The Computer Journal Advance Access published October 17, 2014

doi:10.1093/comjnl/bxu109

Dataflow-Based Scheduling for Scientific Workflows in HPC with Storage Constraints

Yang Wang[1,*] and Wei Shi[2]

[1]*IBM Center for Advanced Studies (Atlantic), University of New Brunswick, Fredericton, Canada*
[2]*Faculty of Business and I.T., University of Ontario Institute of Technology, Oshawa, Canada*
[*]*Corresponding author: yang.wang@ca.ibm.com*

In high-performance computing (HPC), workflow-based workloads are usually data intensive for exploratory analysis of a scientific computation problem that may involve a large parameter space. To achieve the best performance, storage resource constraint is always a pragmatic concern in reality as the potential problem space scale, especially in big data science, as well as its required dataset are ever growing to outpace any increasing rate of storage capacity. Therefore, the workflow computation in a HPC environment with finite storage resources is still a practical topic that is worthwhile studying. To this end, we propose a novel scheduling framework that enhances the scheduling policies of Versioned Name Space and Overwrite-Safe Concurrency, introduced in our earlier work, with abilities to handle the deadlock problem in workflow computation with finite storage constraints. We achieve this goal by leveraging the data dependency information of the workflow to integrate a collection of deadlock resolution algorithms into the workflow scheduler. With such integration, after extensive simulation-based studies we conclude that the enhanced scheduling policies can solve the deadlock problem introduced by the storage constraints caused by big data overflow. More interestingly, we demonstrate that our enhanced scheduling policies perform better than the cases where only pure deadlock algorithms are applied when storage is highly constrained in terms of makespan performance.

Keywords: deadlock resolution; concurrency and computation; workflow scheduling; dataflow

Received 13 May 2014; revised 22 July 2014
Handling editor: Rada Chirkova

1. INTRODUCTION

A computational workflow is usually composed of a variety of standalone application components that are either control-dependent or data-dependent carrying out a well-defined scientific computing process [1–3]. 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 typically for data-intensive processing or parameter-based studies. For example, the CyberShake [3] workload consists of 80 sub-workflow instances, each having more than 24 000 individual jobs and 58 GB data. Another example is the Sextractor [2] workload, which consists of 2611 pipeline instances on the DPOSS [4] dataset, with each instance accessing a different 1.1 GB image to search for bright galaxies. Therefore, resolving the dependencies between instance and maximizing their concurrencies can significantly optimize such computation. However, it is not always available in practice due to a variety of resource constraints.

In this paper, we are particularly interested in the trade-offs between the storage overhead and *makespan* performance. With the advance of high-performance computing (HPC) in big data science, the size of involved datasets is constantly growing up to outpace the increasing rates of any affordable storage capacity [5]. In particular, some practical and system policies still exist in certain cases to limit the freedom of storage uses. For example, when multiple instances of a workflow share the same file system for their concurrent execution, an individual instance may run out of storage space. Consequently, the storage space management remains

important for maintaining high performance and throughput of the computational workflows in HPC [6].

In our earlier work, we developed two workflow scheduling policies: Versioned Name Space (*VNS*) and Overwrite-Safe Concurrency (*OSC*) [7] to show how the data dependencies between the constituent jobs can be exploited to maximize the concurrency at the cost of storage overhead [7]. VNS and OSC represent two extremes along a spectrum of policies that maximize the makespan performance while assuming the storage capacity is infinite. Although those proposed scheduling policies exhibit certain advantages in storage management for high-performance workflow computing, they are incapable of resolving the *deadlock* problem that could be caused by the competition for the finite storage resources from concurrent workflow instances.

Given finite storage resources, it is highly possible that members of a set of the workflow instances hold input storage, but are blocked indefinitely from accesses to the storage resources held by other instances within the set for their output data. Since, according to our computation model (described later), no member of the set releases its own resources before it completes its computing tasks, the deadlock will last forever unless the resolution is involved.

Deadlock is a prevalent problem in HPC when non-preemptive resources in general and storage in particular are constrained. In our case, deadlock not only degrades the system performance but also under-utilizes the storage resources if it is improperly addressed. These problems will become more serious in the era of cloud-based HPC where the resources are typically provisioned on-demand in a *pay-as-you-go* fashion. In this paper, we enhance the proposed workflow scheduling policies to address the storage constraints via deadlock resolution, which allows them to be used in more practical environments.

In practice, there are various ways to handle 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 advocate the deadlock resolution approach as it is more flexible and efficient in storage management. The contributions of this paper are as follows:

(i) We enhance the existing scheduling policies by integrating the proposed deadlock resolution algorithm to maximize the makespan performance and minimize the storage overhead for workflow computation with storage constraints.
(ii) We present in-depth simulation results to show how these enhanced scheduling policies behave with respect to different features of the workloads as well as different storage budget.

The simulation results show that the enhanced scheduling policies not only expand their application range as desired but also show performance advantages over the pure deadlock resolution approaches when the storage is highly constrained. Although there exist some related work in the literature on scheduling data-intensive workflows on storage-constrained resources [8–12], to our best knowledge, we believe we are the first ones to consider the deadlock issues in such a situation. Therefore, our results are deemed to be of great importance to adapting the scheduling algorithms to the workflows in reality.

The rest of the paper is organized as follows: in the next section, we discuss some related work in workflow scheduling with storage constraints. The computation model is presented in Section 3. We outline the scheduling polices to be enhanced as well as the candidate deadlock resolution algorithms in Section 4. We then present the enhanced policies in Section 5 and the evaluation results in Section 6. Finally, we conclude the paper in Section 7.

2. RELATED WORK

The applications of computational workflows to big data science inspired great interest in recent years in scheduling scientific workflows with data and storage managements in mind [6, 9, 11–13]. Pandey and Buyya [6] optimize the resource selections in the context of a data grid in the presence of replicated files for scientific workflow. They identify that scheduling policy and storage constraint are two major challenges in workflow scheduling and advocate to consider them together, which is also our concern. In contrast, Juve and Deelman [13] investigate how data can be managed in shared file systems for efficient workflow computation in the cloud. Given the pay-as-you-go billing model, it is particularly important for users to minimize the resources to support the execution of their workflows on provisioned resources.

The key issue centered around the constrained storage is how to design the scheduling policies to allocate jobs and then reclaim the unused storage resources so that the finite storage capacity can be fully utilized. Ramakrishnan *et al.* [9] pioneered the research to schedule data-intensive workflow onto a set of distributed computational sites with storage constraints. The basic approach is to introduce a *cleanup job* to each data file when that file is no longer used by other jobs in the workflow or when it has already been staged out to some permanent storage. The garbage files are collected in time, and the storage footprint of the workflow can thus be reduced significantly.

Although Ramakrishnan's approach is effective in storage reclamation, it cannot guarantee that a predefined storage limit is not exceeded in the course of the computation. This is a serious drawback as the execution could be aborted when the storage capacity is not sufficient for the workflow under the given job scheduling and garbage-collecting strategies. Bharathi and Chervenak [11] note this and adopt the same concept of the cleanup jobs to propose a *heuristic* and a *genetic* algorithm to address the *storage limit* problem where

the storage limit is respected by examining the cleanup jobs in breadth-first (in heuristic) or the chromosome's (in genetic algorithm) order and waiting for those pending cleanup jobs to complete if the storage limit is broken.

Similar to Ramakrishnan's approach, Bharathi and Chervenak's scheduling algorithm is also on a per-job basis. As noted, this strategy is not quite effective for those large workflows whose number of jobs is on the order of hundreds, thousands or even more CyberShake like workflow. Chen and Deelman [14] study this issue and develop a suit of heuristics to partition a large workflow into a set of sub-workflows so that the storage constraints in each computing site could be respected and in the meanwhile the internal job concurrency is maximized.

Although neither aforementioned studies mention the deadlock problem explicitly due to storage constraints,[1] their approaches by nature are a kind of *deadlock prevention* strategy, which is different from ours that uses storage space-efficient strategies to resolve the deadlocks. On the other hand, our goal is to schedule multiple workflow instances, each of which have an equal number of jobs on shared resources for concurrent execution instead of a single large workflow instance running in a distributed environment with storage constraints. Therefore, our scheduling policies are two-layered: instances first and then jobs.

DAR, *DTO* [15] and *DDS* [16] are designed for space-efficient workflow computation in HPC system with storage constraints. These algorithms take the advantages of the dataflow information of the workflow to either wisely avoid unsafe states in storage allocation or speculatively execute each instance whenever the instantaneous storage space is sufficient for some job executions (but not sufficient for the whole instance) or perform the *rollback* operations on the selected in-progress instances whenever deadlock or performance anomaly is detected. Although these algorithms can resolve deadlocks, they do not consider the trade-offs between the storage overhead and the performance; neither have they been incorporated with scheduling policies to evaluate their real impact on system performance.

3. COMPUTATION MODEL

The computation model adopted in this paper includes two parts: the workflow model that abstracts the key characteristics of real-world workflow-based computation in HPC, and the execution model which represents a typical process to execute the workflow computation.

3.1. Workflow Model

We model a workflow as 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, and the storage for input files could be garbage collected if they are no longer used by the subsequent jobs.

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 multiple instances are usually created for a parameter sweep study, and thus follow the same computation process.

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.

3.2. Execution model

The execution model is based on a physical or virtual cluster of homogeneous compute nodes that are connected via a high-speed network and share the same file system as the media for data communication between jobs. All the compute resources are managed by a batch workflow scheduler.

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 a *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 dataset can enter a *ready* state for execution. Otherwise, they will be in a *pending* state waiting for the storage. Whenever the required storage is available, the jobs in the pending state is changed to the ready state for execution again. The jobs in a *running* state are never stopped until they complete the computation. After a job has completed, it enters a *done* state. A completed job will release the storage space of its input data files if they are no longer used by the subsequent jobs. In contrast, the job will keep the storage for the output data files for future 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 predetermined and remain unchanged during the computation.

[1] Chen and Deelman point out that the partitioned workflow could form a deadlock loop if the cross dependencies between sub-workflows are not resolved. But this form of deadlock is not considered in this paper.

Moreover, in our model, the storage resources are provided in the form of a storage budget which represents the maximal storage capacities that could be used by the computation. Consequently, when the storage is not sufficient for all the competing free jobs (instances) in the computation, those free jobs would enter their pending states, holding the input storage while awaiting the output storage. As such, a deadlock is incurred.

4. SCHEDULING POLICY AND DEADLOCK RESOLUTION

In this section, we first overview the scheduling policies *VNS* and *OSC* [7], which leverage the dataflow information to trade-off storage overhead for performance in workflow scheduling. Then, we outline the basic ideas of the deadlock resolution algorithms that will be integrated with the scheduling policies for more practical use in reality.

4.1. Scheduling policies: VNS and OSC

VNS is to create a separate namespace as a sandbox for each instance to safely execute without interference with each other in terms of file name conflicts, a technique similar to *register renaming* in processor microarchitecture design [17, 18]. *OSC* adopts a different policy similar to the *software pipelining* technique in compiler optimization [19, 20] to allow a subsequent instance to concurrently execute with the instances ahead of it unless there are file name conflicts. Clearly, VNS and OSC represent two extremes along a spectrum of policies that trade off the storage capacity for the makespan performance. Compared with VNS, OSC consumes much less storage, but its performance is limited. Compared with OSC, VNS maximizes performance, but it consumes much more storage.

Both VNS and OSA are built on top of a *Version Namespace Manager* (VNM) which gathers and manages the data dependency information on a per-instance basis and provides its clients (e.g. workflow scheduler) with various data services [7].

Although the proposed scheduling policies exhibit the advantages in storage management for high-performance workflow computing, they only work in the situation where the storage capacity is infinite since both policies fall short of strategies to resolve the *deadlock* that could be caused by the competition for the finite storage resources from concurrent workflow instances.

4.2. Deadlock resolution: DAR and DTO and DDS

DAR and DTO [15] can be viewed as extensions to Lang's algorithm [21] for deadlock avoidance in the computation of workflow-based workloads with arbitrary shapes. The essence of both algorithms is to use different strategies to exploit the dataflow information of the workflow for the reduction of the maximum storage claim of each instance at run time.

In DAR, the maximum claim of each instance is dynamically computed by summing the resource requirements of all the remaining jobs (i.e. those jobs that have not yet been finished), instead of using a static predefined value. Clearly, this value is monotonically decreasing as the computation proceeds, and thus the storage resources could be more effectively utilized by those in-progress concurrent instances, which, as a consequence, speeds up the workload as a whole.

In contrast to DAR, DTO adopts a different strategy to compute the dynamic maximum claim of each instance in the workload. It topologically orders the remaining jobs in the current instance and in the meanwhile computes the minimal maximum claim for the topological sequence in the following ways:

(i) When a new job is ordered, we first ensure that all its input data are available (usually generated by its upstream jobs), and then allocate the storage space for its output data (our execution model).
(ii) When a running job is finished, the storage for its input data will be released immediately if the input data are no longer used by the subsequent jobs (using dataflow dependencies).
(iii) After all the jobs in the instance are ordered, the maximum storage once used in the sorting course is used as the minimal maximum claim of the instance.

Based on this computation, one can clearly see that the maximum claim is not a fixed number for the same set of remaining jobs in the instance since it is related to how the jobs are topologically sorted. However, with the availability of a topological sequence, we also know there is a safety sequence as well since the scheduler can follow the topological sequence to safely schedule the jobs within the claimed maximum storage capacity without incurring a deadlock. The scheduler computes the topological sequence for safety check each time a job is scheduled. This is the basic idea of DTO for deadlock avoidance. Compared with DAR, DTO is more aggressive in storage utilization since, for the same instance, its computed maximum claim is constantly less than that of DAR.

Unlike the previously selected algorithms, the basic idea of the DDS algorithm [16] is based on the deadlock detection and recovery principle. Specifically, given a storage budget Bgt, the DDS algorithm works as follows to detect a deadlock:

(i) Select an instance as well as a job for which the request is less than or equal to Bgt. If such a job is found, the job is granted its request and put into execution. Then, the algorithm continues to search until no more such jobs can be found. If no such jobs exist, go to (3).
(ii) Whenever a job is completed, its current allocated storage resources (for input data) are added to Bgt and then go back to (1) after releasing the ready jobs.

Algorithm 1 Integrated scheduling algorithm.

```
1:  procedure SCHEDULING(bgt)
2:      r ← 1                                              ▷ for computing the associated inst.
3:      sb ← bgt                                           ▷ init. storage budget
4:      while accept(message) do
5:                                                         ▷ when a new instance arrives
6:          if message = NEW_INST then
7:                                                         ▷ a new inst. id and workflow id from VNM
8:              I ← getNewInstance()
9:              do_newinstance(r, sb, I)
10:         end if
11:                                                        ▷ when a notice of a done job arrives
12:         if message = DONE_JOB then
13:             j ← getDoneJob()
14:             do_donejob(r, sb, j)
15:         end if
16:                                                        ▷ when a request for a job arrives
17:         if message = REQ_JOB then
18:             j ← FCFS(ReadyQ)                           ▷ send the job j to the requester
19:             req ← j
20:         end if
21:     end while
22: end procedure
```

(iii) If there are running jobs, the algorithm simply waits for a job completion and go to (2). Otherwise, a deadlock is detected.

After a deadlock is detected, DDS then uses a variety of strategies to compute the amount of storage that needs to be released, select the victims to rollback and finally reallocate the released storage to recover from the deadlock.

5. ENHANCED SCHEDULING POLICIES WITH DEADLOCK RESOLUTIONS

Given the basic idea of the scheduling policies and the deadlock resolution algorithms, in this section we focus squarely on their integration to address the storage constraints. The details of each policy and algorithm can be found in the corresponding references and are not reiterated here. Basically, the scheduling policies and the resolution algorithms are orthogonal in terms of the functionality and implementation mechanism, and thus the integration is relatively simple. The key point is how to manage the storage budget (including resolve deadlock) when new instances arrive and done jobs are noticed. With supports from *VNM*, we sketch the integration process in Algorithms 1–3.

The main scheduling algorithm (Algorithm 1) is abstracted into a 3-state machine to process three kinds of messages. First in Algorithm 2, when a new instance arrives (i.e. `NEW_INSTANCE`), a new instance id i and a new/old workflow id w will be obtained from *VNM*. These ids are used to identify the namespace for this instance in the following computation. After that, for each job j in instance i, we try to find its *dependent jobs* (actually, the jobs' names), i.e. those jobs in instance $i - r$ (of the same workflow) such that the job j cannot be scheduled until all those jobs have finished by instance $i - r$ (*inter-instance control-flow*). This is accomplished by `VNM.findOutJob()` (`Line7`). As for OSC, `VNM.findOutJob()` will return the direct successors (job names) of job j in the dataflow graph.

In contrast, VNS has the least constraints with an empty set of dep_jobs. After determining its dep_jobs, job j will be submitted to `SubQ`. We see that `SubQ` accommodates the submitted jobs on a per-instance basis. Each job in `SubQ` is associated with a set of data structures and functions to store and manage its scheduling information. For example, the data structures of holding the set of intra- and inter-instance dependent jobs, which are returned by the functions `intra()` and `inter()`, respectively. Another important function is `SubQ[i][j].associate()`, which is used to identify the inter-instance dependent jobs of the current job j in instance i, and put them into a set of inter-instance dependent jobs (`Line11`). For each job in `SubQ`, it will be put into `BlockedQ` if its data dependencies are not resolved. Otherwise, the integrated deadlock resolution algorithms is invoked via *Deadlock Resolver* (*DR*). Depending on whether the deadlock can be resolved or not, the job will be put into `ReadyQ` for execution after allocating the storage for its output data or `PendingQ` for requested storage resources (`Lines13−24`).

Whenever a notice of a done job arrives (i.e. `DONE_JOB`) in Algorithm 3, the available storage budget is first updated due to the released storage from the done job, and then the jobs in `PendingQ` are checked one by one using the deadlock resolution algorithm to see if they can be transferred from `PendingQ` to `ReadyQ` (`Lines 4−13`). Following that, all the jobs in the *current instance* are enumerated to see if any *intra-instance* dependency can be resolved.

Algorithm 2 Algorithm for processing new instance.

```
1:  procedure DO_NEWINSTANCE(r,sb,I)
2:      (i, w) ← VNM.getInstanceId(I)
3:      for ∀j ∈ I_i do
4:          dep_jobs ← ∅
5:          if I_{i−r} ≠ ∅ then
6:                                                  ▷ policy could be VNS or OSC
7:              dep_jobs ← VNM.findOutJob(w, j, policy)
8:          end if
9:
10:         SubQ.submit(i, j)                       ▷ SubQ[i] holds I_i
11:         Sub[i][j].associate(dep_jobs, I_{i−r})
12:                                                 ▷ check intra- and inter-instance dependency
13:         if SubQ[i][j].intra() = ∅
14:                 ⋂ SubQ[i][j].inter() = ∅ then
15:                                                 ▷ DR: Deadlock Resolver, alg could be DTO, DAR or DDS
16:             if DR.resolveDeadlock(i, j, sb, alg) then
17:                 sb ← sb − W_ij                  ▷ W_ij is the write size of j ∈ I_i
18:                 ReadyQ.submit(i, j)
19:             else
20:                 PendingQ.submit(i, j)
21:             end if
22:         else
23:             BlockedQ.submit(i, j)
24:         end if
25:     end for
26:
27: end procedure
```

If so, the corresponding job is removed from `BlockedQ` and linked to the `ReadyQ` as long as there is no inter-workflow instance dependencies and success in deadlock resolution (`Lines 15−33`). After resolving intra-instance dependency (of the current instance i), the *inter-workflow instance* dependencies are resolved between the *current* and the next instances by roughly following the same principle as in the intra-instance dependency resolution (`Lines 35−51`). Note that `SubQ[i+r][k].clear(j)` in `Line38` is used to remove job j from job k's dependent job set in `SubQ[i+r]` (computed by `Line 7` and `Line 11` in Algorithm 2). For those jobs in both instances i and $i + r$, when their intra- and inter-instance dependencies are resolved, the algorithm follows the same procedure as that in `NEW_INSTANCE` to resolve the deadlock problem.

Finally, when a new request for job arrives (i.e. `REQ_JOB`), *FCFS* is applied to `ReadyQ` to select the next ready job for executing (`Lines 17−20` in Algorithm 1).

6. PERFORMANCE EVALUATION

6.1. Experimental setup

We simulate the computation model presented in Section 3 and implement a scheduler using the simulation package SMURPH [22]. 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 according to the VNS or OSC policy and dispatches the job based on the deadlock resolver for safety check.

As in [7], we continue to use the two representative structures Fork&Join and Lattice as shown in [7] as the benchmarks in our experiments. The Fork&Join structure is characterized by the number of stages and fan-out factors, while the Lattice structure is characterized by its width and height. Both structures represent a wide range of scientific workflows that could be deployed in HPC systems [23–26].

Since there is no well-accepted model for job service time (JST), in experiments we assume that, for instances of all the examined workflows, the job service time is uniformly distributed in `[500,1000]` time units (i.e. indivisible time unit (ITU)) and their inter-arrival time follows the exponential distribution from `0` to `1600` time units.

To ensure that the observation is made on sufficient workloads, the number of instances in each workload is fixed as a constant `100` in each experiment. Since our studied scheduling policies are not intended to optimize the utilization of a limited number of compute nodes instead, they aim to maximize the degree of concurrency (DOC), and thus, we hope that the maximum DOC is never constrained by the compute nodes. Therefore, to reflect our concern, the computation environment is modeled as an unbounded number of homogeneous compute nodes that can access a shared storage.

Finally, for the purpose of comparison, we use *BASE* in some experiments as a baseline policy that entails the serial execution of the workflow instances in the workload. As such, in BASE, files are never versioned and storage is deallocated

Algorithm 3 Algorithm for processing done job.

```
1:  procedure DO_DONEJOB(r,sb,j)
2:      i ← j.getInstanceId()
3:
                                                  ▷ release the storage that is no longer used
4:      sb ← sb + R_ij                            ▷ R_ij is the read size of j ∈ I_i
5:                                                ▷ transfer jobs from PendingQ to ReadyQ
6:      for ∀k ∈ PendingQ do
7:          l ← k.getInstanceId()
8:          if DR.resolveDeadlock(l, k, sb, alg) then
9:              PendingQ.delete(k)
10:             sb ← sb − W_lk
11:             ReadyQ.submit(k)
12:         end if
13:     end for
14:                                               ▷ resolve intra-instance data dependency
15:     for ∀k ∈ SubQ[i] do
16:         if j ∈ SubQ[k].intra() then
17:             SubQ[k].intra() ← SubQ[k].intra()/{j}
18:         end if
19:
20:         if (k ∈ BlockedQ) ⋂ (SubQ[k].intra() = ∅) then
21:                                               ▷ check (inter-instance) name conflicts
22:             if SubQ[k].inter() = ∅ then
23:                                               ▷ alg could be DTO, DAR or DDS
24:                 if DR.resolveDeadlock(i, j, sb, alg) then
25:                     sb ← sb − W_ij
26:                     ReadyQ.submit(i, j)
27:                 else
28:                     PendingQ.submit(i, j)
29:                 end if
30:                 BlockedQ.delete(i, k)
31:             end if
32:         end if
33:     end for
34:                                               ▷ resolve inter-instance data dependency
35:     if SubQ[i + r] ≠ ∅ then
36:         for ∀k ∈ SubQ[i + r] do
37:             if ¬SubQ[i + r][k].inter() ≠ ∅ then
38:                 SubQ[i + r][k].clear(j)
39:                 if (SubQ[i + r][k].intra() = ∅)
40:                         ⋂ (SubQ[i + r][k].inter() = ∅) then
41:                                               ▷ alg could be DTO, DAR or DDS
42:                     if DR.resolveDeadlock(i, j, sb, alg) then
43:                         sb ← sb − W_ij
44:                         ReadyQ.submit(i, j)
45:                     else
46:                         PendingQ.submit(i, j)
47:                     end if
48:                 end if
49:             end if
50:         end for
51:     end if
52:
53: end procedure
```

after the completion of each instance. Although this straw-man policy is quite simple and inefficient, it is not uncommon in practice.

6.2. Simulation results

In the following experiments, we consider the situation when the storage is constrained. Again, the constrained storage is given as a storage budget that represents the maximal storage that a computation can use. To resolve the incurred deadlock, the scheduling policies can employ our studied deadlock resolution algorithms (i.e. DDS, DAR and DTO). Our purpose is primarily to measure the performance of these policies (combined with the deadlock resolution algorithms if needed) under a variety of *storage budgets*. To this end, we first classify the storage budgets according to the

TABLE 1. Minimal and maximal storage requirements of each scheduling policy.

	BASE		OSC		VNS	
	Min	Max	Min	Max	Min	Max
Fork&Join ($X \times Y$)	$Y+1$	$2 \cdot Y$	$Y+1$	$Y \cdot (X+1)$	$Y+1$	$2 \cdot k \cdot Y$
Lattice ($X \times Y$)	N/A	$4 \cdot \min(X, Y)$	N/A	$2 \cdot X \cdot Y - (X+Y)$	N/A	$4 \cdot k \cdot \min(X, Y)$

Fork&Join: X = Stage, Y = Fan-out Factor. Lattice: X = Width, Y = Height, k = the total number of instances.

requirements of the scheduling policies into a set of storage ranges, and then identify and evaluate the policies that can be applied to each storage range. Additionally, based on the classification, we also measure the storage utilization of each applied policy and investigate the impact of AIT on the performance.

6.2.1. *Storage budgets classification*

To classify the storage budgets, we analyze the maximal and minimal storage requirements for each policy and then use the analytical results to identify a set of storage ranges (see Table 1). More specifically, given a workflow of Fork&Join with stages of X and fan-out factor of Y (i.e. $X \times Y$), for all policies, we at least need Y storage units to enable the first job to execute, and an additional one unit to enable the subsequent jobs to execute. Thus, the minimal storage requirements of all scheduling policies are $Y+1$. Since, in Base policy, there are at most Y concurrent jobs in a single instance, with each job holding two storage units by its read and write dataset, the maximal storage requirements are thus $2 \cdot Y$. For VNS, because of k concurrent instances, this number is naturally $2 \cdot k \cdot Y$. But for OSC, the situation is a little bit complicated; due to the introduced inter-instance dependencies, there are at most $Y \cdot (X+1)$ (i.e. the total number of edges in the dataflow DAG) storage units to hold active files.

For the Lattice workflow, the formulae of the minimal storage requirements for all policies are not available, but the values are independent of policies, similar to the situation in Fork&Join. As for the maximal requirements, we can follow the same analysis in Fork&Join to give their formulae.

In the following experiments, for our Fork&Join workload, the minimal requirement of BASE and OSC policies is `33` since the fan-out factor is `32`, but the maximal is `64` and `128`, respectively, because there are at most `32` concurrent jobs based on BASE and `64` on OSC at any time point during the computation. For VNS, the corresponding values are `33` and `6400` since we have `100` instances in the workload. As for the Lattice workload, we can have a similar analysis. All the values are shown in Table 2. It should be noted that since we do not have formulae to express the minimal requirements of these policies for Lattice workload.

TABLE 2. Minimal and maximal storage requirements of each scheduling policy (based on dataflow DAG).

	BASE		OSC		VNS	
	Min	Max	Min	Max	Min	Max
Fork&Join (3×32)	33	64	33	128	33	6400
Lattice (8×12)	20	32	20	172	20	3200

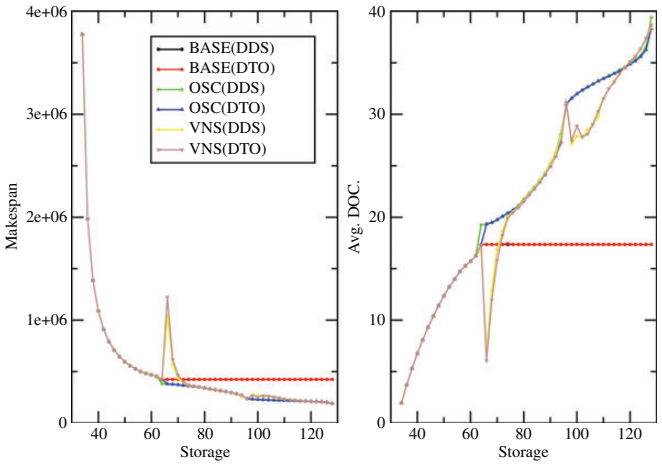

FIGURE 1. Performance comparison of the studied scheduling policies on Fork&Join (3×32) workload when storage budget is in the range of 33–128 units.

6.2.2. *Makespan and average DOC*

Figure 1 shows the performance comparison of the studied scheduling policies on Fork&Join workload when the storage budget is in the range of `33`–`128`. Owing to the strategies for deadlock avoidance in DAR, it is not available for all policies in this particular case (such storage is insufficient to enable it to work).

From this figure, we can see that all the policies combined with the given deadlock resolution algorithms exhibit the same performance in the storage range of `[33, 64]`. It is not difficult to understand that in such a case, the storage is only sufficient for at most one active instance. Thus, all

these policies have to serialize the execution of the workflow instances, leading to the same performance.

When the storage budget is >64, the performances of both BASE policies are kept as a constant because the maximal requirements of the BASE policies are 64. In [65, 72], VNS policies have big performance anomalies. A simple explanation to this phenomenon is the inappropriate admission of a new instance for execution, leading to low storage utilization. For example, when the storage budget is 65, both VNS policies can admit two instances for concurrent execution, each holding 32 storage units, leaving only one free storage unit. After the first job is completed in an instance, 32 jobs will become ready. Unfortunately, only one of them can be executed due to the remaining one free storage unit, reducing the average DOC severely (see Fig. 1 (right)). The severity is mitigated as the storage budget is gradually increased. We can observe the same phenomenon around 100 storage units when three instances can be admitted and the 4 storage units are left to increase the average DOC slightly. Our OSC policies address this problem by introducing control dependencies between consecutive instances.

After the storage budget is >72 and <128, both OSC and VNS outperform BASE policies since BASE policies cannot benefit from the storage increments. As for comparison between OSC and VNS in this particular case, their performance difference is almost indistinguishable.

Based on the above experimental results and analysis, we can draw the following conclusions:

(i) When the storage budget is extremely tight, say, not greater than the maximal requirement of BASE policy, for Fork&Join workload, all studied policies, no matter what kinds of deadlock resolution approaches are used, are reduced to BASE policy.
(ii) When the storage budget is greater than the maximal requirement of BASE policy but less than that of OSC policy, OSC, combined with either the DDS approach (i.e. OSC(DDS)) or the DTO approach (i.e. OSC(DTO)), is the performance leader. In this situation, VNS policy is also a competitor but it suffers from the performance anomaly.

Figure 2 shows the same simulation study, but for the Lattice workload. From this figure, we can observe the same phenomenon related to the BASE policies as that in the Fork&Join workload, that is, the performances of BASE policies are not changed after the storage budget is >32 (i.e. the maximal requirements of BASE policies). However, unlike in Fork&Join workload, the performances of the scheduling policies are different when the storage budget is in the range of [20, 32], no single policy can beat all the others on all the data points in this range.

When the storage budget increases from 32 to 172, the performances of both OSC and VNS are also improved accordingly. OSC leads the improvements because its introduced

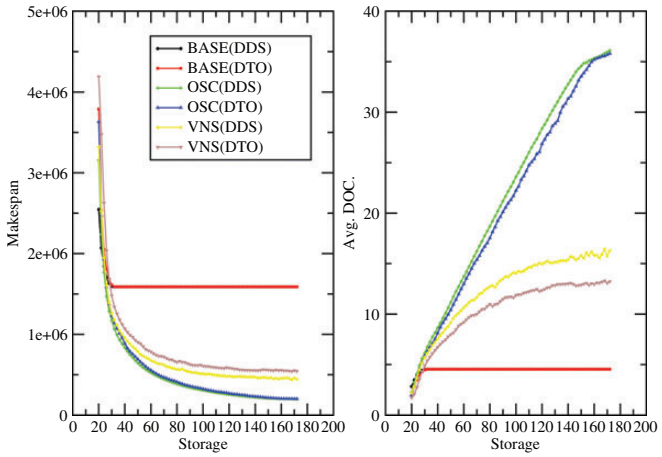

FIGURE 2. Performance comparison of the studied scheduling policies on Lattice (8 × 12) workload when storage budget is in the range of 20–172.

inter-instance dependencies allow the running instances to have more opportunities to acquire the needed storage to finish the computation, consequently reducing the number of blocked instances[2] and improving the storage utilization. As such, OSC has higher average DOC than VNS policies (see Fig. 2 (right)), running somewhat counter to intuition.

In addition, we also notice that both OSC and VNS outperform considerably BASE policies, which is different from the situation in Fork&Join workload (i.e. the performance gaps between BASE and OSC/VNS in Fork&Join is not so large as in Lattice). The reason is that the Lattice is expected to have a lower intra-workflow instance DOC than the Fork&Join because of the additional dependencies between the jobs. Thus, compared with the Fork&Join, given a storage budget, more Lattice instances can be executed concurrently by OSC and VNS. This is evidenced by Fig. 2 (right) where OSC and VNS enjoy much higher average DOC than BASE policies.

Furthermore, for VNS, we found that the performance of VNS(DDS) is better than that of VNS(DTO). We attribute this performance difference to two factors. First, there are always hosts/processors (be aware of the assumption that an unbounded number of homogeneous computational nodes are available) to minimize the overhead incurred by deadlock detection in VNS(DDS). Secondly, the safety checking algorithm in our deadlock avoidance is not optimal and always conservative, which may hurt the performance of VNS(DTO).

Similar to the Fork&Join, we draw the follow conclusions:

(i) In our Lattice workloads, no single policy can outperform all the others when storage budget is less than the maximal requirement of BASE policy.

[2]Blocked instances may hold the storage that cannot be used by other instances.

(ii) As the budget increases to the maximal requirement of OSC, both OSC policies (i.e. OSC(DDS) and OSC(DTO)) are clearly the performance leaders.

Other interesting storage ranges are `[128, 6400]` and `[172, 3200]` for Fork&Join and Lattice, respectively. These ranges represent the budgets that are greater than the maximal requirement of OSC and less than the maximal requirement of VNS with respect to each kind of workloads. Generally, BASE and OSC policies cannot gain more performance when the storage budget is greater than their maximal requirements, and thus, it is impossible for them to be the performance winner. Therefore, in the following experiments, we focus on the VNS policy combined with different deadlock resolution algorithms. But, for the purpose of comparison, we also show the data points of OSC in the figures. Actually, the performance of each VNS policy (i.e. VNS(DDS), VNS(DTO) and VNS(DAR)) reflects the effectiveness of the combined deadlock resolution algorithms.

Figure 3 (left) shows the makespan comparison between three VNS policies. Clearly, VNS(DAR) is the best among the three, which also has the highest AC (Fig. 3 (right)). The performance of VNS(DDS) and VNS(DTO) is indistinguishable. The performance advantage of VNS(DAR) over the other two lies in the fact that it overcomes the performance anomalies shown in the VNS(DDS) and VNS(DTO) policies by reserving storage (i.e. safety checking by using a storage upper bound) for the running instances. As discussed, performance anomalies are resulted from inappropriate storage allocation for some free or pending instances that may not make progress except for holding the storage. As for VNS(DDS) and VNS(DTO), they have the similar freedom to allocate storage to the scheduled job (VNS(DTO) relaxes its safety checking compared with VNS(DAR)), and thus, show similar performances. From this experiment, our conclusions are as follows:

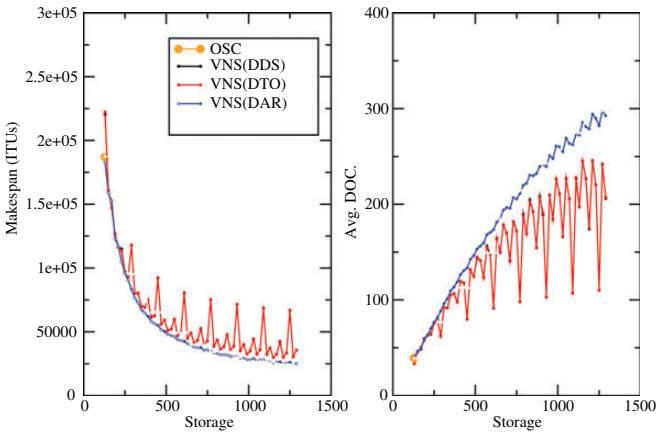

FIGURE 3. Performance comparison of the proposed scheduling policies on the Fork&Join (3 × 32) workload when storage budget is in the range of 130–1300.

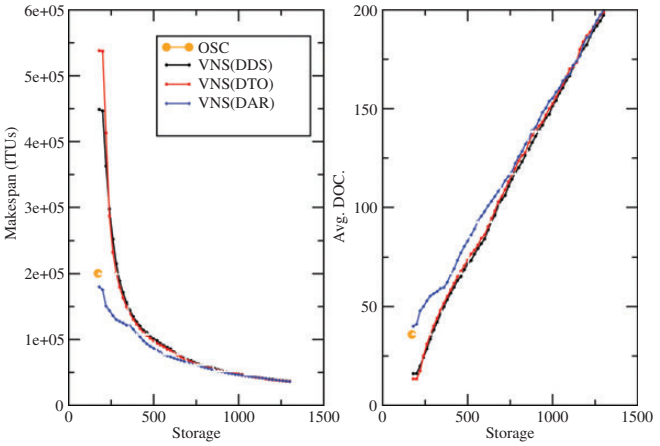

FIGURE 4. Performance comparison of the proposed scheduling policies on the Lattice (8 × 12) workload when storage budget is in the range of 180–1300.

(i) For single instance, VNS(DAR) is the most conservative in terms of storage utilization among the three VNS policies. However, for multiple concurrent instances, such conservativeness does not compromise the overall performance, rather it improves the performance (the best performance among all). This situation is opposite to that of VNS(DDS) and VNS(DTO).

(ii) Big fan-out factor and low average degrees of internal nodes of Fork&Join workload can lead to a performance anomaly even if the storage is not so scarce (i.e. greater than the maximal requirement of OSC).

Figure 4 shows the same experiment for the Lattice workload. Unlike the situation in Fork&Join, VNS(DAR), from the figure, has only a slight performance advantage over VNS(DDS) and VNS(DTO) when the storage budget is not `>700`. After that, all VNS policies enjoy the same performance (i.e. makespan and AC). This observation, again, demonstrates that when the storage is not sufficient for the workflow-based computation, making the running instances to have more opportunities to acquire the needed storage is beneficial to overall performance.

Comparing the two sub-graphs of Fig. 4, we found that the ACs of all the VNS policies increase almost linearly, but the corresponding makespan reductions are not proportional to the AC's increments. The reason is that when the storage is constrained, the makespan of the workload is primarily determined by the available storage, and very sensitive to its increment since there are always a large number of free/pending jobs waiting for storage to be executed. With the gradual increments of the storage, these free/pending jobs can acquire the needed storage for execution, and the ACs are naturally increased in a linear fashion. As such, the intra-instance data dependencies are replacing the storage, and

becoming the dominant factor to determine the makespan. However, the makespan cannot be reduced linearly since it is not determined by the total number of jobs, but the critical path of the workload.

To summarize, we have the following conclusions:

(i) Again, VNS(DAR) exhibits the best performance. The conservativeness of VNS(DAR) for single instance (in terms of storage utilization) improves the performance of multiple Lattice workflow instances. But the improvement is not so pronounced as that of the Fork&Join workload.
(ii) For the Lattice workload, all VNS policies can equally take advantage of storage increments to improve the average DOC and reduce makespan.

When the storage budget continually increases over the maximal requirements of VNS policy (i.e. the storage budget is in [6400, 12 800] for Fork&Join workload or [3200, 17 200] for Lattice workload), the performances of all VNS policies are the same as that of VNS under infinite storage budget, and hence, the best among all studied scheduling policies.

6.2.3. *Storage utilization*

In the following experiments, we investigate the storage utilization of the studied policies. To this end, we classify the storage into three classes. The first class is the storage that is being used by active instances. We call the storage in this class *active storage*; further, we denote the amount of active storage at time t as $S_{active}(t)$. The ratio of the active storage to the total available storage is thus defined as

$$R_{active} = \frac{\int_0^{makespan} S_{active}(t)dt}{makespan \cdot total\ storage}$$

The higher R_{active}, the better is the storage utilization, and thus, better is the performance. As we know, an active instance may become inactive due to the storage constraints. The total storage held by inactive instances belongs to the second class, named *inactive storage*. The total inactive storage at time t is denoted as $S_{inactive}(t)$. Accordingly, we have

$$R_{inactive} = \frac{\int_0^{makespan} S_{inactive}(t)dt}{makespan \cdot total\ storage}$$

One can easily see that inactive storage has a negative impact on the performance since it cannot be reclaimed for use by other active instances. Therefore, for $R_{inactive}$, the lower, the better. The third class of storage is called *free storage*, which neither belongs to the active storage nor to the inactive storage. Actually, it is free to be used in any instance. The free storage is related to scheduling policies, deadlock resolution algorithms, as well as the shapes of workflows. Similarly, at time t, the total free storage is denoted as $S_{free}(t)$. To describe the free storage, we have

$$R_{free} = 1 - (R_{active} + R_{inactive})$$

Figure 5 shows the ratios of these classes of storage for both workloads when the available storage is not greater than the maximal requirement of OSC. In the Fork&Join workload, all policies have the same storage ratios (including R_{active}, $R_{inactive}$ and R_{free}) before the storage budget reaches 64. This observation confirms that when the storage is not greater than the maximal requirements of BASE, all policies have the same performance. The R_{active}s of all policies are decreased in the storage interval of [34, 64]. This is because the single active instance cannot make full use of all available storage. We can verify this by seeing that their $R_{inactive}$s are 0 and the R_{free}s are monotonically increasing. The observation is continually obtainable for BASE policies until the storage reaches 128.

After 64, both OSC and VNS policies have similar storage ratios except that VNS policies have sharp drops in their active storage ratio R_{active}s in the range of [65, 70]. This anomaly is due to the inactive instances that hold storage (see the inactive storage sub-graph in Fig. 5 (top)). Both OSC policies overcome the anomaly by introducing control dependencies between consecutive instances. However, these introduced dependencies slightly hurt their R_{active}s after the storage is >100.

In the Lattice workload, the R_{active}s of the BASE policies are also the same, but in the Fork&Join workload, the R_{active} is dropped from 0.8 instead of 1. This is because the available storage cannot be fully used immediately by the single active instance. There are always some free storage at the beginning stage of executing each instance (see the corresponding R_{free}).

As we expected, due to the introduced inter-instance dependencies, OSC policies have the highest R_{active}, and lower $R_{inactive}$, and thus, they have the best performance among the policies. Compared with OSC(DDS), OSC(DTO) has a little bit worse R_{active} because of the conservativeness in its safety checking. This is further evidenced by the free storage sub-graph (Fig. 5 (bottom), where OSC(DTO) has more free storage than OSC(DDS). Such conservativeness in safety checking is also reflected in VNS(DTO), which has a lower R_{active} and higher R_{free} than VNS(DDS).

Compared with OSC, due to the lack of the inter-instance dependencies, VNS policies can admit more instances to execute, but these active instances may quickly become inactive under the given storage, and thus, have much more inactive storage (i.e. higher $R_{inactive}$). This fact demonstrates again the increasing the degree of inter-instance concurrency cannot always benefit the performance, but rather compromises the performance due to storage constraints.

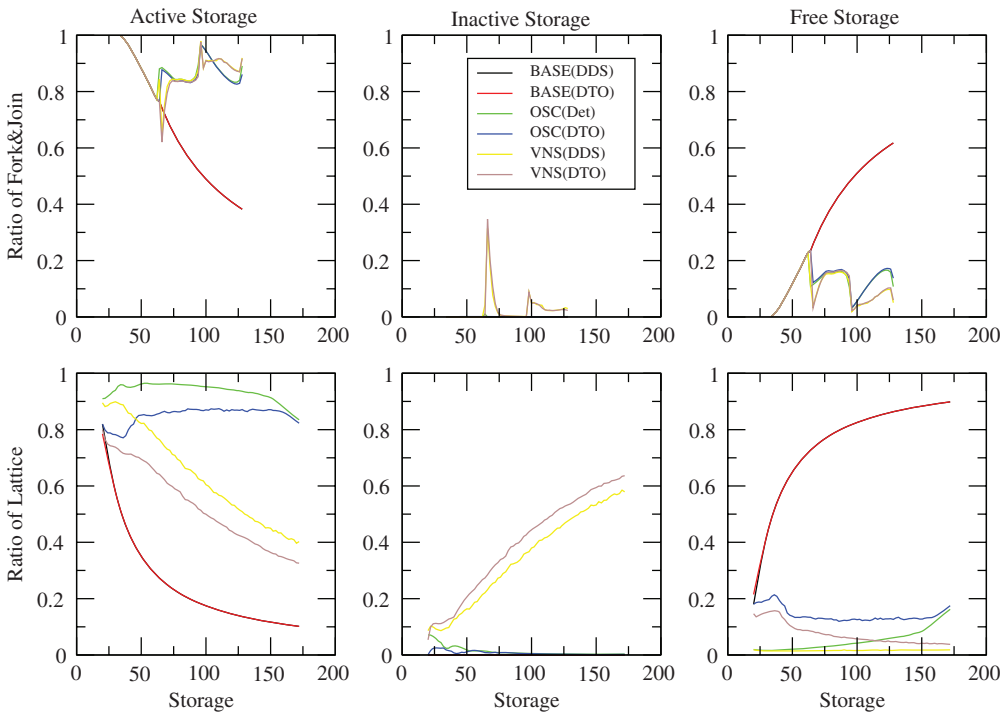

FIGURE 5. Storage utilization of the studied policies on Fork&Join (top) and Lattice (bottom) when the storage budget is half-maximal of OSC requirements.

To summarize, when the storage budget is not greater than the maximal requirement of OSC, we can draw the following conclusions on the storage utilization:

(i) For both workloads, OSC(DDS) exhibits the best performance in terms of storage utilization.
(ii) For our Fork&Join workload, all applied policies except the VNS policies on average have equal capabilities of minimizing the number of inactive instances. Thus, their storage utilization is largely determined by the free storage, more precisely, by the nature of the scheduling policies.
(iii) For our Lattice workload, under the given storage, increasing the degree of inter-instance concurrency will compromise the performance. The storage utilization of OSC is constantly better than that of VNS.

Figure 6 shows the same storage ratios as Fig. 5, but the storage budgets are varied from the maximal requirement of OSC to `1300`, representing moderate storage budgets.

In Fork&Join, VNS(DAR) has the best R_{active}, and both VNS(DDS) and VNS(DTO) exhibit the utilization anomalies, which are consistent with our previous experimental results pertaining to their performance anomalies. By comparing the three sub-graphs of Fig. 6 (top), we found that the overall trends of the R_{active}s of all applied policies are decreasing, but the trends of the corresponding $R_{inactive}$ are increasing with the increments of the storage. Furthermore, VNS(DAR), which is more conservative in the safety checking than VNS(DTO), has better utilization than VNS(DDS) and VNS(DTO). These results further offer the supportive evidence that the improvement of concurrency under the constrained storage does not always lead to the improvement of storage utilization.

Unlike the Fork&Join workload, in the Lattice workload, the R_{active} of VNS(DAR) decreases as the storage increases, which is opposite to both VNS(DDS) and VNS(DTO). The reason lies in the conservativeness of VNS(DAR) in safety checking and the aggressiveness of VNS(DDS) and VNS(DTO) in admitting new instances for execution.[3] Specifically, VNS(DAR) will reserve more storage for active instances than VNS(DDS) and VNS(DTO), and thus, has higher R_{active}.

From Fig. 6 (bottom), we observe that, for VNS(DAR), the R_{active} is decreasing with increments in storage, and the decrement has resulted from the increment in the corresponding R_{free}, rather than the $R_{inactive}$, which is also decreased (after certain point). This phenomenon is different from that in Fork&Join, where the $R_{inactive}$ of VNS(DAR) increases, but the R_{free}s remain largely unchanged. It demonstrates that, for the Lattice workload, VNS(DAR) can take advantage of the storage increment to improve

[3]For VNS(DTO), the aggressive admission of instances for execution is also related to its safety checking.

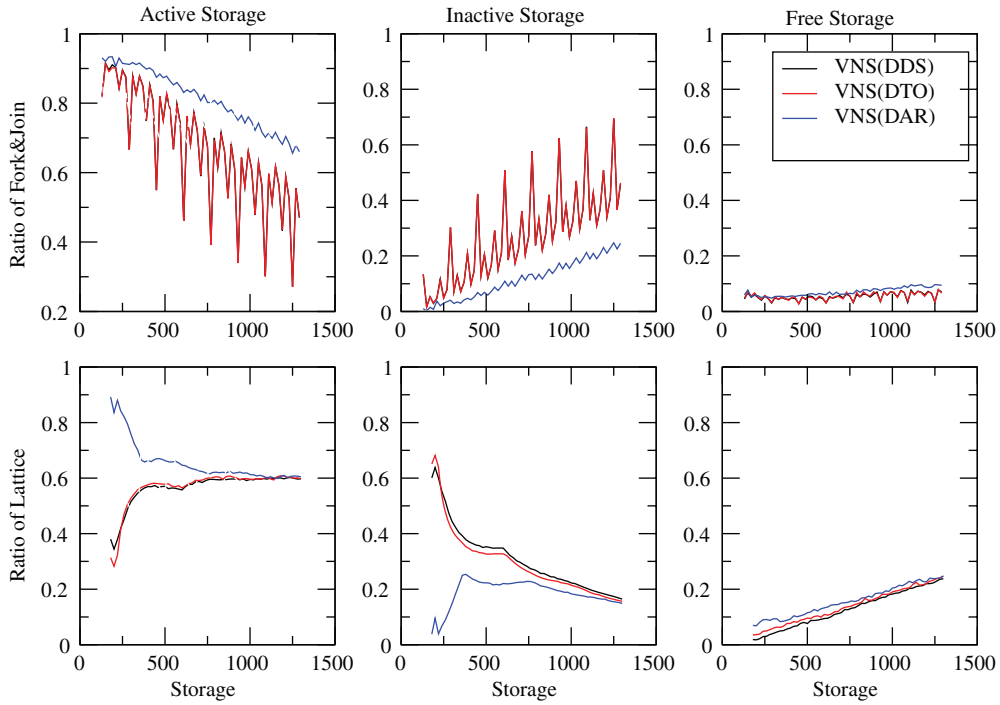

FIGURE 6. Storage utilization of the studied policies on Fork&Join (top) and Lattice (bottom) when the storage budget is equal to the maximal of OSC requirements.

performance, this improvement is limited by intra-instance data/control dependencies instead of the inactive instances as in the Fork&Join workload.

For both VNS(DDS) and VNS(DTO), their R_{active}s are increasing initially and remain largely unchanged ever after the storage budget is >700, and also approaching the R_{active} of VNS(DAR). This observation is also consistent with the experimental results in our previous studies.

When the storage is relatively constrained, due to the aggressiveness of VNS(DDS) and VNS(DTO) to admit instances for execution, they have low R_{active}s and high $R_{inactive}$s. This situation is mitigated as the storage increases, implying that the inactive instances will acquire the needed storage to make progress. We also observe that as the $R_{inactive}$s gradually decreases, the R_{free}s monotonically increase. This is a good phenomenon, demonstrating that the factors that determine the performance of the computation are gradually changed from those of limited storage to intra-instance concurrency.

To summarize this experiment, we have the following conclusions:

(i) For both workloads, in terms of storage utilization, VNS(DAR) exhibits the best performance.
(ii) As the storage increases over the maximal requirements of OSC (i.e. `128`) for the Fork&Join workload, the degradation of the storage utilization of the VNS policies mostly result from the increments of the inactive storage. This fact demonstrates that increasing the degree of inter-instance concurrency of the Fork&Join workload will compromise the storage utilization.
(iii) As the storage increases over the maximal requirements of OSC (i.e. 172) for the Lattice workload, the factors that determine the performance of the computation are gradually changed from those of limited storage to intra-instance concurrency. Thus, the degradation of the storage utilization of the VNS policies mostly result from the data/control dependencies inside the workflow instances.

6.2.4. *Impact of average inter-arrival time*

In the following set of experiments, we examine the relative performance of the studied scheduling policies as we vary the average instance inter-arrival time. To this end, we select some pairs of values `(64, 86)`, `(128, 172)` and `(256, 344)` as the storage budgets for the Fork&Join and Lattice workloads, respectively. These values represent half-maximal, maximal and double-maximal storage that the corresponding OSC requires.

Figure 7 shows the performance comparison of the studied policies when only half-maximal storage of OSC is available. A key observation on the Fork&Join workload is that the performance of each policy is independent of the average instance inter-arrival time. The reason is that the limited

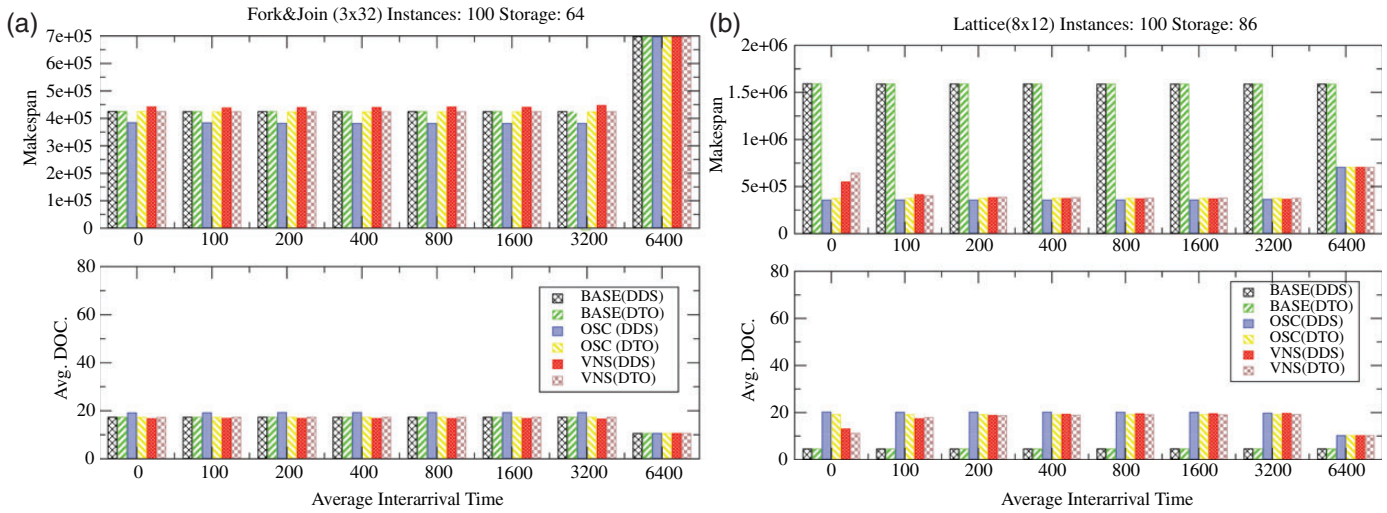

FIGURE 7. Simulation results of Fork&Join and Lattice under half of the maximal storage requirement of OSC (**a**) Fork&Join. (**b**) Lattice.

storage makes each policy to schedule the instances in a serial-like mode, and thus, when the workload is intense, the arrival instances will pile up at the server.

For the Lattice workload, as discussed, we attribute the performance gain of OSC to its introduced inter-instance control dependencies, whereas for VNS (AIT = 0), due to the storage contentions from multiple concurrent instances (jobs), either the conservativeness of deadlock avoidance (in VNS(DTO)) or the low storage utilization (occupied storage in blocked instances) (in VNS(DTO) and VNS(DDS)) compromises their performances. However, the performance loss diminishes as the average instance inter-arrival time increases since the probabilities for the running instances in VNS to acquire the needed storage are also increased.

By comparing both workloads, we can draw the following conclusions:

(i) For Fork&Join, when the storage is equal to half of the maximal requirement of OSC, the performances of all compared policies are independent of the average instance inter-arrival time.
(ii) For the Lattice workload, the performances of VNS(DDS) and VNS(DTO) can be gradually improved with the increments of AIT.

We next consider the performance comparison between the studied policies when the available storage is equal to the maximal requirements of OSC policy (i.e. (128, 172)). The purpose of this experiment is to see whether OSC policy is the best under the given storage as AIT increases. For Fork&Join workload, our conclusion is that OSC and all three versions of VNS (i.e. VNS(DDS), VNS(DTO) and VNS(DAR)) achieve almost the same performance since they have a similar inter-instance control due to the workflow's big fan-out factor (i.e. the big fan-out factor limits the number of concurrent active instances), leading to similar average DOCs (Fig. 8).

In the Lattice workload, when AIT = 0, OSC is superior to both VNS(DDS) and VNS(DTO), but a little bit inferior to VNS (DAR). However, with the increments of AIT, both VNS(DDS) and VNS(DTO) gradually outpace OSC and catch up with VNS(DAR). In batch Lattice workload (i.e. AIT = 0), due to the low intra-instance concurrency, VNS(DDS) and VNS(DTO) initially have more active instances than OSC has. But due to the constrained storage, some of these active instances quickly become inactive, but hold the allocated storage which cannot be used by other active instances, leading to lower average DOC and performance. This situation is different from VNS(DAR) which reserves sufficient storage for the progress of the active instances.

To summarize this third experiment, we have the following conclusions:

(i) Again, for the Fork&Join workload, the performances of OSC and VNS are independent of AIT.
(ii) For the Lattice workload, the VNS(DDS) and VNS(DTO) can benefit from AIT increments.

When we increase the storage budgets to the double-maximal requirements of OSC (i.e. (256, 344)), as discussed, the performance of BASE and OSC will not change since both policies cannot benefit from this storage extension.

In the Fork&Join workload, the performances of VNS policies are relatively stable as AIT increases from 0 to 800, and their values are around half or double of those when the storage is 128 with respect to the makespan or average DOC. It is expected that all VNS policies can take advantage

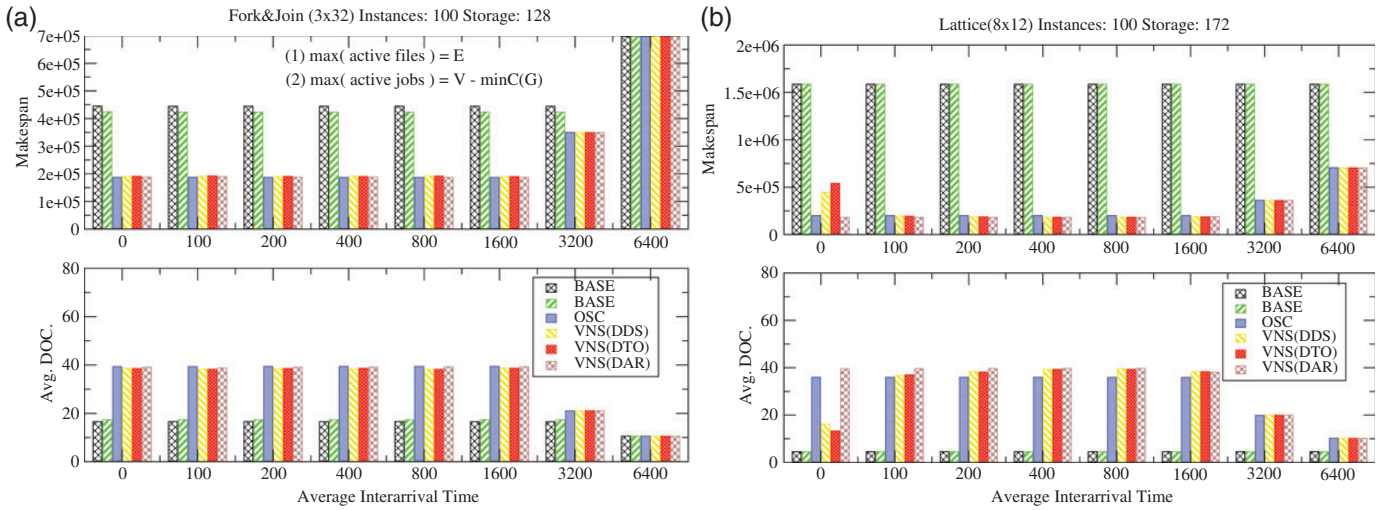

FIGURE 8. Simulation results of Fork&Join and Lattice under the maximal storage requirement of OSC. (**a**) Fork&Join. (**b**) Lattice.

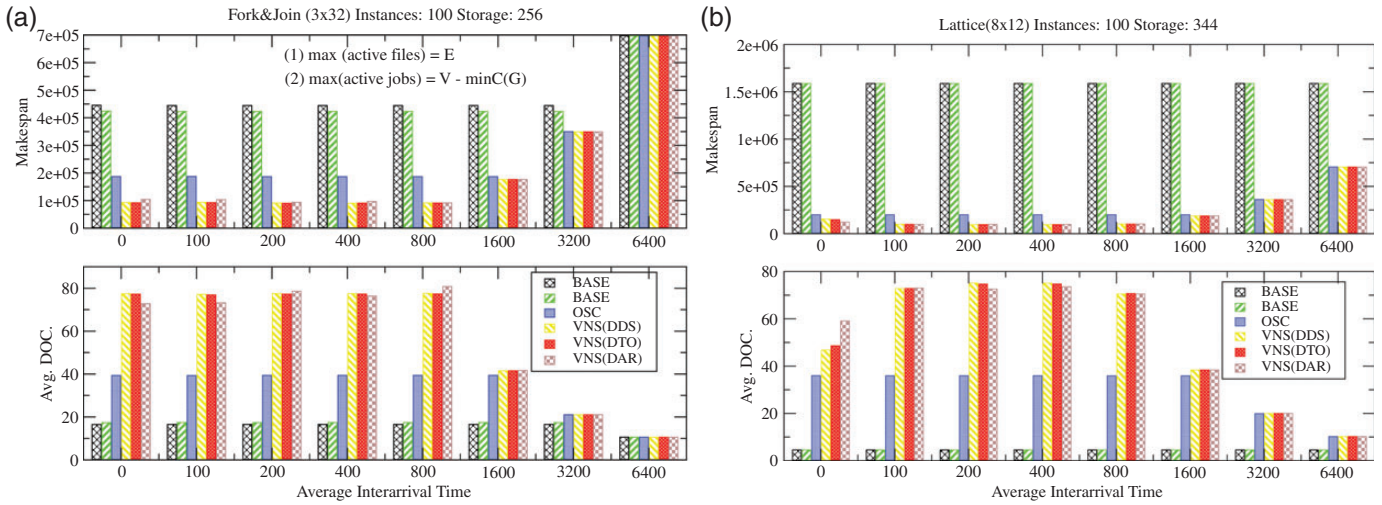

FIGURE 9. Simulation results of Fork&Join and Lattice under the double-maximal storage requirement of OSC. (**a**) Fork&Join. (**b**) Lattice.

of the storage extension to gain performance benefits. As AIT increases over 1600, all VNS policies are reduced to OSC policy since, in the OSC policy, the difference between the time that two consecutive workflow instances can start execution is around $750 \times 2 = 1500$, which is less than AIT ≥ 1600.

For the Lattice workload, we can draw the same conclusions with those when the storage budget is 172 except the situation where AIT $=$ 0 (Fig. 9). Unlike the situation in 172, when the storage budget increases to 344, for the batch workload, VNS(DDS) and VNS(DTO) outperform OSC. This demonstrates again that both VNS policies can take advantage of storage extension to improve computation performance. Another observation is that the performance gaps between the three versions of VNS policies compared with those in the previous situation (i.e, storage budget is 172) are reduced (Fig. 9). This fact shows that, for the Lattice workload, VNS(DDS) and VNS(DTO) can benefit more from the storage extension than VNS(DAR) since, for VNS(DDS) and VNS(DTO), their average DOCs increase $\sim$160% from 172 to 344, whereas the average DOC of VNS(DAR) only increases about 50% in the same situation.

To summarize this above-mentioned experiment, we have the following conclusions:

(i) Compared with OSC, VNS polices can further obtain performance benefits from the storage budget increase.
(ii) For the Lattice workload, VNS (DDS) and VNS(DTO) can benefit more from the storage budget increase than VNS (DAR).

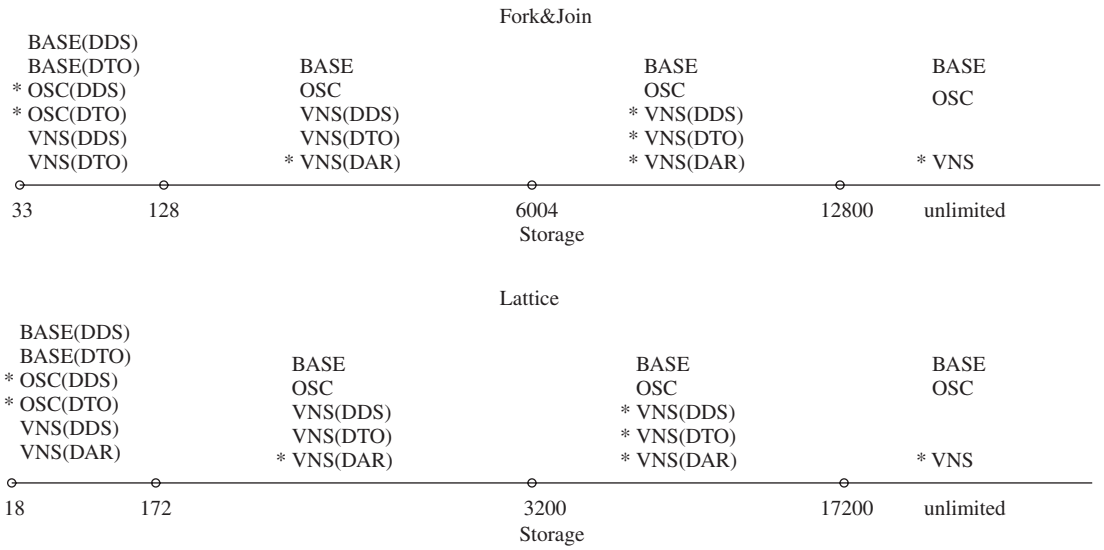

FIGURE 10. The performance leaders (marked ∗) identified when the storage budgets are varied.

7. CONCLUSIONS

Based on the experiment results of our proposed policy and the comparative evaluation results against other policies, we draw further conclusions:

(i) When the storage budget is the bottleneck, namely, not greater than the maximal requirement of BASE policy, regardless of which kinds of deadlock resolution approaches are combined, the performance of all studied policies are indistinguishable for the Fork&Join workload and incomparable for the Lattice workload.

(ii) As long as the storage budget is greater than the maximal requirement of BASE policy, OSC and VNS always perform better than the BASE policy on makespan and storage utilization. More specifically, OSC(DDS) and OSC(DTO) are the performance leaders when the storage budget is less than the requirement of OSC policy, followed by the VNS(DAR). When the storage budget increases over the maximal requirement of VNS, no policy can further improve the performance. All VNS policies are equally effective and exhibit the best performance among all studied policies (Fig. 10).

This phenomenon is further evidenced by comparing the storage utilization of these policies. For both workloads, OSC(DDS) and OSC(DTO) exhibit better performance if the storage budget is not greater than the maximal requirement of OSC. Otherwise, VNS(DAR) performs the best in storage utilization.

(iii) When the given storage budget is not greater than the maximal requirement of OSC, our DDS and DAR algorithms have a similar performance in terms of deadlock resolution for the Fork&Join workload, but the former offers a better Lattice workload than the latter.

After increasing the storage budget over the maximal requirement of OSC, the DAR algorithm joins the competition, and outperforms both the DDS and the DTO algorithms. This phenomenon can also be evidenced by comparing the storage utilization of VNSs combined with the corresponding deadlock resolution algorithms.

ACKNOWLEDGEMENTS

The authors would like to thank anonymous reviewers who gave valuable suggestions that have helped to improve the quality of the manuscript.

REFERENCES

[1] Montage, http://montage.ipac.caltech.edu.

[2] (2014) Sextractor. http://www.astromatic.net/software/sextractor.

[3] Maechling, P. *et al.* (2007) Scec Cybershake Workflows Automating Probabilistic Seismic Hazard Analysis Calculations. In Taylor, I., Deelman, E., Gannon, D. and Shields, M. (eds), *Workflows for e-Science*, pp. 143–163. Springer, London.

[4] Djorgovski, S., Gal, R., Odewahn, S., de Carvalho, R., Brunner, R., Longo, G. and Scaramella, R. (1998) The digital Palomar sky survey (DPOSS). *Wide Field Surv. Cosmol.*, **1**, 10–20.

[5] MSR-TR-2005-10 (2005) Scientific Data Management in the Coming Decade. Technical Report. Microsoft Corporation.

[6] Pandey, S. and Buyya, R. (2008) Scheduling of Scientific Workflows on Data Grids. *Proc. 8th IEEE Int. Symposium on*

Cluster Computing and the Grid, Washington, DC, USA, May 19–22, pp. 548–553. IEEE, Lyon, France.

[7] Wang, Y. and Lu, P. (2011) Dataflow detection and applications to workflow scheduling. *Concurrency Comput. Pract. Exp.*, **23**, 1261–1283.

[8] Bent, J., Thain, D., Arpaci-Dusseau, A., Arpaci-Dusseau, R.H. and Livny, M. (2004) Explicit Control in a Batch-Aware Distributed File System. *Proc. 1st Conf. on Symposium on Networked Systems Design and Implementation*, San Francisco, CA, USA, pp. 365–378. USENIX Association Berkeley, CA.

[9] Ramakrishnan, A., Singh, G., Zhao, H., Deelman, E., Sakellariou, R., Vahi, K., Blackburn, K., Mayers, D. and Samidi, M. (2007) Scheduling Data-Intensive Workflows onto Storage-Constrained Distributed Resources. *Proc. 7th IEEE Int. Symposium on Cluster Computing and the Grid*, May 14–17, pp. 401–409. IEEE, Rio de Janeiro, Brazil.

[10] Zhang, W., Cao, J., Zhong, Y., Liu, L. and Wu, C. (2008) An Integrated Resource Management and Scheduling System for Grid Data Streaming Applications. *Proc. 8th IEEE Int. Symposium on Cluster Computing and the Grid*, Washington, DC, USA, May 19–22, pp. 258–265. IEEE, Lyon, France.

[11] Bharathi, S. and Chervenak, A. (2009) Scheduling Data-Intensive Workflows on Storage Constrained Resources. *Proc. 4th Workshop on Workflows in Support of Large-Scale Science, WORKS'09*, Portland, Oregon, pp. 3:1–3:10. ACM, New York, NY, USA.

[12] Chen, W. and Deelman, E. (2011) Partitioning and Scheduling Workflows Across Multiple Sites with Storage Constraints. *9th Int. Conf. on Parallel Processing and Applied Mathematics (PPAM 2011)*, September 1–14, pp. 11–20. Springer, Berlin, Heidelberg.

[13] Juve, G. and Deelman, E. (2010) Scientific workflows and clouds. *Crossroads*, **16**, 14–18.

[14] Chen, W. and Deelman, E. (2012) Integration of Workflow Partitioning and Resource Provisioning. *Proc. 2012 12th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing, CCGRID'12*, Ottawa, Canada, May 13–16, pp. 764–768. IEEE Computer Society, Washington, DC, USA.

[15] Wang, Y. and Lu, P. (2013) Maximizing active storage resources with deadlock avoidance in workflow-based computations. *IEEE Trans. Comput.*, **62**, 2210–2223.

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

[17] Sima, D., Fountain, T. and Kacsuk, P. (1997) *Advanced Computer Architectures, a Design Space Approach*. Addison Wesley.

[18] Hennessy, J. and Patterson, D. (2005) *Computer Architecture: A Quantitative Approach* (3rd edn). Morgan Kaufmann.

[19] Lam, M. (1988) Software Pipelining: An Effective Scheduling Technique for vliw Machines. *Proc. ACM SIGPLAN 1988 Conf. on Programming Language design and Implementation*, Atlanta, GA, USA, June 20–24, pp. 318–328. ACM, New York, NY, USA.

[20] Rau, B.R. and Glaeser, C.D. (1981). *Proceedings of the 14th Annual Workshop on Microprogramming (MICRO 14)*, Chatham, MA, United States, pp. 183–198. IEEE Press Piscataway, NJ, USA.

[21] Lang, S.-D. (1999) An Extended Banker's Algorithm for Deadlock Avoidance. *IEEE Trans. Softw. Eng.*, **25**, 428–432.

[22] Gburzynski, P. SMURPH, http://www.olsonet.com/pg/PAPERS/side.pdf (accessed October 2, 2012).

[23] Glatard, T., Montagnat, J. and Pennec, X. (2005) Grid-Enabled Workflows for Data Intensive Medical Applications. *18th IEEE Symposium on Computer-Based Medical Systems*, Trinity College Dublin, Ireland, June 23, pp. 537–542. IEEE Computer Society.

[24] Knight, K. and Marcu, D. (2005) Machine Translation in the Year 2004. *In Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, Philadelphia, PA, USA, March 18–23, pp. 965–968. IEEE Computer Society.

[25] Maechling, P.e.a. (2005) Simplifying construction of complex workflows for non-expert users of the Southern California Earthquake Center Community Modeling Environment. *SIGMOD Rec.*, **34**, 24–30.

[26] Rosenberg, A. (2004) On scheduling mesh-structured computations for internet-based computing. *IEEE Trans. Comput.*, **53**, 1176–1186.