# On Performance Resilient Scheduling for Scientific Workflows in HPC Systems with Constrained Storage Resources

Yang Wang
Shenzhen Institutes of Advanced Technology
Chinese Academy of Science, Shenzhen, China
yang.wang1@siat.ac.cn

Wei Shi
Faculty of Business and Information Technology
University of Ontario Institute of Technology
Ontario, Canada
wei.shi@uoit.ca

Eduardo Berrocal
Department of Computer Science
Illinois Institute of Technology
Chicago, USA
eberroca@hawk.iit.edu

## ABSTRACT

Although the storage capacity is rapidly increasing, the size of datasets is also ever-growing, especially for those workflows in HPC that perform the parameter sweep studies. Consequently, the deadlock caused by the storage competition between concurrent workflow instances is still a major pragmatic concern and storage management remains important for high performance and throughput computing. In practice, there are various ways to this issue, ranging from *admission control* to *deadlock resolution*. Despite being a simple solution, the admission control is conservative and not space efficient to storage utilization. Therefore, in this paper, we study the performance of the deadlock resolution approach by proposing a resource allocation algorithm which is performance resilient to the workflows characterized by different features. The algorithm is designed based on our previous result, called *DDS*, which takes advantages of the dataflow information of the workflow to resolve deadlock based on detection&recovery principle. We improve *DDS* to allow it to not only resolve the deadlock but also overcome the performance anomaly, a not yet investigated problem in our previous studies. We thus called the improved algorithm performance-resilience algorithm, denoted as *DDS*$^+$. The studies in this paper can be viewed as a follow-up research on *DDS* and show the performance behavior of the improved algorithm in various conditions. Therefore, the results in this paper are more useful to adapt *DDS*$^+$ to the workflows with different characteristics in reality while keeping the performance stable.

## Categories and Subject Descriptors

D.4 [**Software**]: Operating Systems; D.4.8 [**Performance**]: Modeling and prediction—*simulation*

## General Terms

Performance


*ScienceCloud'15,* June 16, 2015, Portland, Oregon, USA.

DOI: http://dx.doi.org/10.1145/2755644.2755646.

## Keywords

workflow scheduling; workflow management; performance anomaly; performance resilience; storage constraint; deadlock detection

## 1. INTRODUCTION

A complex scientific workflow is usually composed of a variety of standalone application components in terms of control- or data-dependencies to carry out a well-defined scientific computing process. In reality, a scientific workload often consists of multiple instances of the same workflow, with each instance acting on independent input files or different initial parameters, either for parallel data processing or parameter-based studies. For example, the Sextractor [2] pipeline runs, by NCSA astronomers, `2611` instances on the DPOSS [8] dataset with each pipeline instance accessing a different `1.1GB` image to search for bright galaxies. Although maximizing instance concurrency can optimize such computation, it is not always available in practice due to a variety of resource constraints.

In this paper, we are particularly interested in the storage constraints. On one hand, with the advance of high-performance computing (HPC) in big science, the size of involved datasets is ever-growing to outpace the increasing rates of any affordable storage capacity. On the other hand, some practical and system policies still exist in certain situations to limit the freedom of storage uses. For example, in the cloud computing, whose resources are typically provisioned based on "pay-as-you-go" billing model, fully utilizing the provisioned resources in general and storage in particular is essential to achieve cost-effective computation when limited budget for the resources is a restriction. Consequently, the storage space management remains important for high performance and throughput of the workflow computation in HPC. In our case, a concerned problem is the deadlock that is caused by the competition for the storage resources between concurrent workflow instances.

In practice, there are various ways to the deadlock problem, ranging from *admission control* to *deadlock resolution*. Despite being a simple solution, the admission control is conservative and not space efficient to storage utilization. Therefore, we advocated the deadlock resolution approach and proposed *DDS*, a deadlock detection-based scheduling algorithm for workflow computation in HPC systems with storage constraints [21]. The algorithm takes advantages of the dataflow information of the workflow to speculatively execute each instance whenever the instantaneous storage space is sufficient for some job executions (but not sufficient for the whole instance) and perform the *rollback* operation on the se-

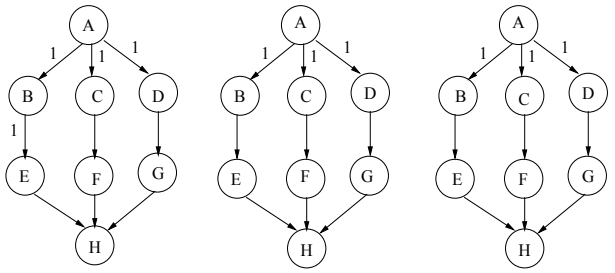



**Figure 1: An example of performance anomaly where three instances are executed concurrently, each with one unit data for each input and output files. Given total storage budget of 10 units, if resource allocation is not well designed, only the left most instance can proceed and the other two have to be blocked, leading to so-called performance anomaly.**

lected in-progress instances whenever deadlock is detected. With some simple yet practical assumptions we showed the advantages of *DDS* in performance over the classic Banker's algorithm. However, the studies still fall short of sufficient investigation on the workflows with more different features, especially, the reaction to performance anomaly during the workflow computation. Thus, we leave some unknown facets of the algorithm when deploying it in reality.

Performance anomaly could happen when multiple workflow instances run concurrently, competing for the finite resources. It is possible as we show in Figure 1 that a large number of instances may become blocked (due to the unavailability of requested resources) while retaining the occupied resources, leaving just a small number of instances to make a slow progress. Although there is no deadlock, the overall performance of the computation is dramatically degraded. Performance anomaly is usually not a common case, more often than not, a corner case, for an algorithm in its practical use. However, it introduces uncertainty to the performance of the algorithm, rendering the algorithm unstable with respect to its diverse inputs. Therefore, the strategies to improve *DDS* with an ability to overcome the anomaly while maintaining or even improving the performance (i.e., performance resilience) is highly desired. However, most of existing researches only focus on the anomaly or fault detection via online or off-line techniques to model or predict the performance of the workflows instead of addressing the anomaly at runtime directly without using modeling [18, 12, 1].

The studies in this paper can be viewed as follow-up research to compensate for this shortage. We improve the algorithm with a *rollback operation* to enable it to not only resolve the deadlock but also overcome the performance anomaly. We thus call this algorithm *DDS*$^+$. The essence of this improvement is simple; it allows the scheduler to selectively rollback the instance with minimum number of completed jobs at each time when a running job is finished. As such, it does not need to monitor the instances' status whereby the rollback decision is made. Although, this strategy could intuitively incur significant overhead due to a large number of re-computed jobs. However, our finding is opposite to this intuition. On the contrary, *DDS*$^+$ not only addresses the performance anomaly but also for certain workflows shows some performance advantages over *DDS* where no rollback operation for the performance anomaly is involved. This is also the reason we call the improved algorithm *performance resilience algorithm*.

In this paper, we present simulation-based evidence to show how *DDS*$^+$ behaves to be performance resilient with respect to different features of the workflows, and as well, the performance anomaly in the computation. Our results would be very useful for the algorithm to adapt to the workflow computation in reality, especially, when cloud platforms are leveraged to achieve cost-effective computation.

The rest of the paper is organized as follows: in the next section, we discuss some related work. We introduce the computation model in Section 3 and outline the *DDS*$^+$ algorithm in Section 4. Then, Section 5 follows to present our simulation results. In the last section, we review lessons learned and propose some future work.

## 2. RELATED WORK

With the awareness of the continued growth of today's extreme-scale datasets in HPC [11, 3, 15, 7], the interest in scheduling workflow computation in storage-constrained systems is ever-increasing. In this section, we first survey some related work in storage-aware workflow scheduling algorithms [5, 16, 19, 6], and then review several state-of-the-art results related to performance resilience in workflow scheduling.

BAD-FS [5] is one of the early efforts in *capacity-aware scheduling* in HPC. It leverages a centralized batch scheduler to allocate storage volumes to the jobs from multiple workflow instances so that storage overflowing or cache thrashing can be avoided. To this end, BAD-FS identifies five possible data allocation strategies which not only influences the execution path of the workloads [4] but also *prevent* deadlock (not stated explicitly by the authors). However, these strategies are unable to make the best use of storage resources for performance optimization. On the other hand, all of their allocation strategies are only designed for batch-pipelined workflows with a sequential structure and thus might not effectively work on the workflows with other structures like those examined in this paper.

Ramakrishnan *et al.* address the later issue by considering the scheduling of data-intensive workflows with more general structures onto a set of distributed storage-constrained compute node [16]. Their basic approach is to add a *cleanup job* for each data file when that file is no longer needed by other jobs in the workflow or when it has already been staged out to some permanent storage. The garbage files are deleted (also called *garbage collection*) in time, and the amount of storage used for the workflow can be reduced significantly. To further reduce the overhead of the large number of cleanup jobs, they also implement a heuristic that uses a single cleanup job for removing multiple files.

A variant of this approach is presented in [19] where Singh *et al.* study the issue of optimizing disk usage in scheduling of large-scale data-intensive workflows onto distributed compute nodes, each with limited storage resources. Their approach is two-fold. First, like [16], they also minimize the amount of space a workflow requires during execution by removing data files at runtime when these files are no longer needed. Second, they demonstrate that workflows may have to be restructured to reduce the overall data footprint of the workflow, which is unique to their approach.

Recently, Chen *et al.* [6] studied the same problem as Ramakrishnan *et al.*'s in a similar context with a storage-aware scheduling but adopted a more elegant algorithm to partition the workflow into a set of non-cross-dependent sub-workflow so that the potential deadlock due to the storage constraints could be prevented. However, similar to the previous studies, this work is also geared toward efficient scheduling of a single workflow instance to a set of distributed storage sites.

Although all the aforementioned studies focus on improving workflow applications on HPC platforms with constrained storage, they do not thoroughly consider the deadlock problem, instead, they either prevent it from happening by using some conservative alloca-

tion [5, 6] or adopt *Ostrich* strategy to resolve it manually [16]. On the other hand, the performance behaviors of these algorithms on some corner cases are also not known if the parameter space studies on the workflows are not performed.

Wang and Lu [22] concentrate on the efficient deadlock resolutions in scheduling scientific workflows on storage-constrained HPC systems. To this end, they design two deadlock avoidance algorithms by exploiting the dataflow dependency inherent in the scheduled workflow. Moreover, in follow-up research, they also present workflow-aware file system, called *WaFS*, to leverage the dataflow information to manage the constrained storage resources for scientific workflow computation in the cloud [23]. Our work can be combined with *WaFS* to extend it for the deadlock resolution.

Deelman and Chervenak discuss general issues in data managements for data-intensive scientific workflows, with data storage as a focused challenge [7]. In contrast, Pandey and Buyya [15] propose workflow scheduling algorithms on data grids with large number of replicated files that incorporate practical constraints in data grids, and identify that constrained storage is one of major challenges in workflow scheduling. However, none of these researches consider the performance resilience issue.

To our best knowledge, the studies on this issue in workflow computation is quite few. Most of existing works in this area either leverage online techniques or resort to off-line infrastructure to detect failure or execution anomaly in large-scale workflows whereby their performance model is established, [18, 12, 1]. Wang *et al.* [20] introduce *ACS*, which is an admission control scheme with deadlock resolutions to facilitate workflow scheduling in the cloud. Although this scheme can optimize the storage utilization, and sometimes, can get rid of the performance anomaly, it is not performance resilient as it fails to address this problem in general sense. Compared to the existing results, our work focuses purely on the workflow scheduling with performance resilience.

# 3. COMPUTATION MODEL

## 3.1 Workflow Model

We model a workflow as a *workflow graph*, a weighted DAG *directed acyclic graph* $G(V,E)$, where $V$ is a set of nodes and $E$ is a set of edges. A node in the DAG represents a *job* which in turn is a program that must be executed in sequential order without preemption. The weight of a node is called the computation cost. An edge represents the communication in terms of dataflow (i.e. write/read file) via the underlying file system from the source node to the destination node; its weight indicates the file size. The precedence constraints of a DAG dictate the execution orders of the nodes in the sense that a node cannot begin execution until all its input files have arrived and no output files are available until the job has finished and at that time all output files are simultaneously accessible to its destination job.

A *workload* consists of multiple instances of the same workflow, with each instance having its own node and edge weights. The node and edge weights as well as the shape of the workflow are provided by users and not changed during the computation. Since in reality the multiple instances are usually created for parameter sweep study, it is reasonable to assume that each edge has roughly the same weight in different instance graphs of the same workflow.

Without loss of generality, single source and sink nodes are assumed in the DAG. These two nodes can be viewed as the jobs in the workflow that stage in the initial input data and stage out the result output data respectively. As such, the net storage after the workflow computation is zero if no intermediate data products are maintained.

**Algorithm 1** $DDS^+$ algorithm

```
1: procedure DDS+(I, j)
2:     if (DDS(I, j) = false) then
3:                         ▷ no deadlock, rollback for performance anomaly
4:         if (schedRollbackInst(I)) then
5:             rollbackInst(I)
6:         end if
7:     end if
8: end procedure
```

## 3.2 Execution Model

During the execution of a workflow instance, the life cycle of a job may experience several states. Initially, all the jobs in a workflow instance are in *blocked* state. A job becomes *free* if it has no parent jobs or all its parent jobs have finished. Every free job can be scheduled but only those who have storage space to accommodate their output data set can enter *ready* state for execution. Otherwise, they will be in *pending* state waiting for the storage. Of course, as soon as the required storage is available, the jobs in the pending state can be changed to the ready state for execution again. The jobs in *running* state are never stopped until they complete the computation. After a job has completed, it enters *done* state. A completed job will release the storage space of its input data set only, which can be reclaimed for other jobs' executions, but keeping the storage for the output data set for the later job use.

Our model is deterministic, at least to the extent that the time and storage space required by any job as well as the data dependencies among the jobs are pre-determined and remain unchanged during the computation.

**Algorithm 2** Rollback algorithm

```
1: procedure SCHEDROLLBACKINST(&I)
2:                     ▷ inst. with the smallest # of done jobs in waitReqQ
3:     J ← waitResQ.get_min()
4:                                   ▷ I is not the only active instance
5:     if (J ≠ I ⋀ waitResQ(J) ≠ ∅) then
6:         return true
7:     else
8:         return false
9:     end if
10: end procedure
```

# 4. $DDS^+$: A PERFORMANCE-RESILIENCE ALGORITHM

In this section, we introduce $DDS^+$, an improved deadlock detection algorithm by extending *DDS* [21] with resilience to performance anomaly. The essence of this algorithm is to leverage $DDS^+$ to resolve the detected deadlock while picking up a selected instance to rollback to release its occupied storage when there is no deadlock detected. *DDS* allows a workflow instance to speculatively execute if the storage is sufficient for some jobs of the instances, but not sufficient for the whole workflow instance. The algorithm expects that as the computation proceeds, the occupied storage (by some instances) can be reclaimed to contribute to the

**Algorithm 3** Workflow Batch Scheduling

```
1:  procedure Sched
2:                                              ▷ When a job j' completed
3:      Budget ← Budget + j.rel                 ▷ release budget
4:                          ▷ Move jobs in waitResQ to readyQ if possible
5:      ......                                  ▷ codes are omitted for simplicity
6:      I ← getInstance(j')
7:      for j ∈ I do
8:          if (j's dependency is resolved) then
9:              if (Budget ≥ j.req) then
10:                 readyQ[I].add(j)
11:                 Budget ← Budget − j.req
12:             else
13:                 waitResQ[I].add(j)
14:             end if
15:         end if
16:     end for
17:     for (I ∈ readyQ) do
18:         for (j^i ∈ readyQ[I]) do
19:                                 ▷ readyQ[I] contains all jobs of instance I
20:                                   ▷ alg could be Banker's, DDS or DDS+
21:             if (Deadlock_Resolver(I, j^i, alg)) then
22:                 schedule j^i                ▷ Schedule job j^i
23:                 readyQ[I] ← readyQ[I] \ {j^i}
24:             end if
25:         end for
26:     end for
27: end procedure
```

computation progress as a whole. To resolve the deadlock, *DDS* rollbacks some later instances based on certain criteria (e.g., least done job first) to reclaim the occupied storage resources and re-allocate them to an earlier blocked instance. It is possible that the given storage resources are insufficient even for a single instance. As such, rollback is always possible in our scenario to resolve the deadlock problem. *DDS*$^+$ can detect this case instead of falling into endless rollback operations. The details of *DDS* can be found in [21]. Here, we only focus on *DDS*$^+$ itself (Algorithm 1) as well as its invoked subroutine *instance rollback algorithm* (Algorithm 2). Note that the main scheduling algorithm which calls *DDS* or *DDS*$^+$ as a deadlock resolver is shown in Algorithm 3.

As shown in Algorithm 3, the completion of a job may free new jobs (i.e., a job is free if all its parent jobs are completed) into either the *ReadyQ* queue if the required storage is available or, otherwise, the *waitResQ* queue, waiting for the required storage (Lines 7-16). Then the algorithm enumerates the ready queues and select schedule the jobs in each selected instance after the deadlock is resolved (Lines 17-25). The *deadlock resolver* is a driver program that simply invokes the pre-defined deadlock resolution algorithms such as the Banker's algorithm, *DDS* and *DDS*$^+$. This strategy, though simple and efficient, could possibly incur performance anomaly when scheduling certain workloads due to the large amount of inactive storage held by the blocked instances without any contribution to the progress of the computation (e.g., fork&join workloads in our studies). We fix this problem by performing a rollback scheduling procedure (i.e., *schedRollbackInst()*) after *DDS* checking in *DDS*+ at the end of freeing each job as shown in Algorithm 1. This procedure is responsible for picking up an instance in the *waitResQ* to rollback. Currently, our strategy is simply to pick up the instance who is not the only running instance, yet has the smallest number of completed jobs (Algorithm 2). As such the algorithm is efficient, which can finish the operation within logarithmic time (Line 3 in Algorithm 2).

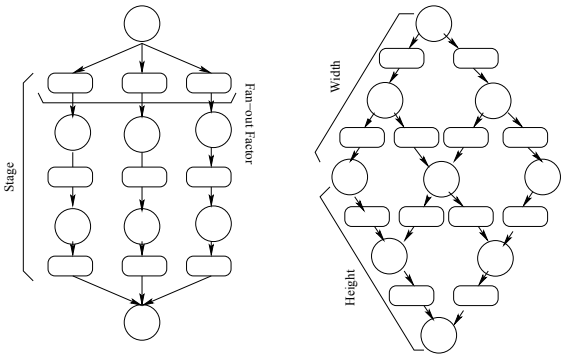



**Figure 2: Benchmark workflow graphs: circle represents job and rounded rectangle represents input/output file. The fork & join (a) is characterized by fan-out factor and the number of stages, whereas the lattice (b) is characterized by height and width.**

Intuitively, this extra scheduling step may increase the number of re-computed jobs and adversely affect the overall performance. However, our simulation results show the opposite effect that this strategy not only addresses the performance anomaly problem in fork&join workloads but also improve the performance of other (e.g., lattice) workloads. More details are shown in the following part.

# 5. PERFORMANCE STUDIES

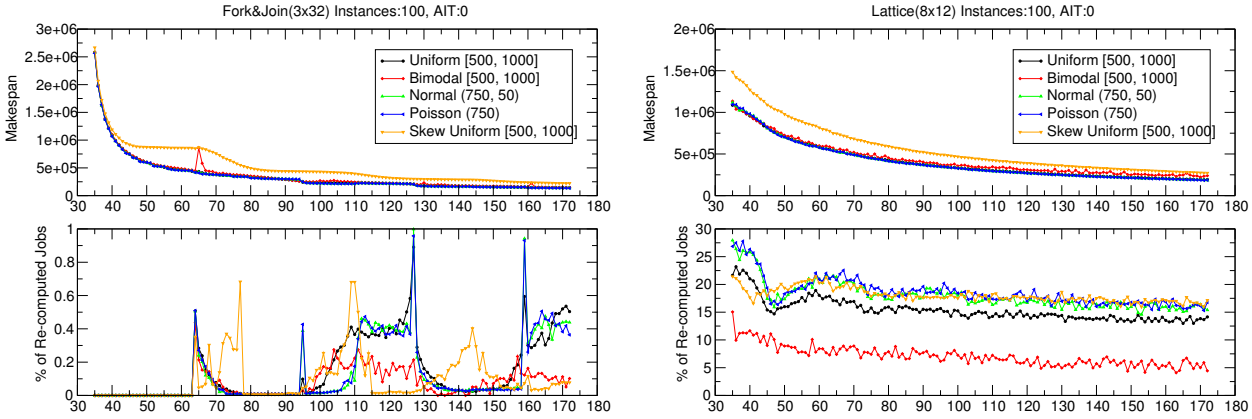
## 5.1 Experimental Setup

We simulate the computation model presented in Section 3 and implement a scheduler using the simulation package SMURPH [9]. The scheduler accepts the dataflow DAG from the user submitting the workflow instances and follows the execution model to manage the submitted workloads. It schedules each job based on the answer from a *deadlock resolver* which runs *DDS*$^+$ for both deadlock resolution and performance resilience.

Like in [21], we continue to use the two representative structures, fork & join and lattice shown in Figure 2 as the benchmarks in our experiments since on the one hand, they represent a wide range of scientific workflows [10, 13, 14, 17], and on the other hand, they exhibit different degree of concurrency (DOC) patterns which make it easy to reason about the experimental results.

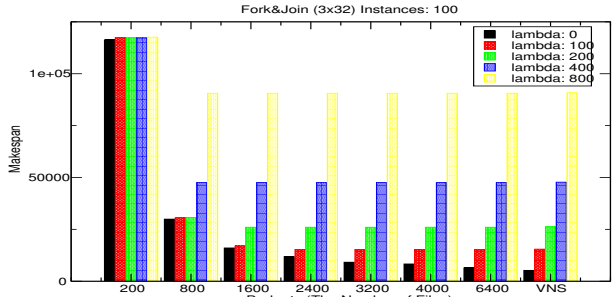
Since we intend to study the performance of the proposed algorithms in general cases, except for the representative DAG structures, we make no further assumption on any *a prior* knowledge of the jobs such as its job service time (JST) and file sizes. As such, for all investigated workflow structures (i.e., fork&join ($3 \times 32$) and lattice ($8 \times 12$)), we consider the job service times that are varied from 500 to 1000] time units in different distributions, while the input/output file sizes are either unit-based or uniformly distributed on $[1, 10]$ storage units, to reflect different situations in reality. For all the examined workflow DAGs, the maximum claim of the job can be computed in an efficient optimal way by exploiting the features of each graph.

Additionally, we also assume that an unbounded number of homogeneous compute nodes are available as constrained storage is only our concern, and thus the maximum DOC should never be constrained by the compute nodes. Finally, in each experiment, a total of 100 workflow instances are in the workload.

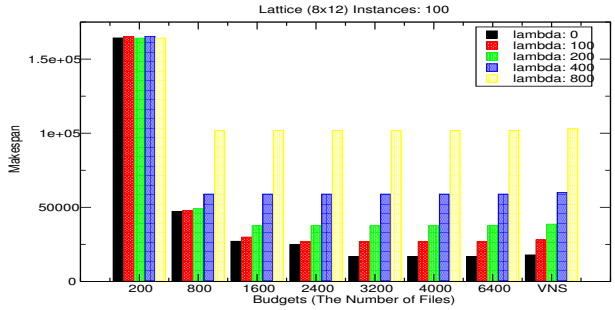



(a) fork&join (b) lattice

**Figure 4: Impact of storage budget vs. JST distribution for workload fork&join and lattice**




(a) fork&join




(b) lattice

**Figure 3: Impact of storage budget vs. instance average inter-arrival time for workload fork&join and lattice**

## 5.2 Simulation Results

In this section, we present our simulation results on diverse aspects of *DDS*$^+$. To this end, we first study *DDS*$^+$ itself with respect to its behaviors to diverse features of the workloads. Then we compare *DDS*$^+$ with the classic Banker's algorithm to show its performance advantage.

### *5.2.1 DDS*$^+$ *Studies*

We study *DDS*$^+$ from several aspects, ranging from the impact of storage budget and average inter-arrival time (AIT) on the makespan performance to the effects of improved rollback operation for performance resilience.

*Storage Budgets vs. Average Inter-arrival Time.*

We first investigate the impact of different storage budgets on the total makespan of *DDS*$^+$ on the both workloads. Each workload consists of 100 instances whose inter-arrival time follows exponential distribution. Figure 3(a) shows the results of fork&join workload. We found from this graph that both the storage budget and the instance average inter-arrival time have big impact on the workload makespan. When the storage budget is very limited (e.g., budget=200), such impact are dominated by the storage budget, irrespective of the different instance average inter-arrival time. It is easy to understand since the limited storage decreases the degree of concurrency and in the meanwhile increases the probability of instance rollbacks. However, as the storage budget increases, these adverse effects diminish, and the impact on the makespan are gradually dominated by the instance average inter-arrival time. This is reasonable since the degree of concurrency of less intense workloads is low, and thus reduces the simultaneous requirements for the storage. From this graph, we can conclude that the storage budget should be determined by the intensity of the workload. The higher the intensity of the workload, the more budgets should be given. When the intensity of the workload is low, more storage budgets cannot definitely guarantee the improvement of the makespan. The same conclusion is also true for the lattice workload (Figure 3(b)).

*Storage Budget vs. JST Distribution.*

Figure 4 shows the results of *DDS*$^+$ with different storage budgets on various JST distributions including *Uniform*, *Bi-modal*, *Normal*, *Poisson*, and *Zipf-like* distributions to reflect diverse situations in reality. The characteristics of fork&join workloads under these distributions are illustrated in Figure 5 where the total amount of workload, the average JST, as well as the length of critical path are obtained when $AIT = 0$ and normalized by the uniform distribution. The lattice workload also exhibits similar observations.

These results are obtained when the average instance inter-arrival time is zero. We observe from the left graph that as the storage budgets increases, all the makespans of the fork&join workloads with different JST distributions are reduced, especially when the storage budgets are highly limited. It demonstrates that the performance of the workload is very sensitive to the increment of the storage budgets when they are low. However, the performance gain is gradually lessened when the storage budgets increase over a certain value. These results are consistent with those of our previous experiments. We explain them as follows, when the storage bud-

gets are low, although there are lots of jobs who can execute concurrently, these jobs have to wait in the *waitResQ* queue for the required storage. As a result, most of these jobs execute sequentially, elongating the total makespan. However, with the growth of the budgets, such jobs can obtain more opportunities to execute concurrently, and thus, shortening the makespan. The performance improvement may result from two factors, the available storage and the degree of concurrency. Although the storage increases gradually, the degree of concurrency is not changed. This fact explains why the performance gain is gradually lessened.

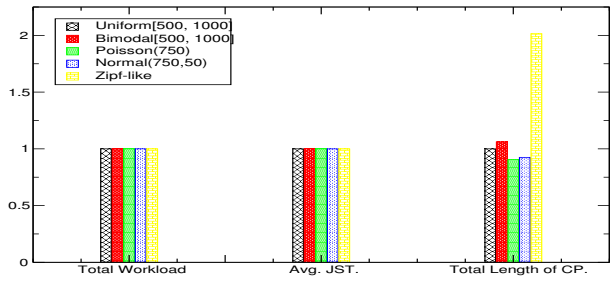



**Figure 5: Workload characteristics for fork&join under different JST distributions**

By comparing the shapes of both workflows, we further observed that for all storage budgets, all the JST distributions, except the Zipf-like, *DDS*$^+$ exhibits almost identical performance trends. In this sense, the performance of *DDS*$^+$ with different storage budgets are insensitive to the JST distributions. Since the Zipf-like distributions may result in some jobs with much higher JST than other jobs, and thus these jobs may block subsequent small jobs and elongate the critical path (Figure 5). As a consequence, the Zipf-like distribution always exhibit the worst performance among the investigated distributions.

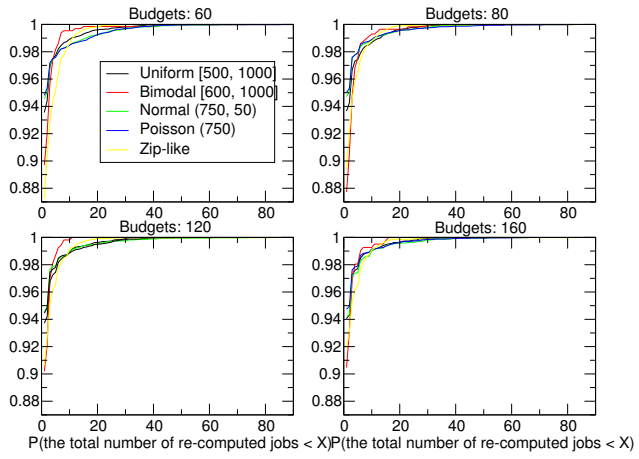



**Figure 6: Cumulative distribution functions (CDFs) of re-computed jobs in lattice workload (AIT=0) when the budget is 60, 80, 120 and 160 units.**

*Cost of Speculative Execution.*

Due to the speculative execution of the workflow instances, we can expect the performance loss with *DDS*$^+$. Unfortunately, such loss cannot be measured accurately since the rollback instances execute concurrently with other normal instances. To address this problem, we measure the percentage of the re-computed jobs to indirectly reflect the cost of the speculative execution. From the bottom graph of Figure 4(a), one can see that the percentage of the re-computed jobs is less than 1%, and all rollback instances have only one done job (not shown in the graph). Since the number of processors/hosts is not our bottleneck, we think the cost of the speculative execution is not significant.

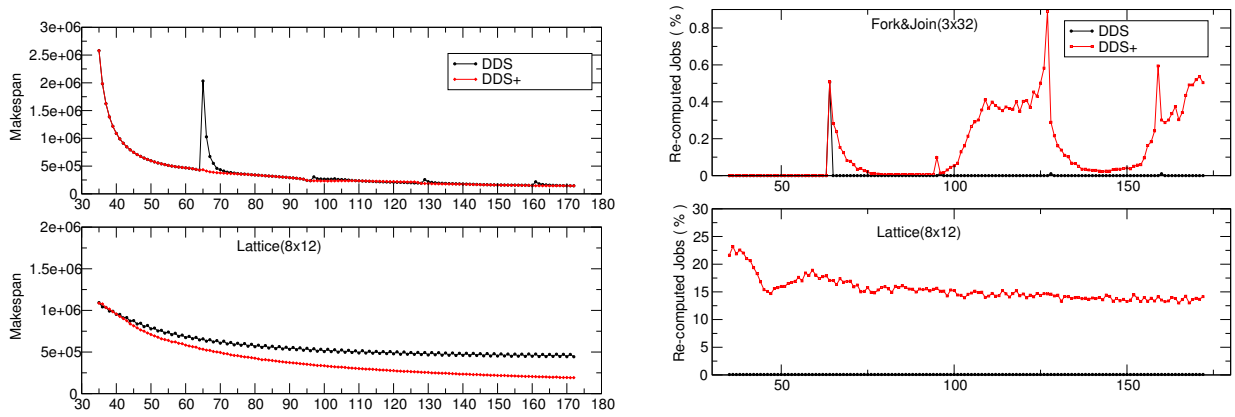
Another interesting observation is that when the storage budget is highly limited (e.g., $< 60$), the percentage of the re-computed jobs is almost 0, less than those with more storage budget, which contradicts our intuition. It demonstrates that our algorithm control the job executions well for the fork&join workload, and make the best use of the storage when the budget is limited. As the budget increases, more instances can execute concurrently, promoting the opportunities to rollback instances as we discussed above.

Compare to the fork&join workload, the performance of lattice workload also exhibits the same trend as the storage budget increases. Due to the degree of concurrency increases progressively in lattice workloads, the makespan curves are relatively stable. In contrast to the fork&join workload, the percentage of re-computed jobs is high in lattice workload, especially when the storage budget is highly limited, which is quite different from the situation in fork&join. We attribute this difference to the shape of the workflow and the dispatch/issue control of our algorithm. In our fork&join workload, a new instance (i.e., the first job of the instance) can be dispatched/issued if there are 32 storage units available. Therefore, when the budget is highly limited, this requirement is seldom satisfied, and at most one instance can be dispatched/issued. On the contrary, in the lattice workload, a new instance can be dispatched/issued if only 2 storage units are available. As the number of concurrent instances increases, the number of rollback instances also increases. Due to the low requirements for the storage to dispatch/issue an instance, the percentage of re-computed jobs in the lattice workload is much higher than that of fork&join workload. Therefore, the speculative execution of the lattice workload may be thought of being costly. Fortunately, more than 98% rollback instances have less than 5 jobs which are completed before rollback. Figure 6 evidences this observation by the cumulative distribution functions (CDFs) given the budget of 60, 80, 120 and 160 units.

*Effects of Scheduling Rollback Instances.*

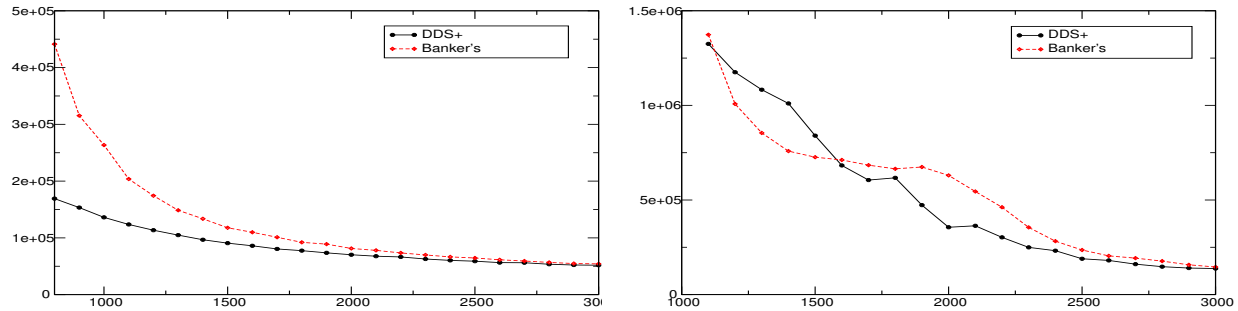
In this section, we evaluate the effects of scheduling rollback instances, which allows *DDS*$^+$ to be performance resilient. Figure 7 shows the performance comparison between the strategies with and without using rollback instance scheduling.

The fork&join workload has performance anomaly when the storage budgets are between 65 and 70 units if no rollback scheduling is considered (see Figure 7(a). We guess this anomaly is resulted from the comprehensive effects of the shape of the workflow and the dispatch/issue control of the scheduling policy. For example, when the budget is 65, only one job in one instance is executing (the second instance holds 32 storage units, but cannot execute), leading to low degree of concurrency. This is evident by the spike in Figure 7(b). By performing an extra rollback scheduling step, this performance anomaly is suppressed. Although such extra step may increase the number of re-computed jobs (Figure 7(b)), the performance is not degraded (Figure 7(a)) since on the one hand, the effect of those re-computed jobs are marginal as shown in the previous experiment, and on the other hand, the extra-step may lead




(a) Makespan comparison (b) Re-computed job comparison

**Figure 7: Performance comparison between *DDS* and *DDS*$^+$ for fork&join and lattice (AIT=0, JST=Uniform). The x-axes labels represent the storage budget units.**




(a) fork&join ($3 \times 32$) (b) lattice ($8 \times 12$)

**Figure 8: Performance comparison between *DDS*$^+$ and the Banker's algorithm. The x-axes labels represent the storage budget units**

to re-distribution of the released storage to the instances with most done jobs, and thus, accelerate the computation as a whole.

The performance improvement is also obtained from the lattice workload in spite of the large percentage of the re-computed jobs (Figure 7). The reason behind this phenomenon has been explained in the previous sub-section. It demonstrates that our instance rollback strategy is effective to make a good use of the storage budgets for performance improvement. In addition to the uniform distribution, the same performance gains are also observed in other investigated JST distributions.

### 5.2.2 *Comparison with Deadlock Avoidance*

In this section, we show how the performance of *DDS*$^+$ is relative to the Banker's algorithm, a classic algorithm for deadlock avoidance. In the experiment, we vary the file sizes from 1 to 10 uniformly, and use x-axis for all graphs to represent storage units, where the leftmost, starting point on the x-axis is based on the largest maximum claim of the Banker's algorithm for all workflow instances.

Figure 8(a) is the performance comparison for the fork&join workload where we can observe that *DDS*$^+$ is constantly better than the Banker's algorithm, especially when the storage budget is highly limited. This is mainly because the large values of the maximum claims computed by the Banker's algorithm prevent more instances from being admitted to execution. This situation is very serious in the case that the storage is highly constrained. However, when the storage resources increase, more and more jobs inside the active instances can run concurrently (i.e., intra-instance concurrency), dramatically improving the overall performance to approach *DDS*$^+$. Although the inter-instance concurrency (i.e., the number of concurrent instances) is not improved that much, fork&join has a high intra-instance concurrency due to its large number of independent jobs in the workflow. On the contrary, *DDS*$^+$ admits instances in an eager way without safety checking, which allows more jobs in different instances to execute at the same time, thus it has a better performance when the storage is highly constrained. However, this advantage is diminished as the storage budget increases.

Unlike the fork&join, the performance advantage of *DDS*$^+$ over the Banker's algorithm is not that pronounced to the lattice workload (Figure 8(b)). When the storage is tight, the Banker's algorithm exhibits some performance advantage over *DDS*$^+$. However, as the resources increase, *DDS*$^+$ gradually outperform the Banker's algorithm. We can attribute this observation to the job concurrency pattern of the lattice which is quite different from that of the fork&join.

This experiment demonstrates that *DDS*$^+$ does not consistently outperform the deadlock avoidance algorithm across all shapes of workflows. It exhibits performance advantages over the deadlock avoidance algorithm when workflows have a relatively high degree of intra-instance concurrency.

# 6. CONCLUSIONS

In this paper, we proposed *DDS*$^+$, a performance-resilience algorithm, which is based on our previous result by adding a rollback operation to *DDS* to overcome the performance anomaly. We investigated the behavior of the *DDS*$^+$ algorithm with respect to its scheduling of the workloads with more practical features when the storage is constrained as well as its ability to handle the performance anomaly. *DDS*$^+$ does not consistently outperform the deadlock avoidance algorithm (e.g., the Banker's algorithm) across all shapes of workflows. It exhibits performance advantages over the deadlock avoidance algorithm when workflows have a relatively high degree of intra-instance concurrency. These results deepen our insight for deploying the algorithm *DDS*$^+$ in reality to improve the performance of the scientific workflow-based workloads.

# 7. REFERENCES


[1] Panorama, 2014. `https://pegasus.isi.edu/projects/panorama`.

[2] Sextractor, 2014. `http://www.astromatic.net/software/sextractor`.

[3] B. Barish and R. Weiss. Ligo and the detection of gravitational waves. *Physics Today*, 52, 1999.

[4] J. Bent, T. E. Denehy, M. Livny, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Data-Driven Batch Scheduling. In *Data-Aware Distributed Computing 2009 (DADC09)*, Munich, Germany, jun 2009.

[5] J. Bent, D. Thain, A. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of Networked Systems Design and Implementation (NSDI)*, pages 365–378, San Francisco, California, USA, 2004.

[6] W. Chen and E. Deelman. Integration of workflow partitioning and resource provisioning. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, pages 764–768, 2012.

[7] E. Deelman and A. Chervenak. Data management challenges of data-intensive scientific workflows. In *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on*, pages 687–692, May 2008.

[8] S. Djorgovski, R. Gal, S. Odewahn, R. de Carvalho, R. Brunner, G. Longo, and R. Scaramella. The digital palomar sky survey (dposs). *Wide Field Surveys in Cosmology*, 1:10–20, 1998.

[9] P. Gburzynski. SMURPH, `http://www.olsonet.com/pg/PAPERS/side.pdf`, access date: Oct. 2, 2012.

[10] T. Glatard, J. Montagnat, and X. Pennec. Grid-enabled workflows for data intensive medical applications. In *18th IEEE Symposium on Computer-Based Medical Systems*, pages 537–542, Trinity College Dublin, Ireland, 2005.

[11] J. Gray, D. Liu, M. Nieto-Santisteban, A. S. Szalay, D. DeWitt, and G. Heber. Scientific data management in the coming decade. Technical Report MSR-TR-2005-10, Microsoft Corporation, 2005.

[12] D. Gunter, E. Deelman, T. Samak, C. Brooks, M. Goode, G. Juve, G. Mehta, P. Moraes, F. Silva, M. Swany, and K. Vahi. Online workflow management and performance analysis with stampede. In *Network and Service Management (CNSM), 2011 7th International Conference on*, pages 1–10, Oct 2011.

[13] K. Knight and D. Marcu. Machine translation in the year 2004. In *In Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 965–968, 2005.

[14] P. Maechling, H. Chalupsky, M. Dougherty, E. Deelman, Y. Gil, S. Gullapalli, V. Gupta, C. Kesselman, J. Kim, G. Mehta, B. Mendenhall, T. Russ, G. Singh, M. Spraragen, G. Staples, and K. Vahi. Simplifying construction of complex workflows for non-expert users of the southern california earthquake center community modeling environment. *SIGMOD Rec.*, 34(3):24–30, sep 2005.

[15] S. Pandey and R. Buyya. Scheduling of scientific workflows on data grids. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid*, pages 548–553, Washington, DC, USA, 2008.

[16] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Mayers, and M. Samidi. Scheduling data-intensive workflows onto storage-constrained distributed resources. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid*, pages 401–409, 2007.

[17] A. Rosenberg. On scheduling mesh-structured computations for internet-based computing. *IEEE Transactions on Computers*, 53(9):1176–1186, September 2004.

[18] T. Samak, D. Gunter, M. Goode, E. Deelman, G. Juve, G. Mehta, F. Silva, and K. Vahi. Online fault and anomaly detection for large-scale scientific workflows. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 373–381, Sept 2011.

[19] G. Singh, K. Vahi, A. Ramakrishnan, G. Mehta, E. Deelman, H. Zhao, R. Sakellariou, K. Blackburn, D. Brown, S. Fairhurst, D. Meyers, G. B. Berriman, J. Good, and D. S. Katz. Optimizing workflow data footprint. *Sci. Program.*, 15(4):249–268, Dec. 2007.

[20] Y. Wang, M. Hu, and K. Kent. ACS: an effective admission control scheme with deadlock resolutions for workflow scheduling in clouds. *Computing*, pages 1–24, 2014.

[21] Y. Wang and P. Lu. DDS: A deadlock detection-based scheduling algorithm for workflow computations in hpc systems with storage constraints. *Parallel Comput.*, 39(8):291–305, Aug. 2013.

[22] Y. Wang and P. Lu. Maximizing active storage resources with deadlock avoidance in workflow-based computations. *IEEE Transactions on Computers*, 62(11):2210–2223, 2013.

[23] Y. Wang, P. Lu, and K. Kent. WaFS: A workflow-aware file system for effective storage utilization in the cloud. *IEEE Transactions on Computers*, PP(99):1–1, 2014.