Efficient Job Scheduling for Clusters with Shared Tiered Storage

Leah E. Lackner, Hamid Mohammadi Fard, and Felix Wolf
Technische Universität Darmstadt, Germany
{lackner, fard, wolf}@cs.tu-darmstadt.de

***Abstract*—New fast storage technologies such as non-volatile memory are becoming ubiquitous in HPC systems with one or two orders of magnitude higher I/O bandwidth than traditional back-end storage systems. They can be used to heavily speed-up I/O operations, an essential prerequisite for data-intensive exascale computing capabilities. However, since the overall capacity of the fast storage available in a system is limited, an individual job may not always benefit if access to fast storage implies longer waiting time in the queue. This is obvious if fast storage is shared across the system. We therefore argue that the decision of whether or not to use fast storage should be supported by the batch scheduler, which can estimate when the amount of fast storage a job desires will become available. We present a scheduling algorithm with this functionality and show in simulations significantly reduced makespan and turnaround times in comparison to always using fast storage, always using slow back-end storage, and random storage assignment.**

***Index Terms*—scheduling, I/O, resource management, tiered storage, non-volatile memory, data-intensive applications**

I. Introduction

Novel storage technologies pave the way to data-intensive exascale computing. In the past, usually a parallel file system (PFS)—possibly backed up by a tape archive—was the only available permanent storage. With the emergence of compact and energy-efficient solid-state drives (SSDs), it became viable to attach hard drives to nodes, serving as burst buffers [2] that cache data on their way to the PFS, significantly mitigating the file-system bottleneck. Burst buffers can be installed locally on each compute node or be provided in a remote shared configuration. Today, *non-volatile memory express* (NVMe) devices with their superior performance mark a further step towards fast I/O. As a generalization of the burst-buffer concept, such devices can be used to store a variety of intermediate data, including check points, without necessarily always propagating the data further down the storage hierarchy. Since such fast storage hardware will soon by available in many machines, it is important to study its impact on system and job performance.

On an abstract level, new fast storage technologies can be interpreted as a kind of accelerator for data-intensive software, playing the same role for I/O that GPUs play for computation. They can speed up a certain part of program execution, but are not necessarily mandatory to complete it. Just like a program can be written in such a way that it runs on regular processors if no GPU is available, an application could as well resort to the back-end PFS if there is no fast storage in place.

Because the fast storage available in a system is limited by capacity, the most resource-efficient way to balance the requirements of a diverse workload across the entire system is to make it globally accessible from all compute nodes, for example, implemented as an interconnected distributed file system, such as HDFS [17] or BeeGFS [8]. This is the scenario we focus on in this paper. In addition to allocating compute nodes, jobs can now also request a share of the available fast storage, another resource that must now become subject to scheduling.

Since users of an HPC system often care most about the turnaround time of their jobs, which also covers the waiting time in the queue, and not so much about the pure execution time on the system, the option to use fast storage confronts them with a choice if fast storage is not immediately available: better waiting for it and risking longer residency in the queue or going ahead without it? Because a user usually lacks the knowledge to draw a quantitative comparison, this decision should rather be delegated to the scheduler. Of course, such a strategy must be supported by accounting incentives. However, we argue that if accepting a longer execution time in exchange for shorter turnaround also helps to increase system utilization, the provider has an incentive not to penalize users for the slow storage overhead they will suffer. To tackle the problems of scheduling globally accessible fast storage alongside compute nodes, this paper makes the following two contributions:

1) Based on backfilling, we propose a novel **storage-aware scheduling algorithm** that efficiently schedules compute nodes together with capacity-bound storage resources.
2) We evaluate the algorithm in simulations, which demonstrate **significantly improved turnaround times and workload makespan** in comparison to always using fast storage, always using slow back-end storage, and random storage assignment.

The remainder of this article is organized as follows: First, we discuss related work in Section II. Then, we outline the storage-aware scheduling algorithm in Section III, after which we present our simulator setup in Section IV, followed by our evaluation in Section V. Finally, we draw a conclusion and discuss future directions in Section VI.

II. Related Work

In our review of related approaches in the field, we focus mostly on the scheduling of data-intensive jobs, considering both flat (traditional) storage and multi-tier storage architectures. Assuming shared storage distributed across a grid, Kosar and Balman [13] proposed a data-aware scheduler that ensures

efficient data placement by ordering data transfer requests using techniques such as first fit, best fit, largest fit, and smallest fit. Our work, in contrast, balances the tradeoff between the desire to access fast storage and its limited capacity.

Wan et al. [21] considered a hybrid storage configuration, including SSD and HDD storage, similar to the one we target here, and proposed automatic object placement techniques, moving objects between different tiers according to their popularity. While they only strive to minimize data transfer times, our approach aims to reduce the turnaround times of jobs via co-allocation of compute and limited fast storage resources.

Herbein et al. [9] proposed I/O-aware versions of two batch-job-scheduling techniques, namely, first-come, first-served (FCFS) and EASY backfilling. They assume burst buffers physically close to compute nodes, whereas our work supports a shared two-tier storage architecture.

The framework introduced by Isaila et al. [10] coordinates data-staging on HPC platforms. While they achieve genericity by fully decoupling control from data planes, they do not consider the increased scheduling complexity caused by storage with limited capacity and miss optimization opportunities arising from this complexity.

Zhang et al. [24] studied the challenges of migrating frequently accessed data in a multi-tier storage environment. They presented an adaptive deadline-aware lookahead data-migration scheme for clouds. Their work aims to improve resource utilization by efficiently using the power of fast SSDs with limited capacity.

Ramakrishnan et al. [15] presented a scheduling approach for data-intensive large-scale scientific workflows on distributed storage, including disks with limited capacity. Different from their work, our scheduler reserves storage in advance, a technique that has been shown to improve resource utilization in several studies [6, 19, 23].

Overall, the most distinctive feature of our work in comparison to the state of the art is the improvement of turnaround times via storage-aware scheduling, hiding the underlying multi-tiered storage configuration from users. In some way, our approach is similar in spirit to backfilling [14, 20], as we strive to run jobs earlier than what would be their start time under normal conditions (i.e., when the requested fast storage become available) if this improves system utilization. Finally, co-allocation of compute and storage resources can also be seen as a variant of scheduling for heterogeneous clusters [1, 18], as requirements for different types of resources must be properly balanced.

III. Storage-Aware Scheduler

In this section, we describe our storage-aware scheduler. Before delving into the details of the scheduling algorithm, however, we first introduce the underlying platform and job model.

A. Platform model

We model the platform as a cluster consisting of a set of homogeneous compute nodes which have access to a shared tiered storage system. For the sake of simplicity, we define two storage tiers, a configuration that, however, is easily extendable to a multi-tier storage architecture. Our model comprises low-performance storage (LPS) with practically unlimited capacity and high-performance storage (HPS) with limited capacity. We imagine LPS to be implemented as a backend parallel file system and HPS to reside on separate I/O nodes equipped with SSDs or NVMe SSDs, although this separation is no mandatory property of our model as long as the HPS storage is globally accessible, for example, through a distributed file system. As shown in Figure 1, LPS and HPS provide different data transfer rates R_{lps} and R_{hps} to and from compute nodes, respectively. The data-transfer rate between the two storage tiers is denoted by R_{stage}. Each of the three links between compute cluster (i.e., compute nodes), LPS, and HPS is shared. This means the available bandwidth is split across all data streams between any two of the three layers, which can, of course, give rise to congestion.

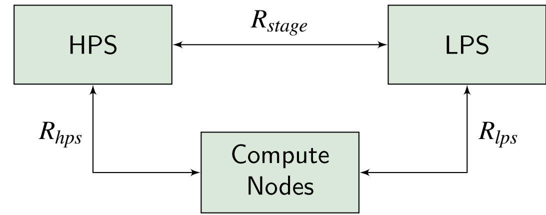


Figure 1. Two-tier storage architecture with slow but unlimited low-performance storage (LPS) and fast but limited high-performance storage (HPS).

B. Job model

We consider a set J of n jobs $J = \{j_1, \ldots, j_n\}$ with different arrival times. For each job j_i, we assume that the user has accurate knowledge of the following job attributes:

- Required number of compute nodes: $nodes_i$
- Desired amount of fast storage: $storage_i$
- Input file size: $input_i$
- Output file size: $output_i$
- Walltime: $walltime_i$
- Amount of intermediate data read and written to files: $data_i$

Asking the user to provide such detailed information may seem unreasonable at the first glance, but we argue that sufficiently precise estimates of these numbers can be easily obtained with standard profiling tools such as Score-P [12], LWM^2 [16], or with storage estimators such as those as proposed by Hazekamp et al. [7]. Note that even if the desired amount of HPS storage is granted, the user can still use LPS for other tasks—in addition to the desired use of HPS. In this case, the time needed to access LPS is factored into the walltime and treated like in a traditional scheduler.

C. Scheduling algorithm

To efficiently schedule compute nodes and fast storage capacity in combination, we introduce a *storage-aware scheduler* (SAS) that evaluates the benefit of using fast storage and

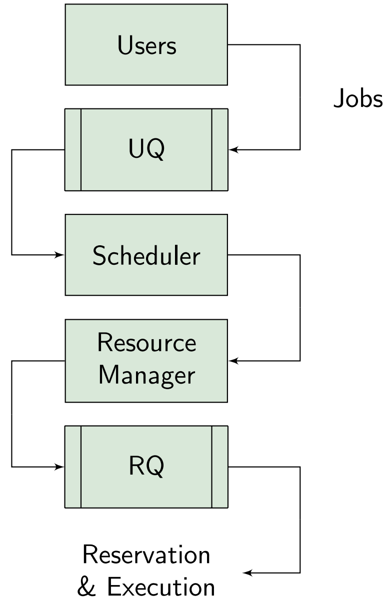


Figure 2. Flow of jobs inside the scheduler from the arrival point to the allocation of resources and the start of execution.

compares it with the overhead of using it. Our scheduler is a dynamic non-preemptive scheduler because the jobs enter the system dynamically and once a job has been started, we are not allowed to stop and resume it afterwards.

As shown in Figure 2, our scheduler uses two queues. The first queue, which is called *UQ*, contains the unscheduled jobs as they are submitted by users. After making the schedule decision for a job j_i, the scheduler dequeues this job from *UQ* and adds it to *RQ*, the queue of running jobs. After a job terminates, the resource manager removes the corresponding entry from *RQ*.

An entry for job j_i in the running-jobs queue *RQ* is a tuple $schedule_i = (j_i, start_i, end_i, nodes_i, storage_i)$, where $start_i$ and end_i denote the start and end time of the reservation, respectively, $nodes_i$ the set of nodes that have been reserved, and $storage_i$ the amount of fast storage to be allocated. Note that slow storage, which offers unlimited capacity, does not require prior allocation.

For an unscheduled job $j_i \in UQ$, which is selected following a shortest-job-first backfilling approach [20], our scheduler needs to decide which storage tier, HPS or LPS, the job is supposed to use. For this purpose, our scheduler calculates two possible turnaround times T_i^{lps} and T_i^{hps} according to Equations 1 and 2.

$$T_i^{lps} = waiting\text{-}time_i^{lps} + walltime_i \tag{1}$$

$$T_i^{hps} = waiting\text{-}time_i^{hps} + walltime_i - \frac{data_i}{R_{lps}} + \frac{data_i}{R_{hps}} + \frac{input_i + output_i}{R_{stage}} \tag{2}$$

Whichever is shorter determines the storage tier assigned to the job. Equation 1 calculates the turnaround time for job j_i in its schedule under the assumption it would use LPS. To calculate the turnaround time when using HPS instead, the scheduler needs to take both the overhead and gain of using HPS into account. Since the capacity of the HPS tier is limited, choosing it may lead to longer waiting times. For the sake of simplicity, the current version of the algorithm does not try to estimate link congestion but rather assumes that each data stream can utilize the entire available bandwidth, a condition which may not always be satisfied in practice.

Of course, accounting must be adjusted not to penalize a user for the longer execution time that an assignment of slow storage may imply. Since the flexibility our scheduler introduces also helps improve system utilization, as we will show in Section V, we believe that system providers can easily create economic incentives for users to accept the scheduler decision that would also serve the providers' interests. Although the details of such an accounting scheme are beyond the scope of this paper, one can imagine charging on the basis of the (hypothetical) execution time with LPS. As an almost equivalent alternative, one could impose a surcharge on the usage of expensive fast storage.

The scheduler computes the queue waiting times that appear in the two equations, namely $waiting\text{-}time_i^{lps}$ and $waiting\text{-}time_i^{hps}$, based on the system state, which includes the schedules in *RQ* and the reservation plan of resources, and the job requirements, such as the desired amount of fast storage $storage_i$, the requested walltime $walltime_i$ and the requested number of compute nodes $nodes_i$.

Algorithm 1 represents the core of the scheduler, which is invoked whenever the state of a job or a resource changes. The first loop (lines 1–5) finds jobs with high priority, while the second loop (lines 6–8) backfills jobs with low priority. The loop to schedule high-priority jobs exits when it finds the first job that has not been scheduled. This triggers the backfilling part of the algorithm. Backfilling low-priority jobs in shortest-job-first fashion, choosing the one with the lowest storage requirements if there is a tie, ensures that jobs with higher requirements do not block resources and prevents jobs from starvation. However, high-priority jobs are still selected based on their position in queue *UQ*. To keep track of the priority jobs in the algorithm, we need a temporary set of priority jobs called *P*, which is used by function *try-start-storage-aware*(), invoked in lines 2 and 7, to backfill jobs.

Algorithm 1 Storage-aware scheduler (SAS)

1: **for** $j_i \in UQ$ **do**
2:     **if** $\neg$*try-start-storage-aware*$(j_i, \{\})$ **then**
3:         **break**
4:     **end if**
5: **end for**
6: **for** $j_i \in UQ \setminus \{j_1\}$ **do**
7:     *try-start-storage-aware*$(j_i, \{j_1\})$
8: **end for**

Function *try-start-storage-aware*() determines the best possible schedule for job j_i, as shown in Algorithm 2. The two

functions *find-lps-schedule*() and *find-hps-schedule*() in lines 1 and 2 return possible candidate schedules for job j_i under the assumption of using either LPS or HPS, respectively. The functions consider the job requirements of j_i to propose possible schedules. Moreover, because HPS has limited capacity, the function *find-hps-schedule*() has to find a time slot with sufficient storage capacity available for the duration of the job's walltime. In lines 3–7, we choose the schedule candidate with the shorter turnaround time, calculated using Equations 1 and 2. If the proposed schedule is executable just right now and it does not delay any job in the set of priority jobs P (lines 8–15), the scheduler inserts the schedule entry to the queue of running jobs RQ. In lines 13 and 16, the function returns *true* if the job was scheduled and *false* otherwise. Afterwards, the resource manager starts the execution of job j_i.

Algorithm 2 *try-start-storage-aware*(j_i, P)

1: $schedule_i^{lps} \leftarrow$ *find-lps-schedule*(j_i)
2: $schedule_i^{hps} \leftarrow$ *find-hps-schedule*(j_i)
3: **if** $T_i^{hps} < T_i^{lps}$ **then**
4: $\quad schedule_i \leftarrow schedule_i^{hps}$
5: **else**
6: $\quad schedule_i \leftarrow schedule_i^{lps}$
7: **end if**
8: **if** $schedule_i$ can start immediately **then**
9: $\quad$ **if** $schedule_i$ does not delay $j_k \in P$ **then**
10: $\quad\quad$ *enqueue*$(schedule_i, RQ)$
11: $\quad\quad$ *dequeue*(j_i, UQ)
12: $\quad\quad$ *start*(j_i)
13: $\quad\quad$ **return** *true*
14: $\quad$ **end if**
15: **end if**
16: **return** *false*

The proposed algorithm can be easily extended to a multi-tier storage architecture with more than two tiers. The existence of the different tiers would be covered by rewriting the main decision-making part of Algorithm 2 in lines 3–7.

IV. Simulation Setup

For our evaluation, we designed a simulation environment based on the simulator framework Batsim [4]. Furthermore, we designed a workload model featuring both computational and I/O requirements, derived from an established workload model. The experiments and their results are described in Section V.

A. Simulation environment

Compared to other simulators, such as Gridsim [3] or Alea2 [11], Batsim is more versatile because the scheduler is separated from the resource management logic. Following this separation, we implemented our simulation and scheduling logic in Python using PyBatsim. The storage is managed in the scheduler, whereas the actual I/O activities are simulated as part of the resource manager.

Our simulated platform comprises 128 compute nodes, with each node offering 16 GB RAM of main memory, which defines the checkpointing requirements per node. To specify the data transfer rates, we assume that the LPS tier uses some kind of back-end PFS technology, such as Lustre or GPFS. Transfer rates of traditional burst-buffer technologies exceed the performance of classical parallel file systems by up to a factor of three (cf. Shaheen in the IO-500 list[1]). Based on comparisons of recent but classic SSDs with new NVMe-based SSDs, the latter of which we chose for our study, we multiply this ratio by a factor of five, resulting in a total throughput ratio of 15 between R_{hps} and R_{lps}. Taking into account that staging involves largely sequential data, which is not necessarily the case for data access at job runtime, we define the transfer rate R_{stage} available to staging traffic between the HPS and the LPS tier to be five times larger than the regular LPS transfer rate R_{lps}. Undoubtedly, actual system performance depends on how the tiers are precisely interconnected. As a constraint imposed by our simulator, the three links between LPS, HPS, and compute nodes appear physically separated, which means there is no interference whatsoever between any of the two links in our simulation. This condition could be violated in a real system, where, for example, staging traffic and regular I/O to the backend file system may compete for bandwidth. Nevertheless, individual links are shared, as described in our platform model in Section III-A.

In our experiments, we consider two fast-storage capacities: (a) 2048 GB, which is sufficient such that a job using 128 nodes can still do checkpointing while utilizing all compute nodes of the cluster, and (b) 4096 GB, which adds additional capacity to be used for I/O beyond checkpointing. In the second scenario, more jobs using the faster storage can be scheduled simultaneously.

B. I/O workload model

Based on the *Feitelson 1996 Model* [5], we designed an algorithm to generate the job specifications, including arrival time, walltime, the amount of input and output data, the amount of intermediate data written to and read from files at runtime, and the amount of requested HPS storage capacity.

We define the overall data volume subject to I/O by filling a certain fraction of a job's runtime with reading and writing file data, based on a fixed LPS data transfer rate R_{lps}. This means the key simulation variables are the fraction of time a job spends performing I/O and the ratios among the three bandwidth values R_{lps}, R_{stage}, and R_{hps} mentioned in Section IV-A. Everything else is scaled proportionally. The precise fractions spent reading input and writing output are randomly selected based on a normal distribution, taking up to 10% of the runtime on average for both together. Imagining checkpointing as a common use case, the amount of intermediate I/O $data_i$ is derived from the amount of main memory available per node with the frequency again adjusted to match a certain fraction of the overall runtime. There is no checkpointing for short jobs. The requested amount of the storage is the sum of the maximum staging space required at any time on the HPS, that is, $max(input_i, output_i)$ and the size of a checkpoint. Table I shows the parameters of the *Feitelson 1996 Model* used in our jobs simulation.

[1] http://www.io500.org

Table I. Parameters used for the instantiation of the *Feitelson 1996 Model* to create the workloads.

	Parameter	Value
Feitelson 1996 Model	Number of jobs	500
	Max. repetitions	2
	Arrival factor	500
	Shortest job time [s]	1800
	Longest job time [s]	86400

Each job can request fast HPS, but the final decision is made by the scheduler. We simulate each job as a sequence of steps, where the exact composition of this sequence depends on the storage type that is actually used (e.g., if the job is assigned to HPS, additional staging of input and output might be required): (1) If the job uses HPS the input is staged in from LPS to HPS before starting the job. (2) Although, in practice, reading of input data may occur anytime during the lifetime of the job—also because it may not fit into main memory—we simulate it immediately at the begin of a job's execution, a simplification with neutral effect on the overall runtime. (3) Subsequently, the job operates on the data, which involves computation as well as communication with other nodes belonging to the job's allocation. While the job is running, additional disk I/O can occur, for example, to write checkpoints down to either LPS or HPS. (4) Even though, the writing of results back to LPS or HPS may also happen anytime during the lifetime of the job, we simulate it right at the end of a job's execution, similar to the reading of input data in step (2), again a simplification with neutral effect on the overall runtime. (5) If HPS was used, the job needs to stage the results out to LPS to store them permanently.

V. Evaluation

To evaluate the performance of SAS, our storage-aware scheduler, we conducted several simulation experiments, divided into two parts. First we compare SAS with random storage assignment under the assumption that the user estimates the job requirements accurately. In a next step, we show how well SAS resists inaccurate I/O requirement specifications. Applying the workload model from Section IV-B and using the same set of parameters, we generated ten different workloads using distinct random seeds. The workloads share key properties, in particular, they have the same number of jobs and similar job distributions and I/O requirements for individual jobs. As a consequence, their makespans have the same order of magnitude. We aggregated our results for makespan and turnaround time by computing the arithmetic mean across all workloads.

A. SAS vs. random storage assignment

In Figure 3, we show how SAS improves makespan, turnaround time, and resource utilization in comparison to random storage assignment. As explained in Section IV, we consider two different storage capacities for HPS, namely 2048 GB and 4096 GB. The random storage assignment for each job follows a Bernoulli distribution, such that p denotes the probability to choose the fast HPS tier and $1-p$ represents the probability to choose the slower LPS tier. We ran experiments with different probabilities p, ranging from 0 to 1 in steps of 0.05. The extreme probabilities of 0 on the one hand and 1 on the other represent two baselines. 0 corresponds to always using slow storage and 1 to always using fast storage. To increase statistical significance, we repeated the experiments ten times for each probability value p and took the average.

According to Figure 3a, SAS shows remarkably better makespan than random storage assignment. As we increase the probability of using HPS, the makespan of random storage assignment reaches a local minimum around the probability of 60–80%, after which it increases again. This behavior can be explained with longer waiting times, caused by a significant portion of the jobs insisting on using the HPS tier. Obviously, random storage assignment fails to fully exploit the potential of tiered storage, a gap that SAS closes much better. Note that the makespan for a HPS capacity of 4096 GB is shorter than for 2048 GB. This suggests to scale up the HPS capacity as far as possible. However, the higher cost of fast storage may impose an economic limit here.

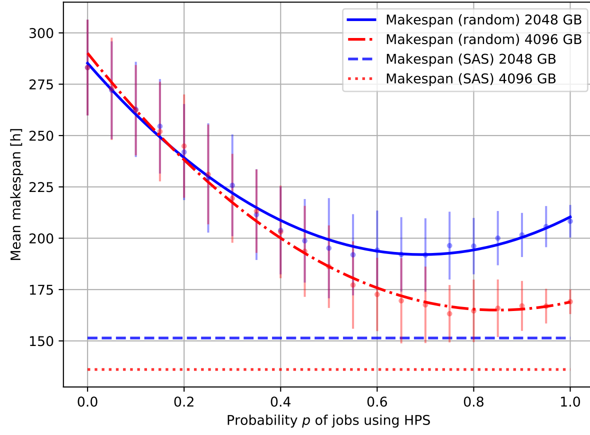
Figure 3b shows the arithmetic mean of the turnaround times for all jobs of the ten workloads. Similar to the makespan results, the mean turnaround time under random storage assignment improves as the probability of HPS usage increases towards $p = 0.4$, after which the trend is reversed. This behavior again reflects the growing time jobs wait for HPS.

Compute-node and HPS-storage utilization are visible in Figures 3c and 3d, respectively. We calculate the total node utilization U across all n experiments, involving our ten workloads, using the formula below:

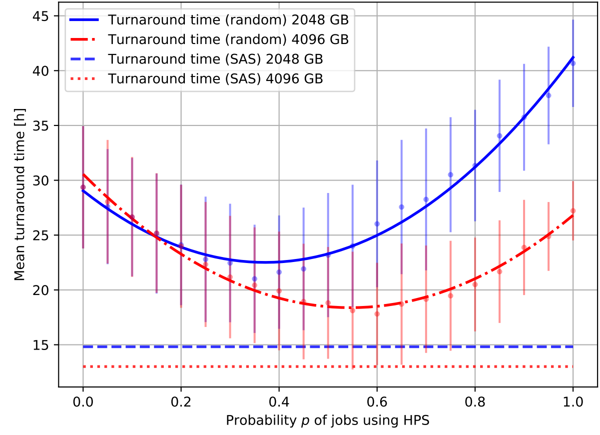
$$U = \frac{\sum_{i=1}^{n} u_i}{\sum_{i=1}^{n} a_i} \tag{3}$$

u_i denotes the used node hours and a_i the available node hours in the i-th experiment. The former is the sum of the product of job runtime and node allocation across all jobs in a workload. The latter is the product of workload makespan and the nodes available in the cluster. We determine the storage utilization analogously, just substituting storage space for nodes. As expected under random storage assignment, because increasing p raises the number of jobs using HPS, the HPS utilization grows steadily. At the same time, the node utilization experiences a dramatic slump, while an increasing number of jobs are idling in the queue, waiting for the limited amount of HPS storage to become available. On the other hand, SAS achieves remarkably high resource utilization: around 80% for compute nodes and around 70% for HPS storage.

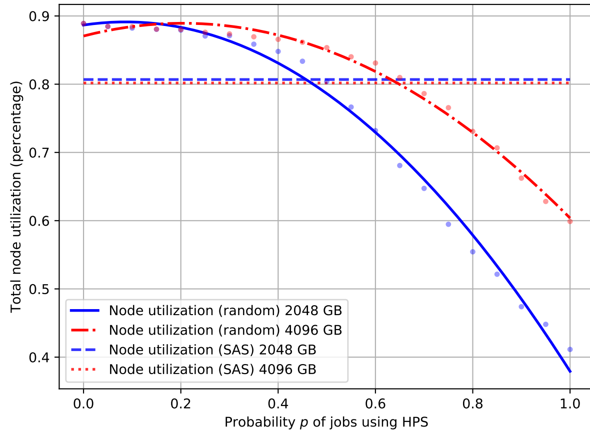
As discussed in Section IV-A, we assume R_{stage} to be five times bigger than R_{lps} because of the fast sequential data transfer rate between *LPS* and *HPS*. Figure 4 shows the impact of the staging bandwidth in relation to R_{lps} on makespan and turnaround time. Specifically, we consider $R_{stage} = k \cdot R_{lps}$ with $k \in \{1, 2, 10\}$. We saw already in Figures 3a and 3b the results for $k = 5$, the ratio we deem most realistic. With R_{stage} being



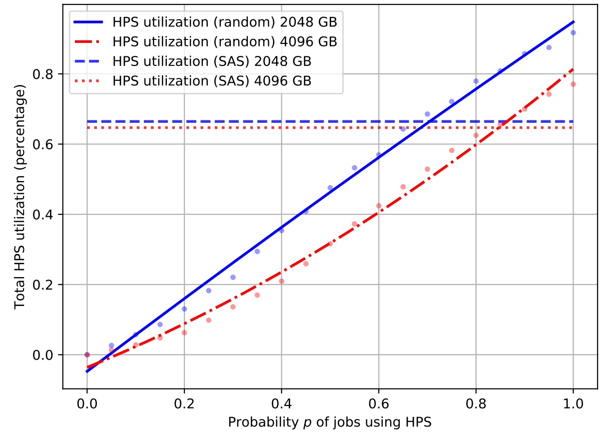
(a) Mean makespan



(b) Mean turnaround time



(c) Total compute-node utilization



(d) Total HPS utilization

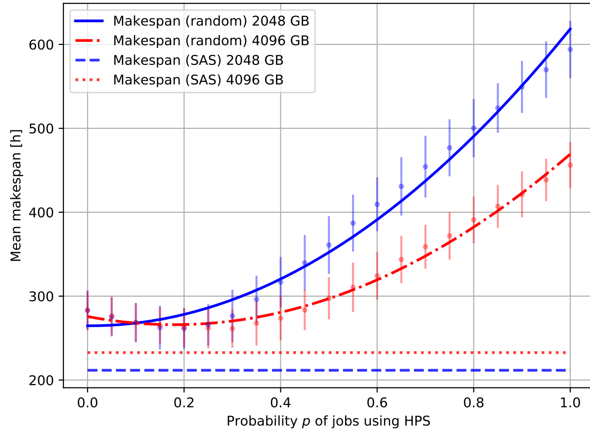
Figure 3. Comparison between our storage-aware scheduler (SAS) and random storage assignment, assuming two different storage capacities for HPS (2048 GB and 4096 GB). We show results for random storage assignment with probabilities p between 0 and 1 in steps of 0.05. The parameter p specifies the portion of jobs to be scheduled on HPS, with the precise selection of these jobs being randomized. $p=0$ corresponds to always using slow storage and $p=1$ to always using fast storage. To highlight the general trend, we fit the results measured for random storage assignment with a polynomial curve of second degree. The error bars depict the standard deviation of all results that we gathered while repeating our experiments for a single value of p. The makespan and turnaround times represent the arithmetic mean across ten workloads and ten repetitions for each value of p.

closer to R_{lps}, SAS still mostly outperforms random storage assignment although makespan and turnaround time tend to be higher than in a scenario where $R_{stage} \gg R_{lps}$. Remarkably, if the staging bandwidth is low then increasing the capacity of high-performance storage prolongs makespan and turnaround time unexpectedly. This is because the staging bandwidth is shared among all jobs attempting to stage their data, which makes it a bottleneck if many jobs stage simultaneously. The more fast storage is available the more jobs need staging. Consequently, properly balancing the amount of fast storage and the staging bandwidth is essential to the efficiency of our two-tier-storage scheduler. Determining the optimal staging bandwidth for a given amount of fast storage will be the subject of future studies.

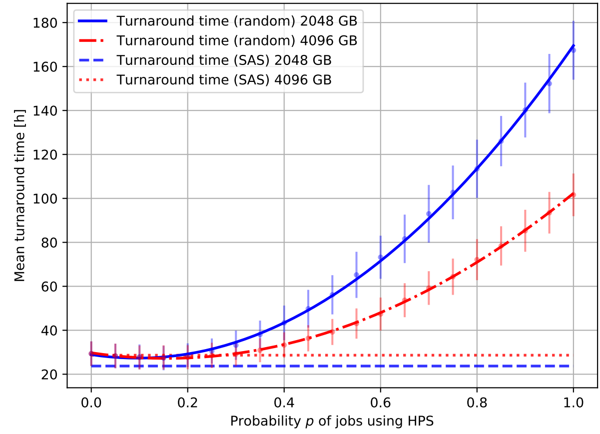
B. Impact of inaccurate I/O requirements

Up to now, we have always assumed that users provide accurate estimates of their job's I/O requirements. Now, we analyze how well SAS can tolerate inaccurate I/O requirement specifications. For this purpose, we ran a new set of experiments where the user input for each job follows a normal distribution $N(\mu,\sigma^2)$ around the correct value.

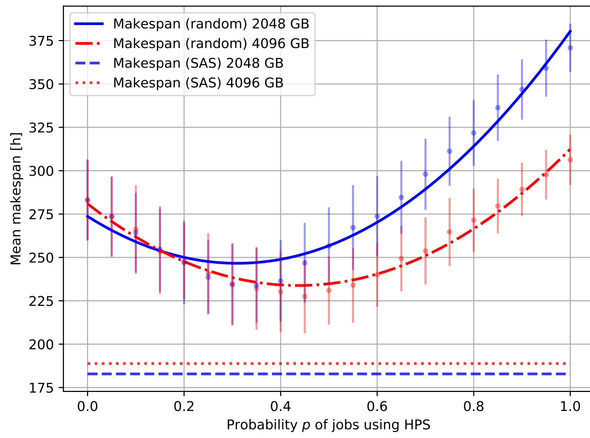
We evaluate the inaccuracy of user input based on the factor θ, where $\mu = data_i$ and $\sigma = data_i \cdot \theta$. The factor θ indicates how much the values in the normal distribution differ from the correct value. In other words, θ reflects the relative standard deviation of user input from the correct value. A similar distribution is considered for $input_i$ and $output_i$. The deviation from the perfect estimate is shown through scaling θ from 0 to 1.0 in steps of 0.025, with $\theta = 0$ representing accurate estimation. Like in our comparison with random-



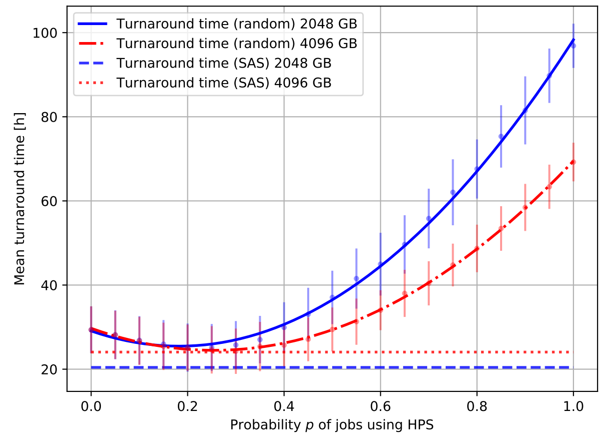
(a) Mean makespan for $k = 1$



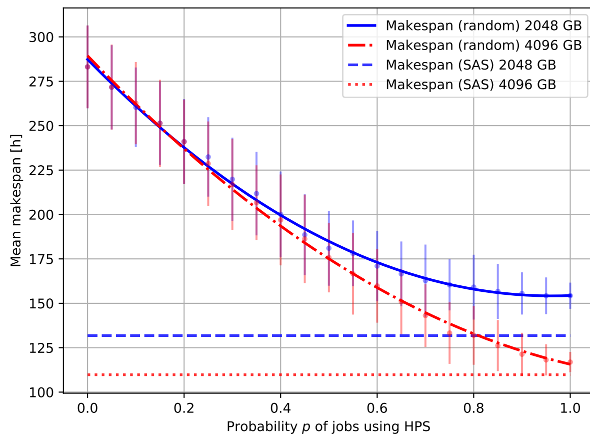
(b) Mean turnaround time for $k = 1$



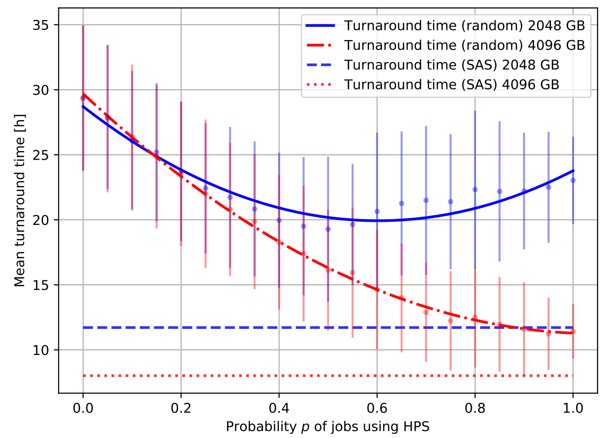
(c) Mean makespan for $k = 2$



(d) Mean turnaround time for $k = 2$

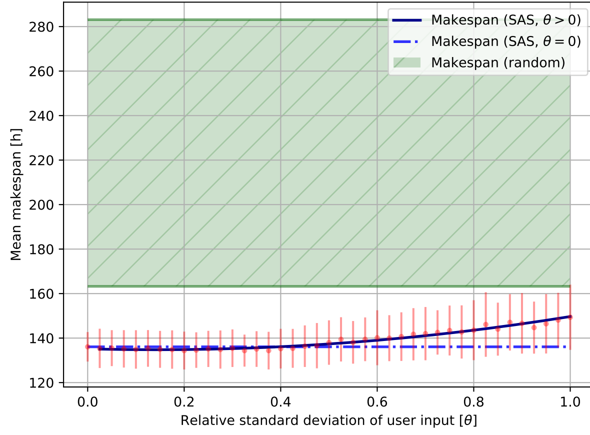


(e) Mean makespan for $k = 10$

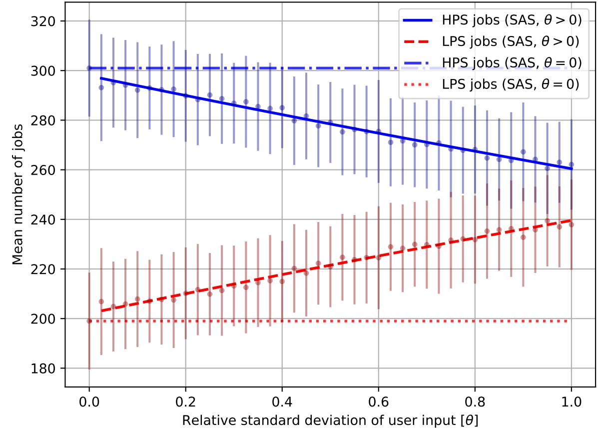


(f) Mean turnaround time for $k = 10$

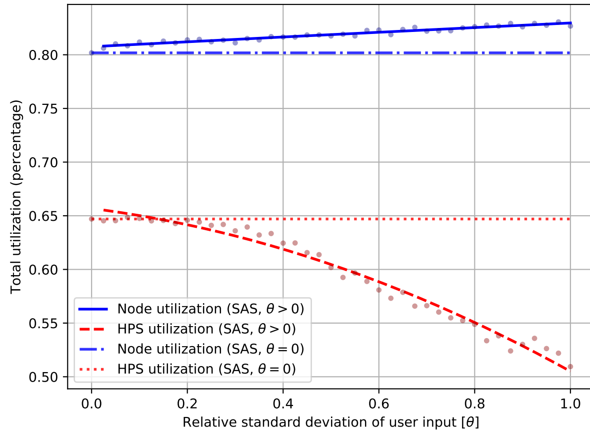
Figure 4. Impact of the staging bandwidth in relation to the LPS bandwidth. We compare our storage-aware scheduler (SAS) with random storage assignment, assuming $R_{stage} = k \cdot R_{lps}$ with $k \in \{1, 2, 10\}$. To highlight the general trend, we fit the values measured for random storage assignment with a polynomial curve of second degree. The error bars depict the standard deviation of all results that we gathered while repeating our experiments for a single value of p. The figures represent the arithmetic mean across ten workloads and ten repetitions for each value of p.



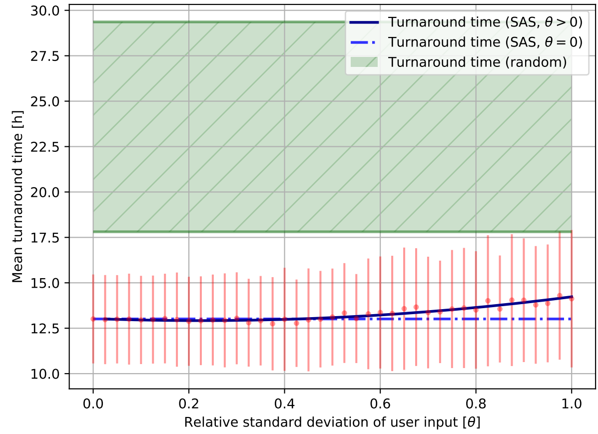
(a) Mean makespan



(b) Mean number of jobs assigned to each storage tier



(c) Total node and storage utilization



(d) Mean turnaround time

Figure 5. Effects of inaccurate user input shown for an HPS storage capacity of 4096 GB. The green (shaded) areas in Figures 5a and 5d cover the entire probability range of random storage assignments. User inputs are drawn randomly based on a normal distribution $N(\mu, \sigma^2)$ around the correct value, with $\mu = bytes$ and $\sigma = bytes \cdot \theta$. The factor θ, shown from 0 to 1.0 in steps of 0.025 on the x-axis, steadily increases the deviation from the correct user input. To highlight the general trend, we fit the values measured for various values of θ with polynomial curves of first and second degree, respectively. The error bars depict the standard deviation of all results we gathered while repeating our experiments for a single value of θ. With the exception of utilization, these figures represent the arithmetic mean across ten workloads and ten repetitions for each value of θ. The total utilization is the accumulated resource usage divided by the total amount of available resources across all experiments.

storage-assignment, we strove to increase statistical significance by repeating each simulation ten times for each factor θ and taking the average.

Figure 5a compares the makespans resulting from perfect user estimation ($\theta = 0$) with those resulting from inaccurate estimation ($\theta \neq 0$). The simulations were carried out for an HPS storage capacity of 4096 GB. The ratio between R_{stage} and R_{lps} is again five. The green (i.e., shaded) area in this figure covers random storage assignment with the full range of probabilities $(0-1)$ for comparison. As can be observed, even in the view of inaccurate user input in the range specified by the normal distribution, SAS still outperforms random storage assignments.

Figures 5b and 5c illustrate the impact of inaccurate user requirements on resource utilization. Figure 5b shows that the number of jobs scheduled on each storage tier is directly affected by changing accuracy of the user input. As the accuracy decreases, the number of jobs that use HPS is being reduced while the number of jobs that use LPS is being increased. This behavior leaves the HPS tier underutilized, as shown in Figure 5c. On the other hand, the influence on utilization of compute nodes is low, which is to be expected because low HPS utilization means less waiting time for limited HPS storage.

Figure 5d compares the mean turnaround times of SAS with those of random storage assignment when varying θ. With small deviations of the user input, the scheduler behaves partly unstable and we can observe a short interval in which the results of SAS become slightly worse than the best random storage assignments. But compared to the wide dispersion

of the random storage assignment results spread across the green (dark) area, the results of SAS are consistently positioned at the lower end.

Since users might simply omit the specification of storage requirements, we finally need to discuss the behavior of SAS in the absence of user inputs. Having the value of 0 for all user inputs leads to $T_i^{lps} = T_i^{hps}$, and in such a case Algorithm 2 (*try-start-storage-aware*()) only chooses the LPS tier. In other words, by providing no input, the scheduler skips the HPS tier altogether to avoid waiting for limited HPS storage. This is a fair scheduler decision because although SAS adds no improvement, at least it does not show worse performance than having no HPS in the model.

One way of discouraging users from manipulating scheduler decisions with deliberately wrong I/O requirement specifications would be to monitor I/O and storage usage and penalize inaccurate requirement statements. However, a detailed treatment of such policies is outside the scope of this article.

VI. Conclusion

The emergence of new storage technologies such as NVMe can pave the way to overcoming the challenges of running I/O-bound applications on HPC systems. Although the idea of tiered storage is not new, the upcoming high-performance storage will have revolutionary impact on the runtimes of large-scale data-intensive applications, as we have already seen with burst-buffer technologies. In this work, we introduced a novel scheduling algorithm that considers a high-performance storage tier with limited capacity alongside traditional HDD back-end storage within the same cluster infrastructure. We showed in simulations how delegating the decision as to which storage type a job should use to the scheduler improves makespan and turnaround times. However, we also learned that increasing the amount of available fast storage beyond a certain level does not necessarily lead to faster turnaround times unless it is backed by higher staging bandwidth. Moreover, we found that our scheduling algorithm is to some degree even tolerant of inaccurate user estimates of I/O requirements. An extension of the scheduler to cover more than two storage tiers is straightforward.

Inspired by the encouraging outcome of this study, we have started to implement our two-tier storage scheduler inside the resource manager Slurm [22]. Another logical next step is the refinement of current accounting schemes to properly reflect the performance of the storage system the scheduler ultimately assigns.

Acknowledgment

This research has received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under Grant Agreement No. 720270 (Human Brain Project SGA1) and Grant Agreement No. 785907 (Human Brain Project SGA2). The underlying simulations were carried out on Lichtenberg, the high-performance computer of Technische Universität Darmstadt. Finally, we would like to express our gratitude to the Batsim developers at INRIA for their support. In particular, we thank Pierre-François Dutot, Michael Mercier, Millian Poquet, and Olivier Richard.

References

[1] Jorge Manuel Gomes Barbosa and Belmiro Daniel Rodrigues Moreira. "Dynamic Job Scheduling on Heterogeneous Clusters". In: *Proc. of the 8th International Symposium on Parallel and Distributed Computing (ISPDC'09)*. IEEE, 2009, pp. 3–10.

[2] Wahid Bhimji, Debbie Bard, Melissa Romanus et al. "Accelerating Science with the NERSC Burst Buffer Early User Program". In: *Proc. of the Cray User Group (CUG2016)*. 2016.

[3] Rajkumar Buyya and Manzur Murshed. "Gridsim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing". In: *Concurrency and Computation: Practice and Experience*. 14.13-15 (2002), pp. 1175–1220.

[4] Pierre-François Dutot, Michael Mercier, Millian Poquet and Olivier Richard. "Batsim: a Realistic Language-Independent Resources and Jobs Management Systems Simulator". In: *Proc. of the 20th Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2016, pp. 178–197.

[5] Dror G. Feitelson. "Packing Schemes for Gang Scheduling". In: *Proc. of the 2nd Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1996, pp. 89–110.

[6] Ian Foster, Carl Kesselman, Craig Lee et al. "A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation". In: *Proc. of the 7th International Workshop on Quality of Service (IWQoS'99)*. IEEE, 1999, pp. 27–36.

[7] Nicholas Hazekamp, Nathaniel Kremer-Herman, Benjamin Tovar et al. "Combining Static and Dynamic Storage Management for Data Intensive Scientific Workflows". In: *IEEE Transactions on Parallel and Distributed Systems* 29.2 (2018), pp. 338–350.

[8] Jan Heichler. *An Introduction to BeeGFS v2.0*. Whitepaper. ThinkParQ, 2018. URL: http://www.beegfs.com/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf (visited on 19th Mar. 2019).

[9] Stephen Herbein, Dong H. Ahn, Don Lipari et al. "Scalable I/O-Aware Job Scheduling for Burst Buffer Enabled HPC Clusters". In: *Proc. of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 69–80.

[10] Florin Isaila, Jesus Carretero and Rob Ross. "Clarisse: A Middleware for Data-Staging Coordination and Control on Large-Scale HPC Platforms". In: *Proc. of the 16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016, pp. 346–355.

[11] Dalibor Klusáček and Hana Rudová. "Alea 2: Job Scheduling Simulator". In: *Proc. of the 3rd International Conference on Simulation Tools and Techniques*. ICST, 2010, p. 61.

[12] Andreas Knüpfer, Christian Rössel, Dieter an Mey et al. "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir". In: *Proc. of the 5th International Workshop on Parallel Tools for High Performance Computing*. Springer, 2012, pp. 79–91.

[13] Tevfik Kosar and Mehmet Balman. "A New Paradigm: Data-Aware Scheduling in Grid Computing". In: *Future Generation Computer Systems.* 25.4 (2009), pp. 406–413.

[14] David A. Lifka. "The ANL/IBM SP Scheduling System". In: *Proc. of the 1st Workshop on Job Scheduling Strategies for Parallel Processing.* Springer, 1995, pp. 295–303.

[15] Arun Ramakrishnan, Gurmeet Singh, Henan Zhao et al. "Scheduling Data-Intensive Workflows onto Storage-Constrained Distributed Resources". In: *Proc. of the 7th International Symposium on Cluster Computing and the Grid (CCGrid).* IEEE, 2007, pp. 401–409.

[16] Aamer Shah, Felix Wolf, Sergey Zhumatiy and Vladimir Voevodin. "Capturing Inter-Application Interference on Clusters". In: *Proc. of International Conference on Cluster Computing (CLUSTER).* IEEE, 2013, pp. 1–5.

[17] Konstantin Shvachko, Hairong Kuang, Sanjay Radia and Robert Chansler. "The Hadoop Distributed File System". In: *Proc. of the 26th Symposium on Mass Storage Systems and Technologies (MSST).* IEEE, 2010, pp. 1–10.

[18] Tim Süß, Nils Döring, Ramy Gad et al. "Impact of the Scheduling Strategy in Heterogeneous Systems that Provide Co-Scheduling". In: *Proc. of the 1st COSH Workshop on Co-Scheduling of HPC Applications.* IOS Press, 2016, p. 37.

[19] Atsuko Takefusa, Hidemoto Nakada, Tomohiro Kudoh and Yoshio Tanaka. "An Advance Reservation-Based Co-Allocation Algorithm for Distributed Computers and Network Bandwidth on QoS-Guaranteed Grids". In: *Proc. of the 15th Workshop on Job Scheduling Strategies for Parallel Processing.* Springer, 2010, pp. 16–34.

[20] Dan Tsafrir, Yoav Etsion and Dror G. Feitelson. "Backfilling Using System-Generated Predictions Rather than User Runtime Estimates". In: *IEEE Transactions on Parallel and Distributed Systems.* 18.6 (2007), pp. 789–803.

[21] Lipeng Wan, Zheng Lu, Qing Cao et al. "SSD-Optimized Workload Placement with Adaptive Learning and Classification in HPC Environments". In: *Proc. of the 30th Symposium on Mass Storage Systems and Technologies (MSST).* IEEE, 2014, pp. 1–6.

[22] Andy B. Yoo, Morris A. Jette and Mark Grondona. "Slurm: Simple linux utility for resource management". In: *Proc. of the 9th Workshop on Job Scheduling Strategies for Parallel Processing.* Springer, 2003, pp. 44–60.

[23] Kenneth Yoshimoto, Patricia Kovatch and Phil Andrews. "Co-Scheduling with User-Settable Reservations". In: *Proc. of the 11th Workshop on Job Scheduling Strategies for Parallel Processing.* Springer, 2005, pp. 146–156.

[24] Gong Zhang, Lawrence Chiu and Ling Liu. "Adaptive Data Migration in Multi-Tiered Storage Based Cloud Environment". In: *Proc. of the 3rd International Conference on Cloud Computing (CLOUD).* IEEE, 2010, pp. 148–155.