenly distributed across the allocated burst buffer nodes and I/O operations are performed on burst buffer nodes concurrently.

Since we cannot modify the resource scheduler in Cori system, we implement a runtime prototype of CARS running in the client node to control the jobs to access burst buffer nodes by different resource allocation policies.

5.1.3. Results and analysis

We submitted 10 jobs (each job listed in Table 2 was selected twice) and assigned these jobs to run on burst buffer nodes under different resource allocation strategies. We set the number of request capacity of each job as two chunks and one burst buffer node will contain maximum two chunks at one time. We conducted the emulation evaluation and also used the experimental results to compare with the theoretical results that can be calculated from the model described in Section 4.

Fig. 6(a) plots the average job efficiency under different allocation policies. As we can see from the figure, contention-aware (CA) strategy achieved the highest efficiency, up to 93%, and outperformed the bandwidth (BA) policy by 25% on average. This is because, with the contention-aware policy, job with high I/O load like Job 1 will be shared among burst buffer nodes with low I/O load, such as Job 4. This strategy allows each job to get sufficient bandwidth and achieve almost the same performance as they were allocated with the burst buffer nodes exclusively. The interference (IN) policy achieved the worst performance. The reason is rather straightforward, as each job was only allocated half number of burst buffer nodes compared to other policies. In all cases, the error between theoretical model and experimental results is only up to 8%.

Fig. 6(b) shows the average system utilization under different allocation policies. The contention-aware strategy outperformed the bandwidth policy by 20% on average and made the utilization of each burst buffer node much more balanced. For the case of the bandwidth policy, jobs with high I/O load ran together on the same burst buffer nodes, resulting clearly observable contention on these nodes, while other nodes were underutilized due to low I/O load. The imbalance of utilization was even worse in the interference policy. The experimental results were close to theoretical results too, and they confirm the similar trends.
The contention-aware and bandwidth policy had the least waiting time. This is because the jobs with less I/O load finished in shorter time and the pending jobs in the job queue got executed earlier, resulting in reduced waiting time. For the interference policy, jobs with less write size finished much longer since they were allocated with fewer nodes, causing longer wait time for the pending jobs.

In summary, these results demonstrate the advantage of the proposed contention-aware policy. These results match the theoretical results reasonably well, and the errors were between 2% to 10%, which verified the model and analysis we have described earlier.

5.2. Simulation experiments

To study the effectiveness of the proposed contention-aware resource scheduling on large-scale HPC systems, we conducted simulation evaluation too. Taking the parameters of I/O workloads and burst buffer system as input, the simulation executes resource scheduling policy to simulate the results.

5.2.1. I/O workload and BB system configuration

We conducted simulation experiments with the I/O workloads that represent large-scale applications. Based on the analysis of Darshan I/O logs on petascale HPC systems [20], we created a group of I/O workloads to mimic the real-world applications. Table 3 shows the workload configuration, including the number of I/O processes and the amount of data written. In addition, we divided the workloads into two categories. The first category is consisted of large-scale jobs with more than 1024 processes. The second category is composed of medium to small scale jobs that use 1024 processes or less. We further created different workload by combining different scales of workloads to represent the scenarios of a mix of jobs accessing the shared burst buffer system concurrently. In the simulation, we used three sets of workload combinations described below.

- Set1: all jobs are large scale jobs;
- Set2: all jobs are medium to small scale;
- Set3: a mixture of large scale and medium to small scale jobs.

For the burst buffer system, we set the total number of burst buffer nodes as 256, the peak bandwidth per node as 6.5 GB/s, and the maximum capacity as 6.4TB per node. This configuration is close to the configuration of the Cori burst buffer system [6].

5.2.2. Results and analysis

In the simulation experiments, two scales of jobs, 100 large-scale jobs or 1000 medium- to small-scale jobs, were used for each workload set. In addition, three job scheduling policies (FCFS, SJF, and PS) were simulated to decide the execution order of jobs in each workload set. For the FCFS (First come first serve) policy, we simply randomize the execution order of the jobs. For the SJF (Short job first) policy, jobs with less write data size were given higher priority for execution. For the PS (Priority scheduling) policy, jobs with larger writes were given higher priority to run. Each workload set for each job scheduling policy was repeated 5 times and we present the mean value in this section.

Fig. 7 presents the average job efficiency of three workload sets separately. We first observe that the contention-aware strategy can improve the job efficiency compared to the other two policies in most cases. The exception happened in Set 2 using SJF, which has almost the same job efficiency with the bandwidth policy. It is due to the similar allocation behaviors under these two different policies. We also observe that the contention-aware strategy achieved the largest improvement by more than 20% compared to the bandwidth policy in Set 3. The reason is that small-scale jobs can have sufficient I/O bandwidth they need and can finish quickly under the contention-aware policy, which in turn improved the average job efficiency.

Fig. 8 shows the average system utilization on three workload sets. This performance metric is system-oriented and we can clearly observe that our approach achieved higher system utilization on all workload sets, which indicates that the contention aware policy is more efficient in utilizing the available bandwidth of burst buffer system than other policies. The highest system utilization is observed with SJF in workload set 3, where the system utilization achieved nearly 90%. This is most likely because most jobs can accurately select the underutilized burst buffer nodes to access and get enough bandwidth in this case.

Fig. 9 reports the average waiting time on three workload sets. The contention-aware scheduling strategy has the shortest wait time, but only slightly shorter than that of the bandwidth policy. This indicates that the job performance of medium- to small-scale jobs was close in these two policies, which lead to similar wait time. As for the interference policy, medium-/small-scale jobs obtained very limited bandwidth and had longer I/O time than the other two policies.
6. Related work

Numerous research studies have examined the burst buffer system and I/O contention issue in HPC systems. In this section, we review these existing studies and compare them with this study.

6.1. Burst buffer

Several studies [4,11,21] have demonstrated integrating burst buffer into HPC systems is a promising solution for addressing the I/O bottleneck for data-intensive applications. Chen et al. [13] proposed an active burst buffer architecture to enhance the existing burst buffer system with data analysis capabilities. Wang et al. [10] studied node-local burst buffers to deliver scalable and efficient aggregation of I/O bandwidth for checkpoint and restart of data-intensive applications. Bent et al. [22] also studied node-local burst buffers to deliver scalable write performance for local I/O requests. Our study focuses on the shared burst buffer systems that can be accessed by all compute nodes in the HPC system. Additionally, Han et al. [23] proposed a user-level I/O isolation scheme to minimize the overhead for SSDs in burst buffers. Thapaliya et al. [24] investigated I/O scheduling techniques as a mechanism to mitigate burst buffer I/O interference. Kougkas et al. [25] proposed a dynamic, interference-aware I/O scheduler for shared I/O buffering system to handle concurrent accesses. In our work, we focus on studying the burst buffer resource allocation strategies for the scenarios of multiple concurrent data-intensive applications.

6.2. I/O contention

Numerous studies [26–28] focused on the I/O contention issue in HPC systems. Lebre et al. [29] introduced a scheduling method to efficiently aggregate and reorder I/O requests. Zhou et al. [30] proposed a novel I/O-aware batch scheduling framework to coordinate ongoing I/O requests on petascale computing systems. Gainaru et al. [31] proposed several scheduling approaches to mitigate I/O congestion by analyzing the effects of interference on application I/O bandwidth. Herbein et al. [32] proposed a batch job scheduling method to reduce contention by integrating I/O awareness into job scheduling policies. These efforts mainly focus on reducing I/O contention in parallel file system. In our study, we aim to address the I/O contention issue through the resource management in the burst buffer system.
7. Conclusion

In recent years, burst buffer has emerged as a promising solution to addressing the I/O bottleneck issue of many data-intensive applications by absorbing bursty I/O traffic. However, the resource allocation and management strategies for burst buffer are still largely underdeveloped. In this paper, we have presented a new resource scheduling strategy, called contention-aware resource scheduling (CARS), to mitigate the I/O contention and improve the performance of burst buffer systems. We have studied different resource scheduling policies of the existing burst buffer systems in detail and provided modeling and analysis about the I/O performance. We have derived the I/O contention model and different metrics for evaluation. We have also conducted comprehensive experiments including both emulation and simulation and the results have shown that our proposed CARS strategy outperforms the existing allocation strategies and improves both the job performance and the system utilization. Such a strategy can be particularly useful for large-scale systems where the access contention becomes the norm. In the future, we plan to further investigate resource management and scheduling problems on other burst buffer architectures and storage medium, such as an architecture with node-local NVMe as in the latest Summit supercomputer. In addition, we plan to explore different I/O patterns of real-world applications and leverage these patterns to further optimize resource management and scheduling approaches.

Acknowledgment

We are thankful to the reviewers for evaluating this study and providing valuable feedback. This research is supported in part by the National Key Research and Development Program of China (2017YFB0202002), the National Science Foundation under grant CNS-1338078, CNS-1362134, CCF-1409946, and CCF-1718336.

References
