A Cleanup Algorithm for Implementing Storage Constraints in Scientific Workflow Executions

Sudarshan Srinivasan
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
Email: email2sudarshan@gmail.com

Gideon Juve, Rafael Ferreira da Silva, Karan Vahi, Ewa Deelman
Information Sciences Institute
University of Southern California
Email: {gideon,rafsilva,vahi,deelman}@isi.edu

***Abstract*—Scientific workflows are often used to automate large-scale data analysis pipelines on clusters, grids, and clouds. However, because workflows can be extremely data-intensive, and are often executed on shared resources, it is critical to be able to limit or minimize the amount of disk space that workflows use on shared storage systems. This paper proposes a novel and simple approach that constrains the amount of storage space used by a workflow by inserting data cleanup tasks into the workflow task graph. Unlike previous solutions, the proposed approach provides guaranteed limits on disk usage, requires no new functionality in the underlying workflow scheduler, and does not require estimates of task runtimes. Experimental results show that this algorithm significantly reduces the number of cleanup tasks added to a workflow and yields better workflow makespans than the strategy currently used by the Pegasus Workflow Management System.**

I. Introduction

Hosted computing infrastructures such as campus clusters, grids and clouds have become daily instruments of scientific research. Applications running on these infrastructures have grown in size and complexity. As a result, many scientists now formulate their computational problems as scientific workflows [1]. Workflows allow researchers to easily express multi-step computational and data manipulation tasks, for example: retrieve data from an instrument or a database, reformat the data, and run an analysis. When executing large, data-intensive workflows composed of hundreds or thousands of tasks that generate terabytes of data, it is important to carefully manage the storage space on the execution resources. Workflow management systems often clean up temporary data only after the workflow finishes. However, in the case of larger workflows the total storage required by a workflow for all inputs and outputs may exceed the available storage space, leading to workflow failures. As a result, it is necessary to remove intermediate data from the storage system as the workflow is executing. This can be achieved by adding in the workflow data cleanup tasks that are responsible for removing intermediate data when it is no longer required. For example, the inputs for a task are no longer required once the task has finished successfully. Similarly, the outputs generated by a task are no longer required once all the tasks that use those outputs have finished successfully, or when data is staged out to permanent storage.

Scheduling scientific workflows under storage constraints has been addressed in previous work [2]–[5]. However, the approaches proposed there either required the development of a new scheduler that is aware of application data and storage resources, or they required accurate task runtime estimates, which are difficult to predict in practice [6]. In addition, online approaches for solving this problem may result in deadlocks, where progress cannot be made without removing data that will be required in the future. Such deadlocks can only be resolved by backtracking and re-executing tasks, which results in wasted time and resources.

In this paper, we propose a novel offline algorithm that implements storage constraints by inserting cleanup tasks into the workflow task graph. The algorithm provides guaranteed limits on the amount of storage space required by the workflow without requiring task runtime estimates or data-aware schedulers. Our approach is based on a static rearrangement of the workflow during a planning step just prior to execution. The algorithm adds cleanup tasks to the workflow, and makes computational tasks dependent on them in a way that ensures sufficient storage space has been freed before a dependant task can begin executing. This approach guarantees that, if the algorithm is able to find a feasible solution, then the workflow will be able to run to completion without exhausting the available storage space.

Inserting cleanup tasks into the workflow has two potentially negative effects: 1) it adds tasks to the workflow that may increase the makespan of the workflow, and 2) it creates dependencies in the workflow that may add bottlenecks and reduce parallelism. In order to understand these issues, we performed simulation experiments using synthetic workflows based on two real science applications. When compared our algorithm with the existing cleanup algorithm [2] used in the Pegasus Workflow Management System [7], [8]. The results show that the new algorithm generates far fewer cleanup tasks and yields better workflow makespans.

The remainder of this paper is organized as follows. Section II summarizes the related work. Section III provides relevant background information and describes the problem in detail. Section IV describes the proposed storage-constrained cleanup algorithm. Section V describes the results of several experiments that were used to evaluate the proposed algorithm. Finally, Section VI concludes the paper and discusses possible future work.

II. Related Work

Ramakrishnan et al. [2] and Singh et al. [3] introduced the concept of cleanup tasks to reduce the amount of peak storage

978-1-4799-7067-4/14 $31.00 © 2014 IEEE
DOI 10.1109/WORKS.2014.8

required by a workflow, and proposed several algorithms for storage-constrained workflow execution. The first algorithm adds cleanup tasks as leaf nodes in the task graph. For each file, the algorithm adds a cleanup task that depends on all the compute tasks that use the file as input. This algorithm reduces the footprint of a workflow, but does not implement storage constraints since the amount of storage used depends on the order in which the tasks are executed. In addition, since it creates a cleanup task for every file in the workflow, this algorithm may result in a large number of cleanup tasks relative to non-cleanup tasks, which may significantly impact the performance of the workflow. Singh et al. developed a variation of this algorithm that adds task clustering to reduce the number of cleanup tasks. This algorithm groups together the cleanup tasks generated by Ramakrishnan's algorithm so that the number of cleanup tasks in the workflow is no more than the number of non-cleanup tasks. Singh's algorithm is the default cleanup algorithm used in The Pegasus Workflow Management System [7], and it is used in this paper for comparison with our approach.

Ramakrishnan et al. [2] also developed a storage constraint algorithm that partitions a task graph across a set of grid sites so that each partition does not exceed the available storage at its assigned site. This algorithm implements storage constraints, but it does not work for single sites or when the total storage available across all sites is less than the maximum workflow footprint. Chen and Deelman [5] proposed a similar algorithm that also attempts to minimize the amount of data transferred between execution sites.

Singh et al. [3] also investigated workflow restructuring to reduce storage requirements. They used an ad-hoc approach, where dependencies were added to the workflow manually to force a specific execution order. This new execution order, when combined with their cleanup algorithm, resulted in a smaller maximum storage footprint. In this paper, we propose an algorithm to achieve similar restructuring in an automated way.

Bharathi et al. [4] proposed several algorithms for implementing storage constraints, including one simple greedy algorithm and two solutions based on metaheuristics. All three algorithms begin with a task graph containing cleanup tasks as proposed by Ramakrishnan et al. and Singh et al. The greedy algorithm identifies non-overlapping partitions of the task graph rooted at each cleanup task. The algorithm examines these partitions at a regular time interval in breadth-first order and releases the next partition for execution when sufficient storage space is available for the entire partition. In the same work, the authors proposed two other approaches based on genetic algorithms: one that considers the ordering of the cleanup tasks only, and one that considers the ordering of all tasks. Only valid topological orderings that use less than the available storage space are allowed, and fitness is defined as a function of makespan and storage space required for execution. All three algorithms required significant changes to the scheduler to make it data-aware. In comparison, the algorithm proposed in this paper requires no changes to the scheduler.

III. Problem Definition

Scientific workflows describe complex, multi-step computational pipelines used to automate computer modeling, simulation, and data processing activities for science and engineering applications. This paper considers a class of workflows that can be represented as a directed acyclic graph (DAG), where each node in the graph is a computational task and the edges between the nodes represent data or control flow dependencies. More specifically, in these workflows a task represents the invocation of a command-line program with a given set of arguments, and each invocation may involve reading several input files and writing several output files. This is the workflow application model supported by Pegasus [7], [8] and many other workflow management systems [9]–[11].

Workflows are often used to automate large computations that may involve thousands of tasks and terabytes of data. Executing such workflows requires a significant amount of temporary disk space to store intermediate data products. These data products are generated by the workflow, but do not need to be saved after the workflow finishes executing. In the case of large, data-intensive workflows, the storage available on the execution site may not be sufficient to store all of the intermediate data required for the workflow to execute to completion. The storage space may be limited by the amount of disk space available on the storage system, or by user quotas. To prevent these problems, *data cleanup* tasks may be inserted into the task graph to remove intermediate data after all the tasks that require it finish execution. In addition, the workflow can be restructured such that non-cleanup tasks depend on cleanup tasks to ensure that storage is freed for non-cleanup tasks before they are allowed to execute. It is possible to use these dependencies to enforce upper limits on the maximum amount of storage used by the workflow.

The addition of cleanup tasks can be performed automatically by the workflow system through static analysis of the task graph. The data cleanup problem with storage constraints can be defined as the problem of adding data cleanup tasks to the workflow task graph, such that the peak storage used does not exceed a given storage limit. Solutions to this problem must ensure that the modified task graph maintains the same ordering of non-cleanup tasks as the original task graph. Adding cleanup tasks to a workflow introduces delays caused by overheads in the execution infrastructure [12], and the addition of dependencies may create bottlenecks that result in reduced parallelism. Therefore, better solutions to this problem will insert cleanup tasks in a way that minimizes the resulting makespan of the modified workflow.

IV. Storage-Constrained Cleanup Algorithm

The algorithm proposed in this paper builds upon the work done by Ramakrishnan et al. [2] and Singh et al. [3], who developed offline algorithms to insert cleanup tasks into workflow task graphs. In Singh's algorithm, the number of cleanup tasks added to the workflow can be very large. In the worst case, it is equal to the number of tasks in the workflow. At the same time, while the algorithm does reduce the peak storage requirements of the workflow, it is unable to provide any guarantees regarding the total storage space that will be used by the workflow at runtime.

Our proposed algorithm attempts to resolve these issues. It inserts cleanup tasks and automatically restructures the task

graph by adding extra edges to ensure that the maximum storage footprint of the workflow does not exceed a given storage constraint. It also attempts to minimize the number of cleanup tasks added to the workflow by inserting cleanup tasks only when they are required. Unlike previous approaches, the storage used by the workflow does not depend on the order in which the workflow execution engine, such as HTCondor DAGMan [13], releases tasks at runtime. The algorithm achieves this by adding extra edges between the cleanup task and the subsequent computational (non-cleanup) tasks. As a result, subsequent computational tasks only start executing once the parent cleanup task has finished. Intuitively, cleanup tasks act as barrier points in the execution of the workflow. They prevent other tasks from executing until sufficient storage has been reclaimed for their outputs.

This algorithm only assumes that data sizes for task inputs and outputs are provided. The decision about where to place cleanup tasks uses a greedy approach based on a simulated execution of the workflow. Tasks are selected using a heuristic based on the size of data generated by the task, and the size of data that can be cleaned up after the task finishes executing. The algorithm identifies potential storage constraint violations by "executing" tasks until the heuristic chooses a task that requires more than the available storage space. At that point, the algorithm adds one or more cleanup tasks to remove all intermediate data that is no longer required. It is important to note that this algorithm does not guarantee that the restructured workflow will run in the minimum storage space possible. Instead, it guarantees that, if it is able to find a valid solution for the given storage constraint, then the restructured workflow will not use more storage than the constraint value. If the algorithm does not find a valid schedule for a given constraint then it will not return a solution. The pseudo code for the storage constraints algorithm is shown in Algorithm 1.

The number of cleanup tasks added and the minimum storage constraint for which a solution can be found, depends on the heuristic employed to select which candidate task from the queue to execute next. We implemented several different heuristics for the GETNEXTCANDIDATETASK procedure based on the amount of storage space that can be freed after executing a task, and the amount of storage space consumed by the task. **Max Freed** is a simple heuristic that selects the task that maximizes the amount of disk space that can be freed. **Min Required** is another simple heuristic that selects the task that has the smallest storage space required. The **Max Required** heuristic is the opposite of **Min Required**. The heuristic selects the task with the largest storage space required, since these tasks are the most difficult to accommodate. The **Balance Factor** heuristic selects the task with the largest *balance factor*, which is defined as the difference between the storage space that can be freed when the task completes and the space consumed by the task when executed. The balance factor can be viewed as an estimate of the impact of running the task on available storage space. If none of the candidate tasks free any storage space, then the task with the largest balance factor is equivalent to the task with the smallest space required. If multiple tasks have the same balance factor, then the one with the smallest required space is chosen.

Algorithm 1 Storage-constrained algorithm.

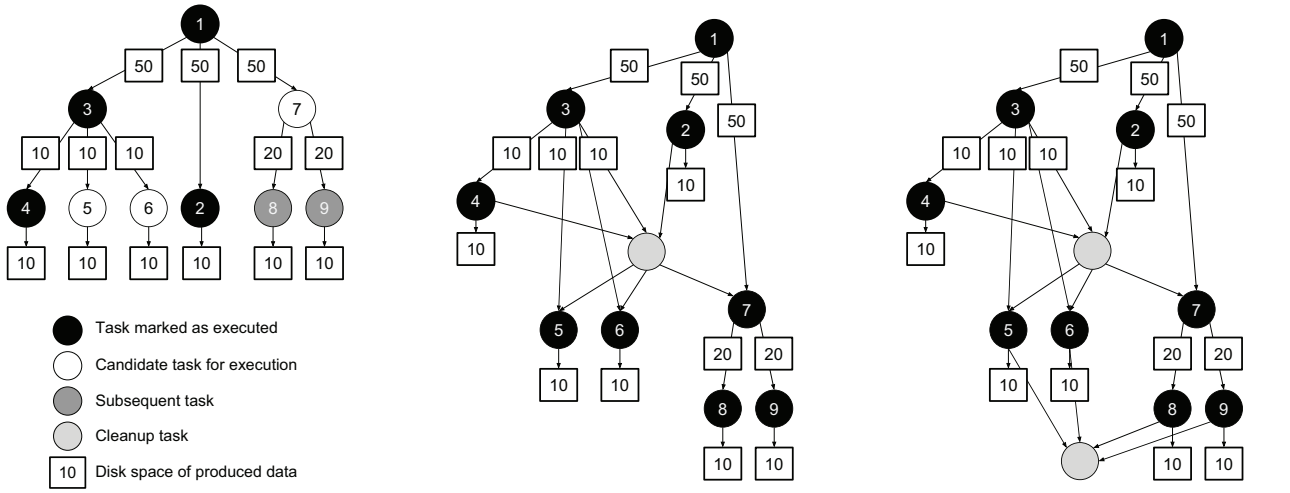
Require: W: workflow with task data size estimates; and S: storage constraint

```
1: procedure SIMULATE(W, S)
2:     used_disk ← 0
3:     avail_space ← S
4:     while there are tasks to run do
5:         t ← GETNEXTCANDIDATETASK(W)
6:         if avail_space ≥ t.req_space then
7:             mark task t as executed
8:             avail_space ← avail_space − t.req_space
9:         else
10:            U ← ADDCLEANUPTASKS(W)
11:            avail_space ← avail_space + storage_freed(U)
12:            if avail_space < t.req_space then
13:                stop workflow execution and return no solution
14:            end if
15:        end if
16:    end while
17:    add a cleanup task to remove all remaining data
18: end procedure
```

Inserting cleanup tasks has the potential to create bottlenecks in the workflow that reduce parallelism and increase the workflow's makespan. To investigate this issue we implemented several different strategies for adding cleanup tasks in the ADDCLEANUPTASKS procedure. The **Single Task** approach adds a single cleanup task to the workflow, sets the parents of the cleanup task to be the non-cleanup tasks that use the data that is cleaned up, and sets the children of the cleanup task to be all of the currently queued tasks. The **Queued Tasks** approach adds cleanup tasks up to the number of tasks in the queue (i.e. candidate tasks for execution), sets the parents of the cleanup tasks to be the non-cleanup tasks that use the data that is cleaned up, and sets the children of the cleanup tasks to be all of the currently queued tasks. The **Random Tasks** approach adds a random number of cleanup tasks between 1 and the number of tasks in the queue. The **Resources Tasks** approach adds cleanup tasks up to the number of resources available for execution. Both *Random Tasks* and *Resources Tasks* approaches follow the same rules to set the parents and children tasks for the cleanup tasks. Note that cleanup tasks are balanced by the size of the data to be removed, except for the *Single Task* approach, which is composed by a single cleanup task.

Regardless of the cleanup strategy used, after all the non-cleanup tasks have been processed, the algorithm adds a final cleanup task to the workflow to remove all remaining data. This final cleanup task appears as a sink node in the graph and depends on all of the non-cleanup, leaf nodes.

Figure 1 illustrates a simple example with a storage constraint of 200 units using the Balance Factor heuristic with a Single Task cleanup strategy. Tasks 1 to 4 have been marked as executed because there is enough space to run them without any cleanup tasks. A storage constraint violation is detected when there is not enough space to run the next queued task selected by the Balance Factor heuristic, which would be Task 7 (Figure 1a). At that point, the algorithm adds one cleanup task to remove all data not required for the remainder of the workflow. In our example, intermediate data between tasks $(1, 2)$, $(1, 3)$, and $(3, 4)$ are removed releasing 110 units (Figure 1b). The queued candidate tasks become children of the cleanup task, and tasks 2, 3, and 4 (which use the data being cleaned up) become parents of the cleanup task. A final cleanup task is added as a sink node for the entire workflow to ensure that all intermediate data is removed from the execution



(a) The algorithm proceeds until there is insufficient disk space to run the next task

(b) A cleanup task is inserted to remove all data that is no longer required

(c) A final cleanup task is inserted to ensure that all intermediate data is removed

Fig. 1: A sample execution of the proposed algorithm with the storage limit set to 200 units using the *Single Task* approach. Note that this algorithm runs during the planning phase and that tasks marked as *executed* have not actually executed; they have merely been marked as executed in the simulated run.

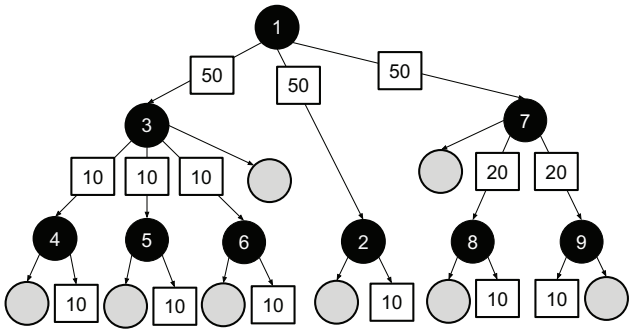


Fig. 2: Workflow from Figure 1 using Singh's cleanup algorithm. In this case, Singh's algorithm adds a cleanup task for almost every non-cleanup task.

site when the workflow finishes (Figure 1c).

Figure 2 illustrates the cleanup jobs that Singh's algorithm would add for the same workflow. In this case, since each intermediate file is only used as input to a single task, Singh's algorithm would add a cleanup task for almost every non-cleanup task in the workflow.

V. Evaluation

This section presents experiments that evaluate the performance of our storage-constrained algorithm and its impact on the storage footprint and makespan of workflow executions. The algorithm was implemented in the Pegasus Workflow Management System as an alternative to the built-in cleanup algorithm [3].

A. Experiment Conditions

Two real scientific workflow applications are used in the experiments. *Montage* [14] is an astronomy application that is used to construct science-grade astronomical image mosaics. The workflow re-projects images to a common plane, adjusts their brightness, and adds them together. *CyberShake* [15] is a seismology application used in probabilistic seismic hazard analysis. The workflow computes seismic hazard curves by computing the effects of many scenario earthquakes on ground motions at a given geographic site. Figures 3 and 4 illustrate small examples of the Montage and CyberShake workflows before cleanup tasks are added, and Table I shows their main characteristics.

We collected execution traces, including scheduling overheads and task runtimes, from real runs of the two workflow applications [6], [16]. These traces were used as input to the workflow generator toolkit [17]–[19] to create a collection of synthetic workflows. The workflow generator uses information gathered from real workflow executions to construct realistic synthetic workflows. For each workflow application, we generated a set of 100 synthetic workflows with 1000 tasks each. The storage required to execute the largest task (total size of inputs and outputs) is used as a lower bound on the minimum amount of storage space required to execute a workflow. Note that this value is an absolute lower bound and may not be achievable in practice. For Montage workflows, the largest task uses approximately 31% of the total storage required by the workflow, and for the CyberShake workflows, the largest task uses approximately 24% of the total.

Three sets of experiments were conducted to evaluate our storage-constrained algorithm and compare it with the algorithm developed by Singh et al. All experiments were performed using a version of Pegasus that was modified to

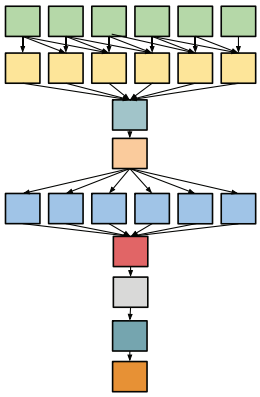


Fig. 3: Small example of the Montage workflow colored by task type.

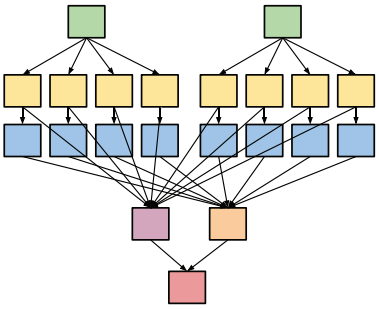


Fig. 4: Small example of the CyberShake workflow colored by task type.

Application	Jobs	Avg. Exec. Time	Avg. Data Size
Montage	158	11.60s	∼4 MB
CyberShake	1675940	460840.60s	∼132 MB

TABLE I: Characteristics of the workflow applications.

include the new algorithm. Experiment 1 evaluates the effect of storage constraints on the number of cleanup tasks added to the workflow, the impact of the number of resources on the workflow's makespan, and whether storage constraints are met when the number of resources is varied. Experiment 2 evaluates the effects of using different heuristics for the GETNEXTCANDIDATETASK procedure on the workflow's makespan and the peak storage space actually used by the workflow. Experiment 3 evaluates the impact of using different strategies for inserting cleanup tasks in the ADDCLEANUPTASKS procedure on the workflow's makespan.

After planning the synthetic workflows with our modified version of Pegasus, a simulator based on the CloudSim toolkit [20] was used to simulate the execution of the synthetic workflows. The simulator prioritizes cleanup tasks over non-cleanup tasks to ensure that storage is freed as soon as possible. The choice of the next task to execute, from a ready task set with the highest priority, is random. The simulator also takes into account scheduling overheads by using overhead values measured in [12]. Scheduling overhead accounts for approximately 25% of the makespan of the Montage workflow, and approximately 12% of the makespan of the CyberShake workflow. The simulator uses these values to estimate the overhead when scheduling a task.

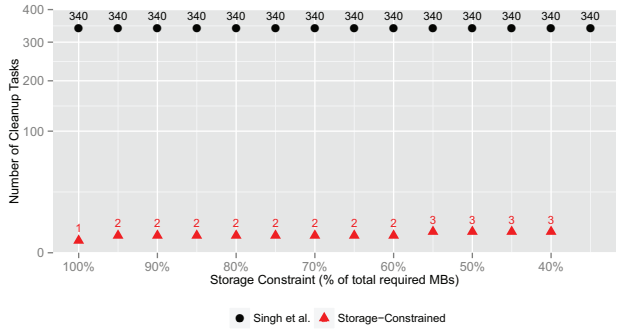
B. Results and Discussion

Experiment 1. This experiment set is composed of three sub-experiments for each application. The first experiment evaluates the impact of storage constraints on the number of cleanup tasks added to the workflow. The storage constraint is set to be a percentage of the total storage required by all tasks in the workflow. Initially, the storage constraint is set to 100% (i.e. no cleanup tasks would be required to execute the workflow), and it is decreased until the storage constrained algorithm is unable to find a solution. Note that the storage constraint has no impact on the behavior of Singh's algorithm.

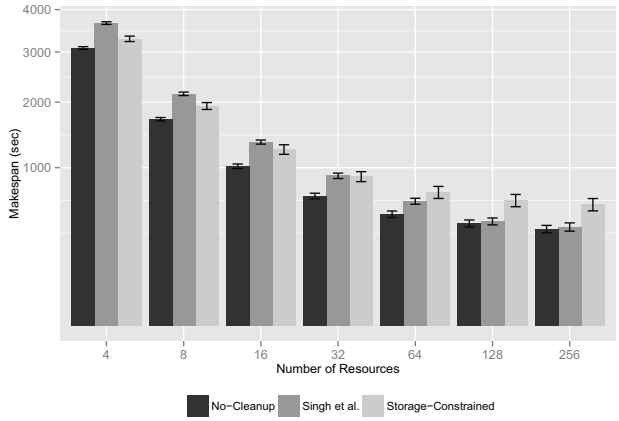
Figure 5a shows the average number of cleanup tasks for the Montage workflow. The minimum storage constraint achieved by the storage-constrained algorithm is 40%, while Singh's algorithm reaches 35% (the lower bound is no less than 31%). The storage-constrained algorithm requires only 2 cleanup tasks to guarantee the storage constraint for the majority of constraint values, and 3 cleanup tasks are required only when the constraint is below 55%. As Singh's algorithm does not consider storage constraints, the number of cleanup tasks remains constant at 340 tasks regardless of the constraint specified. Figure 6a shows the number of cleanup tasks for the CyberShake workflow. The lowest storage constraint achieved by the storage-constrained algorithm is 30%, while Singh's algorithm reaches 25% (the lower bound is no less than 24%). Similarly, our proposed algorithm significantly reduces the number of cleanup tasks from 1001 generated by Singh's algorithm to only 2 for the majority of constraint values. As the constraint decreases, the storage constrained algorithm cleans up more frequently (adding up to 6 cleanup tasks) until it is finally unable to find a solution.

The second experiment evaluates the impact of the number of resources on the workflow's makespan. The number of resources ranges from 4 to 256, and the storage constraint is fixed at 75%. For comparison, the results include the makespan for runs without cleanup tasks (`No-Cleanup`). Note that runs without cleanup tasks and those using Singh's algorithm will violate the storage constraint for some cases in this experiment since both do not consider storage constraints, but runs using the storage-constrained algorithm will not.

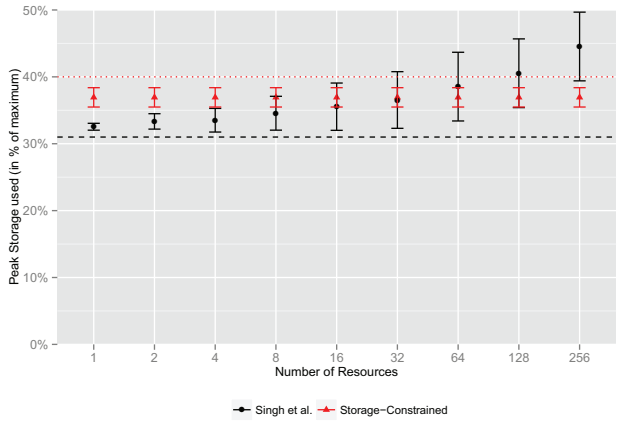
For the CyberShake workflow (Figure 6b), the storage-constrained algorithm produces smaller makespans than Singh's algorithm in all cases. This performance advantage is simply a result of the storage-constrained algorithm generating fewer cleanup tasks to execute. For the Montage workflow (Figure 5b), the storage-constrained algorithm yields smaller makespans in cases where fewer resources are available, while Singh's algorithm yields smaller makespans in cases where more resources are available. This is likely due to the way in which the storage-constrained algorithm adds dependencies between cleanup tasks and non-cleanup tasks, which tends to reduce the parallelism of the workflow. In comparison, Singh's algorithm always adds cleanup tasks as leaf nodes in the task graph, so it has no negative impact on the parallelism



(a) Number of cleanup tasks added for different storage constraint values

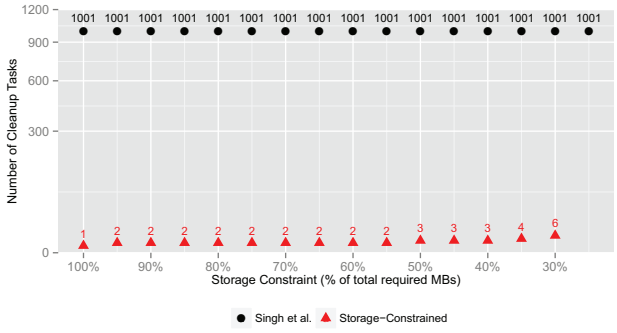


(b) Average makespan for a given number of resources with a storage constraint of 75%

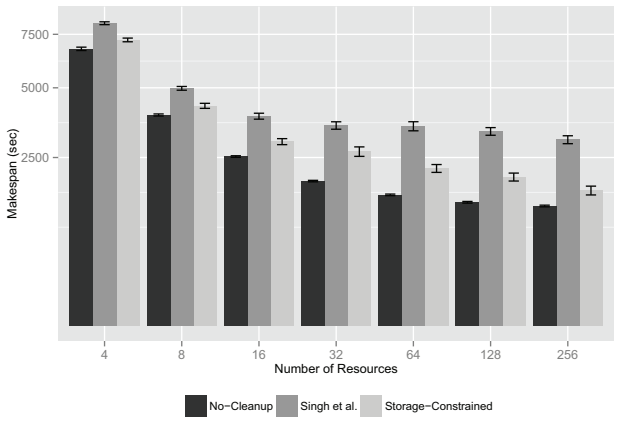


(c) Peak storage used (in % of maximum) for a given number of resources with a storage constraint of 40% (dotted red line). The dashed black line represents a lower bound on peak storage of 31%.

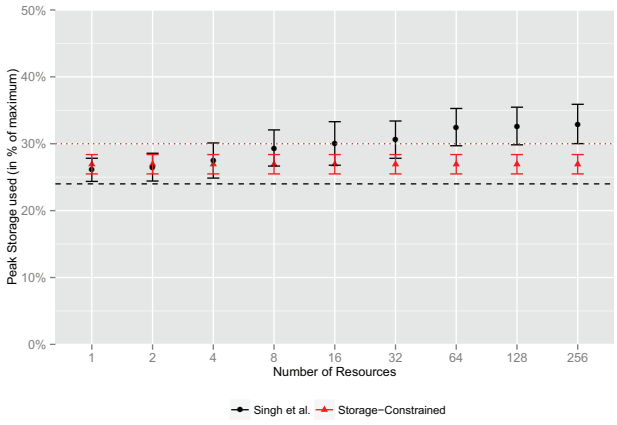
Fig. 5: Experiment 1: results for the *Montage* application with 100 workflows composed of 1000 tasks each.



(a) Number of cleanup tasks added for different storage constraint values



(b) Average makespan for a given number of resources with a storage constraint of 75%



(c) Peak storage used (in % of maximum) for a given number of resources with a storage constraint of 30% (dotted red line). The dashed black line represents a lower bound on peak storage used of 24%.

Fig. 6: Experiment 1: results for the *CyberShake* application with 100 workflows composed of 1000 tasks each.

of the workflow. In cases where resources are scarce, the storage constrained-algorithm performs better because it adds fewer cleanup tasks, but when resources are plentiful, Singh's algorithm performs better because it is able to execute more tasks in parallel.

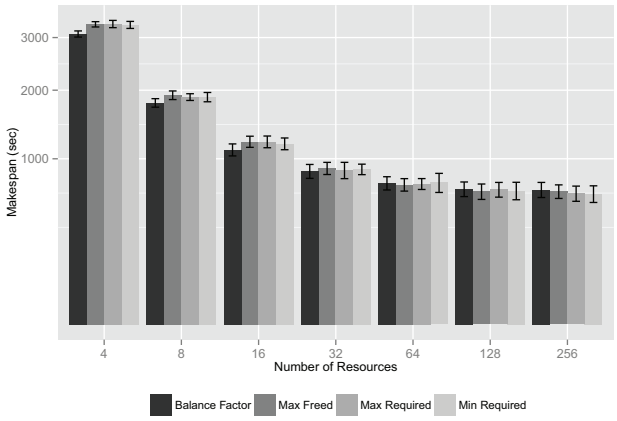
The last experiment evaluates whether storage constraints are met when the number of resources available to execute the workflow increases from 1 to 256. In this experiment the storage constraint is set to the lowest value for which the storage-constrained algorithm was able to find a solution, which is 40% for Montage, and 30% for CyberShake. Figures 5c and 6c show the peak storage space actually used by the workflow at runtime. For comparison, the figures also include the storage constraint (dotted red line) and the average space required for the largest task in the workflow (dotted black line), which is a lower bound on the amount of storage required to execute the workflow.

The figures show that using the storage-constrained algorithm results in disk usage that does not exceed the storage constraint, while Singh's algorithm exceeds the constraint when the number of resources exceeds a certain threshold (around 32 in the case of Montage and around 4 in the case of CyberShake). Since the simulation prioritizes cleanup tasks over non-cleanup tasks, Singh's algorithm is able to stay under the storage constraint for low numbers of resources, because only a few non-cleanup tasks are able to generate data before a higher-priority cleanup task is released to remove data. When the number of resources increases, more non-cleanup tasks are able to run in parallel, and the cleanup tasks are not able to maintain lower levels of disk usage. In comparison, the storage-constrained algorithm is able to keep disk usage below the constraint regardless of the number of resources available to execute the workflow, because the dependencies added by the algorithm prevent a task from running until sufficient storage space has been freed for its outputs.

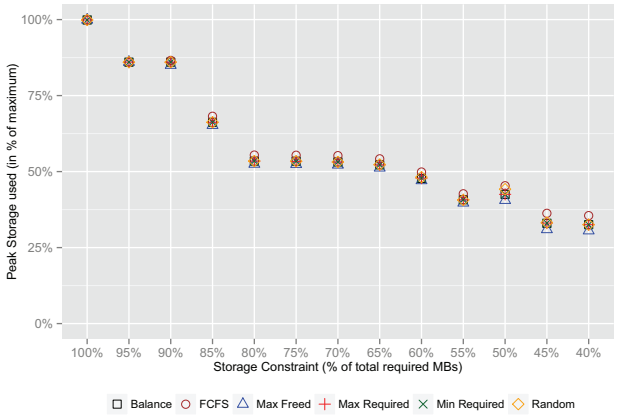
Experiment 2. This experiment set is composed of two sub-experiments for each application. The first experiment evaluates the impact of using different heuristics to select a candidate task from the queue on the workflow's makespan. The number of resources ranges from 4 to 256, and the storage constraint is fixed to the lowest value achieved by our storage-constrained algorithm (40% for the Montage workflow, and 30% for the CyberShake workflow).

For the Montage workflow (Figure 7a), the *Balance Factor* approach produces slightly smaller makespans when resources are scarce (4 to 16 resources). For larger amounts of resources the impact on the produced makespans is negligible independently of the heuristic used. Similarly, for the CyberShake workflow (Figure 8a), the *Balance Factor* approach produces slightly smaller makespans in all cases. This insignificant difference in the workflow's makespans is likely due to the way extra edges are added between the cleanup tasks and the subsequent computational tasks, which tends to be the same regardless of the selected candidate task.

The second experiment evaluates the impact of the various heuristics for selecting the candidate task on the peak storage space actually used by the workflow. Figure 7b shows the average peak storage used (in % of maximum) for a given storage constraint for the Montage workflow (error bars are



(a) Average makespan for a given number of resources with a storage constraint of 40%

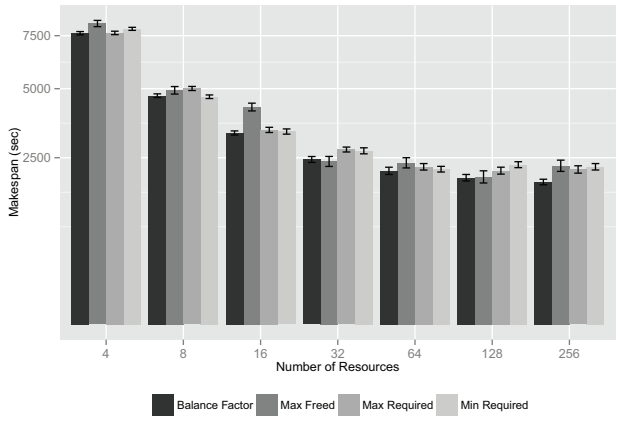


(b) Peak storage used (in % of maximum) for a given storage constraint

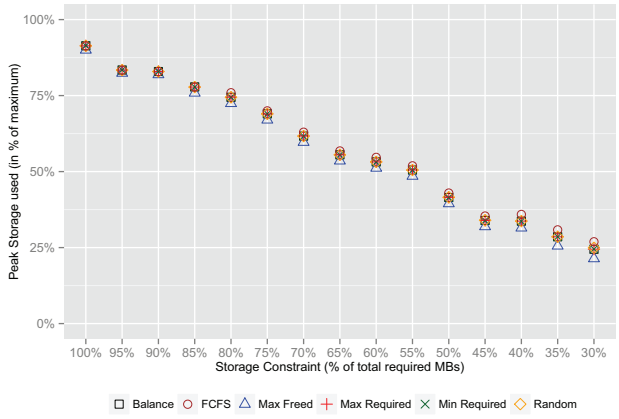
Fig. 7: Experiment 2: results for the *Montage* application with 100 workflows composed of 1000 tasks each.

not shown as they are under 1%). In addition to the 4 proposed heuristics, we also implemented *first-come first-served* (FCFS) and random heuristics. Results show that when the storage constraint is lax, all heuristics behave the same. For stricter constraint values, a small difference is observed: the FCFS heuristic consumes a larger storage footprint, while the `Max Freed` heuristic has a smaller footprint. Figure 8b shows the average peak storage used for the CyberShake workflow. Similar to the Montage experiment, the difference in the peak storage space used is negligible in most cases, with a slight increase in the peak value observed when using the FCFS heuristic, and a smaller footprint with the `Max Freed` heuristic.

Experiment 3. Figures 9 and 10 show the average makespan for a given number of resources when using different strategies for adding cleanup tasks. For the Montage workflow (Figure 9), the variation in the workflow's makespan is negligible regardless of the strategy used. Since cleanup tasks are often added



(a) Average makespan for a given number of resources with a storage constraint of 30%



(b) Peak storage used (in % of maximum) for a given storage constraint

Fig. 8: Experiment 2: results for the *CyberShake* application with 100 workflows composed of 1000 tasks each.

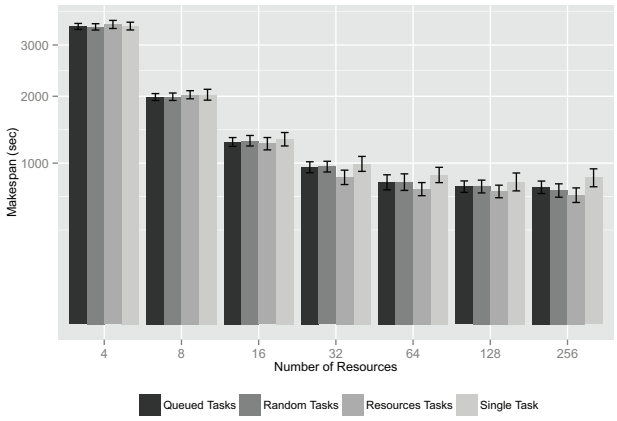


Fig. 9: Experiment 3: average makespan for a given number of resources with a storage constraint of 40% for the *Montage* application with 100 workflows composed of 1000 tasks each.

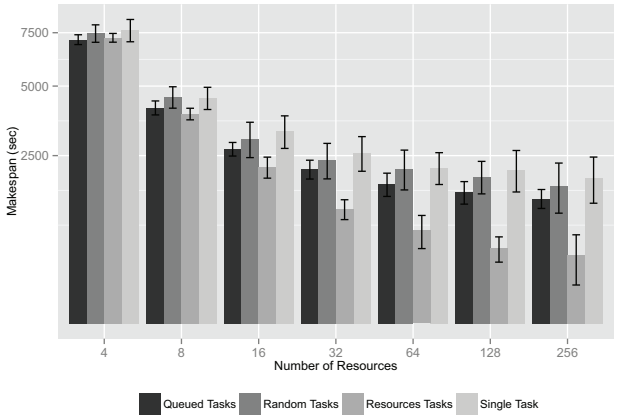


Fig. 10: Experiment 3: average makespan for a given number of resources with a storage constraint of 30% for the *CyberShake* application with 100 workflows composed of 1000 tasks each.

into one of the pipelines (see Montage workflow structure on Figure 3), the gain from parallelizing cleanup tasks is minimal when compared to the overall workflow's makespan. However, for the CyberShake workflow (Figure 10), the `Resources Tasks` approach yields speedup values up to a factor of 2 when compared to the `Single Task` approach. Using multiple cleanup tasks increases the parallelism of the workflow when compared to the `Single Task` strategy. Since the `Queued Tasks` and `Random Tasks` strategies are driven by the number of queued tasks, which is often higher than the number of available resources, the amount of cleanup tasks added by these strategies adds a significant overhead to the workflow execution. `Queued Tasks` and `Random Tasks` might produce similar results to the `Resources Tasks` strategy when the number of available resources is similar to the number of queued tasks.

VI. Conclusions

In this work, we proposed, implemented, and experimentally validated a novel and simple algorithm to implement storage constraints for scientific workflows modeled as a task graph. This algorithm inserts data cleanup tasks into the workflow task graph to guarantee that the storage footprint of the workflow's execution never exceeds a user-supplied limit, regardless of the scheduling algorithm used. We implemented this algorithm in the Pegasus workflow management system, but it can be applied to any workflow system that models workflows as a directed acyclic task graph.

The algorithm was evaluated through simulation using synthetic workflows based on two real workflow applications. The experimental results show that our algorithm guarantees that storage constraints are met regardless of the number

of resources available to execute the workflow. In addition, the results show that, in comparison to an existing cleanup algorithm that does not implement storage constraints, the new algorithm significantly reduces the number of cleanup tasks inserted into the workflow, and does not cause an increase in workflow makespan for the majority of cases examined. In fact, the makespan is reduced in many cases due to the reduction in the number of cleanup tasks.

We also proposed and evaluated several simple heuristics to select which candidate task from the queue to execute, and we varied strategies for adding cleanup tasks. Experimental results show that the workflow's makespan and footprint have no impact regardless of the heuristic used to select the candidate task. However, the makespan is significantly impacted by the number of cleanup tasks inserted.

Although the current algorithm achieves the primary goal of implementing storage constraints, there are several ways in which it could be improved. The current implementation assumes that the workflow runs on a single execution site, but in practice workflows are sometimes executed across several sites. Addressing this use-case would require extensions to the algorithm to support cross-site optimization. In addition, the algorithm's decisions depend heavily on the number of cleanup tasks added. Future work is required to investigate alternative strategies that may result in better parallelism and improved makespan. For example, it may be possible to improve parallelism by analyzing the data flow to reduce dependencies between cleanup and non-cleanup tasks. It may also be useful to select only a subset of the unused data for cleanup, instead of removing all of it, which could result in shorter duration cleanup tasks and more parallelism. Finally, we plan to integrate a version of this algorithm into a future production release of Pegasus WMS.

ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under the SI^2-SSI program, award number 1148515. Sudarshan Srinivasan was supported by the Viterbi-India Summer Research Program, which is funded jointly by the Indo-US Science and Technology Forum (IUSSTF) and the USC Viterbi School of Engineering.

REFERENCES

[1] I. Taylor, E. Deelman, D. Gannon, and M. Shields, *Workflows for e-Science*. Springer, 2007.

[2] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi, "Scheduling data-intensive workflows onto storage-constrained distributed resources," in *Seventh IEEE International Symposium on Cluster Computing and the Grid*, 2007.

[3] 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," *Scientific Programming*, vol. 15, no. 4, pp. 249–268, 2007.

[4] S. Bharathi and A. Chervenak, "Scheduling data-intensive workflows on storage constrained resources," in *4th Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2009.

[5] W. Chen and E. Deelman, "Partitioning and scheduling workflows across multiple sites with storage constraints," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, 2012, vol. 7204, pp. 11–20.

[6] R. Ferreira da Silva, G. Juve, E. Deelman, T. Glatard, F. Desprez, D. Thain, B. Tovar, and M. Livny, "Toward fine-grained online task characteristics estimation in scientific workflows," in *8th Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2013.

[7] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[8] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, p. to appear, 2014.

[9] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. Seragiotto, Jr., and H.-L. Truong, "Askalon: a tool set for cluster and grid computing: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 2-4, pp. 143–169, 2005.

[10] T. Oinn and et. al., "Taverna: lessons in creating a workflow environment for the life sciences: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 18, no. 10, pp. 1067–1100, 2006.

[11] M. Abouelhoda, S. Issa, and M. Ghanem, "Tavaxy: Integrating taverna and galaxy workflows with cloud computing support," *BMC Bioinformatics*, vol. 13, no. 1, p. 77, 2012.

[12] W. Chen and E. Deelman, "Workflow overhead analysis and optimizations," in *6th Workshop on Workflows in Support of Large-scale Science (WORKS)*, 2011.

[13] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger, "Workflow in condor," in *Workflows for e-Science*, I. Taylor, E. Deelman, D. Gannon, and M. Shields, Eds. Springer Press, 2007.

[14] G. B. Berriman, E. Deelman, J. C. Good, J. C. Jacob, D. S. Katz, C. Kesselman, A. C. Laity, T. A. Prince, G. Singh, and M. Su, "Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand," in *SPIE Conference on Astronomical Telescopes and Instrumentation*, vol. 5493, 2004, pp. 221–232.

[15] R. Graves, T. Jordan, S. Callaghan, E. Deelman, E. Field, G. Juve, C. Kesselman, P. Maechling, G. Mehta, K. Milner, D. Okaya, P. Small, and K. Vahi, "CyberShake: A Physics-Based Seismic Hazard Model for Southern California," *Pure and Applied Geophysics*, vol. 168, no. 3-4, pp. 367–381, 2011.

[16] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013.

[17] "Workflow Generator," https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator.

[18] "Workflow Archive," http://www.workflowarchive.org.

[19] R. Ferreira da Silva, W. Chen, G. Juve, K. Vahi, and E. Deelman, "Community resources for enabling and evaluating research on scientific workflows," in *10th IEEE International Conference on e-Science*, ser. eScience'14, 2014, p. to appear.

[20] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.