# Overhead Robust Scheduling in Scientific Workflows

Weiwei Chen*, Rafael Ferreira da Silva‡, Ewa Deelman*,
*University of Southern California, Information Sciences Institute, Marina Del Rey, CA, USA
{wchen,deelman}@isi.edu
‡University of Lyon, CNRS, INSERM, CREATIS, Villeurbanne, France
rafael.silva@creatis.insa-lyon.fr

***Abstract*—Workflow overheads play an important role in workflow performance particularly the scheduling performance. However, existing task scheduling strategies only provide a coarse-grained approach that relies on an over-simplified workflow model that ignores or underestimate the impact of overheads. In this work, we first identify the major overhead patterns in a general workflow management system. Next, we evaluate the performance of existing scheduling heuristics while varying overhead parameters (duration, etc.). Finally, we propose overhead robust heuristics that leverage overhead patterns even without a overhead duration prediction to improve the reliability of scheduling algorithms. A trace-based simulation shows there are significant difference in terms of overhead robustness of these algorithms.**

**1. show it has strong influence 2. estimation of overhead is different 3. overhead friendly scheduling we utlize the knowledge of the pattern 1 matlab problem show the influence of adherence problem first**

***Keywords*—scientific workflow, scheduling, system overhead, log analysis**

## I. Introduction

Many computational scientists develop and use large-scale, loosely-coupled applications that are often structured as scientific workflows, which consist of many computational tasks with data dependencies between them. When executing these applications on a multi-machine distributed environment, such as the Grid or the Cloud, significant system overheads may exist [1] and the problem of choosing robust schedules becomes more and more important. Traditionally, a carefully crafted schedule is based on deterministic or statistic estimates for the execution time of computational activities that compose a workflow. However, in such environment, this traditional approach may prove to be grossly inefficient, as a result of various unpredictable overheads that may occur at runtime. Thus, to mitigate the impact of uncertain overheads, it is necessary to choose a schedule that guarantees overhead robustness, that is, a schedule that is affected as little as possible by various overhead changes.

There are several ways to achieve overhead robustness. A first approach is to integrate the overhead estimation into the job scheduling problem. A static or statistic estimation of communication cost or data transfer delay [2], [3] has been considered in the scheduling problem. Once we have the deterministic or statistic information of overheads, we can treat the system overhead as computational activities and the goal is to minimize the overall runtime including overhead duration. However, this approach only applies to the estimation of data transfer delay since the highly unpredictable variability and variety of other overheads make it a challenging work and not efficient in practice. Our prior work [1] has shown the variation of overheads may be comparable to the job runtime and thus makes it unrealistic in a real environment.

A significant amount of work [2]–[5] in the literature has focused on proposing algorithms that are aware of the dynamic changes of runtime environments. Task rescheduling [6], [7] is a typical approach that dynamically allocates tasks to an idle processor in order to take into account information that has been made available during the execution. Specifically, resource load [2] can be used to estimate the variance. However, rescheduling a task is costly as it implies some extra communication and synchronization costs. Relevant studies [6] indicate that it is important to have a static schedule with good properties before the start of the execution. Therefore, even if a dynamic strategy is used, a good initial placement would reduce the possibility of making a bad decision.

Another approach is to overestimate the execution time of individual jobs. This results in a waste of resources as it induces a lot of idle time during the execution, if the overhead is shorter than the estimation. Second,the overheads do not simply work as an attachment to the job runtime and it involves a lot more complicated patterns such as periodicity [1].

In this paper, we first present our work on evaluating the overhead robustness of scheduling heuristics and we indicate a list of heuristics that are overhead robust even without an estimate of the overhead duration. Second, since the estimate of overhead duration is difficult, we develop new heuristics that leverage the pattern information of workflow overheads, which represents a new approach to design algorithms. To the best of our knowledge, so far, no study has systematically tried to evaluate the scheduling heuristics with respect to the overhead robustness. The next Section gives an overview of the related work, Section III presents our workflow and execution environment models, Section **??** details our heuristics and algorithms, Section VI reports experiments and results, and the paper closes with a discussion and conclusions.

## II. Related Work

Some work in the literature has attempted to define and model robustness. In [8], the authors propose a general method to define a metric for robustness. First, a performance metric is chosen. In our case, this performance metric is the overall runtime including overhead duration as we want the execution time of an application to be as stable as possible. Second, one has to identify the parameters that make the performance metric uncertain. In our case, it is the duration of the individual overheads. Third, one needs to find how a modification of these parameters changes the value of the performance metric. In our case, the answer is, as an increase of the overhead generally implies an increase of the overall runtime. A schedule $A$ is said to be more robust than another schedule $B$ if the variation for $A$ is larger than that for $B$. Following this approach, Canon [9] analyzed the robustness of 20 static DAG scheduling heuristics using a metric for robustness the standard deviation of the makespan over a large number of measurement. Braun et al. [10] evaluated 11 heuristics examined and for the cases studied there, the relatively simple Min-min heuristic performs well in comparison to the other techniques. In comparison, we focus on varying the parameters related to overhead instead of computational tasks.

A plethora of studies on task scheduling [2], [3], [5], [11] have been developed in the distributed and parallel computing domains. Many of these schedulers have been extended to consider both the computational cost and communication cost. A static or statistic estimation of communication cost or data transfer delay [2], [3] has been considered in the scheduling problem. In contrast, we focus on the scheduling overheads that have been ignored or underestimated for long and we demonstrate how their unique timeline patterns influence the overhead robustness.

Workflow patterns [12], [13] are used to capture and abstract the common structure within a workflow and they give insights on designing new workflows and optimization methods. Yu [12] proposed a taxonomy that characterizes and classifies various approaches for building and executing workflows on Grids. They also provided a survey of several representative Grid workflow systems developed by various projects world-wide to demonstrate the comprehensiveness of the taxonomy. Juve [13] provided a characterization of workflow from 6 scientific applications and obtained task-level performance metrics (I/O, CPU and memory consumption). They also presented an execution profile for each workflow running at a typical scale and managed by the Pegasus workflow management system [14]. Compared to them, we aim to discover the common patterns exist in system overheads while executing scientific workflows. We also leverage this knowledge to evaluate the overhead robustness of existing heuristics and develop new heuristics.

Overhead analysis [**?**] [1] is a topic of great interest in the grid community. Stratan [**?**] evaluates workflow engines including DAGMan/Condor and Karajan/Globus in a real-world grid environment. Sonmez [**?**] investigated the prediction of the queue delay in grids and assessed the performance and benefit of predicting queue delays based on traces gathered from various resource and production grid environments. Prodan [**?**] offers a grid workflow overhead classification and a systematic measurement of overheads. Our prior work [1] further investigated the major overheads and their relationship with different optimization techniques. In this paper, we leverage these knowledge to enhance the existing scheduling heuristics and provide insights on designing new algorithms.

Opportunistic Load Balancing, Minimum Execution Time, Minimum Completion Time, Min-min, Max-min, Duplex, Genetic Algorithm, Simulated Annealing, Genetic Simulated Annealing, Tabu, and A*

The 11 static mapping heuristics were evaluated using simulated execution times for an HC environment. Because these are static heuristics, it is assumed that an accurate estimate of the expected execution time for each task on each machine is known prior to execution and contained within a ETC (expected time to compute) matrix. The assumption that these estimated expected execution times are known is commonly made when studying mapping heuristics for HC systems (e.g., [19, 26, 40]). (Approaches for doing this estimation based on task profiling and analytical benchmarking are discussed in [27, 30, 37].)

## III. Model and Design

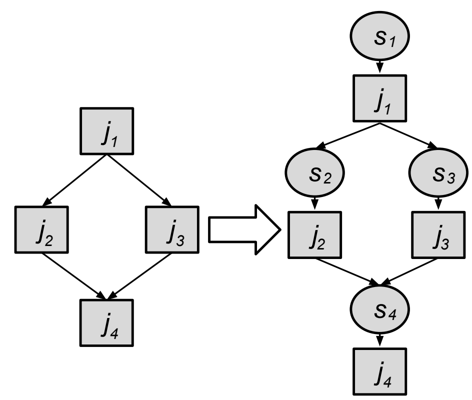


Fig. 1: Extending DAG to o-DAG.

A workflow is modeled as a Directed Acyclic Graph (DAG) as shown in 1. Each node in the DAG often represents a workflow job ($j$), and the edges represent dependencies between the jobs that constrain the order in which the jobs are executed. Dependencies typically represent data-flow dependencies in the application, where the output files produced by one job are used as inputs of another job. Each job is a single execution unit and it may contains one or multiple task, which is a program and a set of parameters that need to be executed. Fig. 1 (left) shows an illustration of a DAG composed by four jobs. This model fits several workflow management systems such as Pegasus [**?**], Askalon [15], and Taverna [16].

Fig. 2 shows a typical workflow execution environment. The submit host prepares a workflow for execution (clustering, mapping, etc.), and worker nodes, at an execution site, execute jobs individually. The main components are introduced below:

*a) Workflow Mapper:* generates an executable workflow based on an abstract workflow provided by the user or workflow composition system. It also restructures the workflow to optimize performance and adds jobs for data management and provenance information generation.

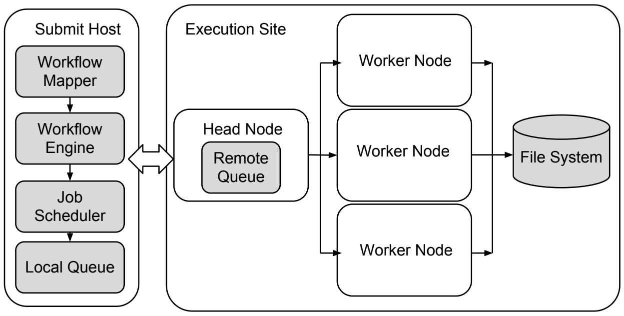


Fig. 2: A workflow system model.

*b) Workflow Engine:* executes jobs defined by the workflow in order of their dependencies. Only jobs that have all their parent jobs completed are submitted to the Job Scheduler. Workflow Engine relies on the resources (compute, storage, and network) defined in the executable workflow to perform the necessary actions. The time period when a job is free (all of its parents have completed successfully) to when it is submitted to the job scheduler is denoted the workflow engine delay. The workflow engine delay is usually configured by users to assure that the entire workflow scheduling and execution system is not overloaded.

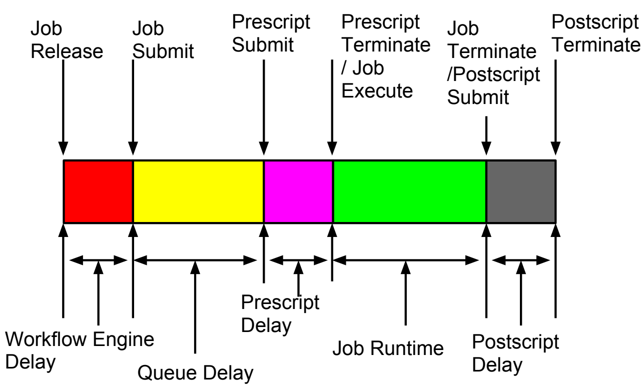


Fig. 3: Overhead Classification

*c) Job Scheduler and Local Queue:* manage individual workflow jobs and supervise their execution on local and remote resources. The time period when a job is submitted to the job scheduler to when the job starts its execution in a worker node is denoted the queue delay. It reflects both the efficiency of the job scheduler and the resource availability.

The execution of a job is comprised of a series of events as shown in Figure 3 and they are defined as:

1) Job Release is defined as the time when the workflow engine identifies that a job is ready to be submitted (when its parents have successfully completed).
2) Job Submit is defined as the time when the workflow engine submits a job to the local queue.
3) Job Execute is defined as the time when the workflow engine sees a job is being executed.
4) Task Execute is defined as the time when the job wrapper sees a task is being executed.
5) Pre/Postscript Start is defined as the time when the workflow engine starts to execute a pre/postscript.
6) Pre/Postscript Terminate is defined as the time when the pre/postscript returns a status code (success or failure).

Figure 3 shows a typical timeline of overheads and runtime in a compute job. We do not specify the data transfer delay in this timeline because data transfer is handled by data transfer jobs (stage-in and stage-out jobs).

As shown in our prior work [1], we have classified workflow overheads into three categories as follows.

1) Workflow Engine Delay measures the time between when the last parent job of a job completes and the time when the job gets submitted to the local queue. The completion time of the last parent job means this job is released to the ready queue and is waiting for resources to be assigned to it. The workflow engine delay reflects the efficiency of a workflow engine (i.e., DAGMan [**?**]).
2) Queue Delay is defined as the time between the submission of a job by the workflow engine to the local queue and the time the local scheduler sees the job running. This overhead reflects the efficiency of the local workflow scheduler (e.g. Condor [**?**]) to execute a job and the availability of resources for the execution of this job.
3) Pre/Postscript Delay is the time taken to execute a lightweight script under some execution systems before/after the execution of a job. For example, prescripts prepare working environment before the execution of a job starts and postscripts examine the status code of a job after the computational part of this job is done.

The overhead aware DAG model (o-DAG) we use in this work is an extension of the traditional DAG model. System overheads play an important role in workflow execution and constitute a major part of the overall runtime when tasks are poorly clustered. Fig. 1 shows how we augment a DAG to be an o-DAG with the capability to represent system overheads ($s$) such as workflow engine delay and queue delay.

In summary, an o-DAG representation allows the specification of high level system overhead details, which is more suitable for the study of overhead aware scheduling.

## IV. Overhead Patterns

In this section, we introduce the common overhead patterns in workflow execution. A overhead pattern is We first use a real trace of workflow execution to illustrate the common patterns in workflow overheads. Fig. 4 shows a real trace captured from an existing execution of a workflow[1].

### A. Adhere Pattern

In this pattern, an overhead is attached to the beginning or the end of a task/job. The duration of this overhead is

[1]Details: http://www.isi.edu/ wchen/fgrid/run

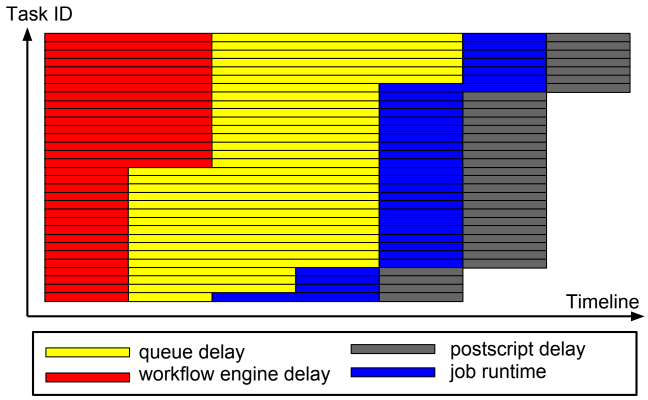


Fig. 4: Workflow Execution Gantt Chart

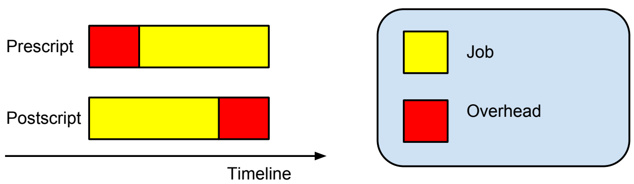


Fig. 5: Adherence Pattern

independently distributed. For example, the prescript and the postscript delay. For this pattern, we have shown in our WorkflowSim paper that it does not have much influence on the overhead robustness. Repeat what has been shown in WorkflowSim and make it more detailed.

### *B. Incremental and Cyclic Pattern*

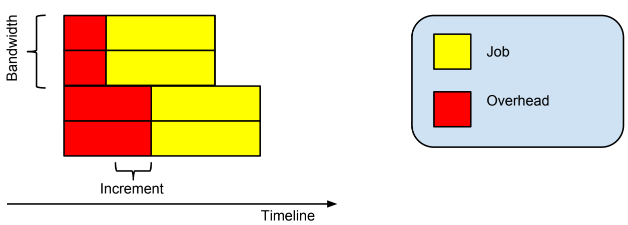


Fig. 6: Incremental and Cyclic Pattern

In this pattern, the overheads in each cycle remain the same but in different cycles they may increase. For example, workflow engine only releases 5 jobs at a cycle and thus the workflow engine delay for the jobs in the same cycle is same but it will increase with more and more cycles. We define the maximum number of jobs a component may release as 'Bandwidth'. For the figure below, its bandwidth is 2.

## V. Overhead Friendly Heuristics

Heuristics

MINMIN: The Min-min heuristic begins with the set U of all unmapped tasks. Then, the set of minimum completion times, M=[minj¡+(ct(ti, mj)), for each t, is found. Next, the task with the overall minimum completion time from M is selected and assigned to the corresponding machine (hence the name Minin). Last, the newly mapped task is removed from U, and the process repeats until all tasks are mapped (i.e., U is empty) [3, 17, 23]. Minin is based on the minimum completion time, as is MCT. However, Minin considers all unmapped tasks during each mapping decision and MCT only considers one task at a time. Minin maps the tasks in the order that changes the machine availability status by the least amount that any assignment could. Let ti be the first task mapped by Minin onto an empty system. The machine that finishes ti the earliest, say mj , is also the machine that executes ti the fastest. For every task that MINMIN maps after ti, the Minin heuristic changes the availability status of mj by the least possible amount for every assignment. Therefore, the percentage of tasks assigned to their first choice (on the basis of execution time) is likely to be higher for Minmin than for Maxin (defined next). The expectation is that a smaller makespan can be obtained if more tasks are assigned to the machines that complete them the earliest and also execute them the fastest.

Maxin: The Maxmin heuristic is very similar to Minin. The Maxin heuristic also begins with the set U of all unmapped tasks. Then, the set of mini- mum completion times, M, is found. Next, the task with the overall maximum com- pletion time from M is selected and assigned to the corresponding machine (hence the name Maxmin). Last, the newly mapped task is removed from U, and the process repeats until all tasks are mapped (i.e., U is empty)

We define the **Impact Factor Variance** ($IFV$) of tasks as the standard deviation of their impact factor. The intuition of Impact Factor is that we aim to capture the similarity of tasks/jobs in a graph by measuring their relative impact factor or importance to the entire graph. Intuitively speaking, tasks with similar impact factors should be merged together compared to tasks with different impact factors. Also, if all the tasks have similar impact factors, the workflow structure tends to be more 'even' or 'regular'. The **Impact Factor** ($IF$) of a task $t_u$ is defined as follows:

$$IF(j_u) = \sum_{j_v \in Child(j_u)} \frac{IF(j_v)}{L(j_v)} \tag{1}$$

where $Child(j_u)$ denotes the set of child jobs of $j_u$, and $L(j_v)$ the number of parent jobs of $j_v$. For simplicity, we assume the $IF$ of a workflow exit job (e.g. $j_5$ in Fig. **??**) as 1.0. For instance, consider the two workflows presented in Fig. **??**. $IF$ for $j_1$, $j_2$, $j_3$, and $j_4$ are computed as follows:

$$\begin{aligned} IF(j_7) = 1.0, IF(j_6) = IF(j_5) = IF(j_7)/2 = 0.5 \\ IF(j_1) = IF(j_2) = IF(j_5)/2 = 0.25 \\ IF(j_3) = IF(j_4) = IF(j_6)/2 = 0.25 \end{aligned}$$

Thus, IFV($j_1$, $j_2$, $j_3$, $j_4$) = 0. In contrast, $IF$ for $j'_1$, $j'_2$, $j'_3$, and $j'_4$ are:

$$IF(j_7') = 1.0, IF(j_6') = IF(j_5') = IF(j_1') = IF(j_7')/2 = 0.5$$
$$IF(j_2') = IF(j_3') = IF(j_4') = IF(j_6')/3 = 0.17$$

The idea of these overhead aware heuristics is we would like to improve the overhead robustness even when we are not able to precisely predict the duration of overheads. Instead, workflow overheads have common patterns, which can determine the relative performance of different heuristics.

### A. *Breadth First vs. Depth first*

### B. *MAXMIN vs. MINMIN*

### C. *Fertile Task First vs. Unfertile Task First*

### D. *Important Task First vs. Unimportant Task First*

## VI. Experiment and Evaluation

The experiments presented hereafter evaluate the performance of the four pairs of heuristics mentioned ahead, which are widely used by workflow management systems.

### A. *Experiment Conditions*

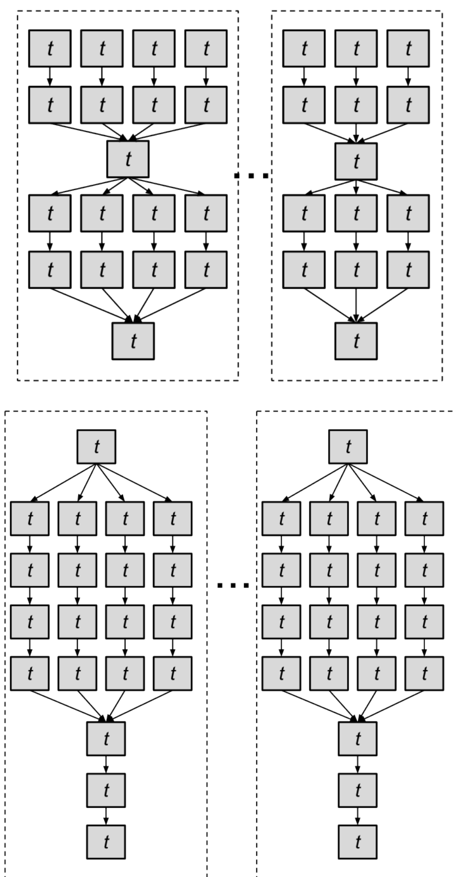

Fig. 7: A simplified visualization of the LIGO Inspiral workflow (top) and Epigenomics workflow (bottom).

We extended the WorkflowSim [17] simulator with the overhead model to simulate a distributed environment where we could evaluate the overhead robustness of scheduling algorithms when varying the average overheads and bandwidth. As an initial attempt, we focus on the workflow engine delay ($d$) in this paper. The simulated computing platform is composed by 20 single core virtual machines (worker nodes), which is the quota per user of some typical distributed environments such as Amazon EC2 [18] and FutureGrid [19]. Each machine has 512MB of memory and the capacity to process 1,000 million instructions per second.

| Heuristics | Overhead Friendly | Overhead Unfriendly |
|---|---|---|
| Experiment 1 | MAXMIN | MINMIN |
| Experiment 2 | BFS | DFS |
| Experiment 3 | FFS | UFFS |
| Experiment 4 | IFS | UIFS |

Three workflows are used in the experiments: Broadband [**?**] is an application that enables researchers to combine long-period deterministic seismograms with high-frequency stochastic seismograms. Montage [] is an astronomy application used to construct large image mosaics of the sky. CyberShake [] is a seismology application that calculates Probabilistic Seismic Hazard curves for geographic sites in the Southern California region. All workflows are generated and varied using the WorkflowGenerator[1]. Each workflow instance is composed by around 100 tasks and its workflow structure is presented in Fig. 7. Runtime (average and task runtime distribution) and overhead (workflow engine delay, queue delay, and network bandwidth) information were collected from real traces production environments [1], [13], then used as input parameters for the simulations.

Four sets of experiments are conducted. Experiment 1 evaluates the relative robustness of MAXMIN and MINMIN by comparing the performance gain of the overhead friendly heuristic (MAXMIN in this experiment) over the overhead unfriendly heuristic (MINMIN), while varying the average workflow engine delay ($d$) and bandwidth of workflow engine ($b$). Table VI-A shows the heuristics compared in these experiments. Simulation results present a confidence level of 95%. Thus, for values of *Performance Gain* $> 0$, the overhead friendly heuristics perform better than the respective overhead unfriendly heuristics. Otherwise, the overhead friendly heuristics perform poorer.

In these experiment sets, we vary the average bandwidth of workflow engine from 1 to 20 and we show the results of $b = 1, 5, 15, 20$. The original bandwidth of workflow engine in these traces is 5 and a bandwidth that is larger than 20 does not influence the performance much since we have only 20 worker nodes in our experiments. We also vary the average workflow engine delay from 0 to 100 seconds, which represents a typical range of workflow engine delay as shown in [1] and this range is able to show the difference of overhead robustness in these heuristics.

[1] https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator

### B. Results and Discussion

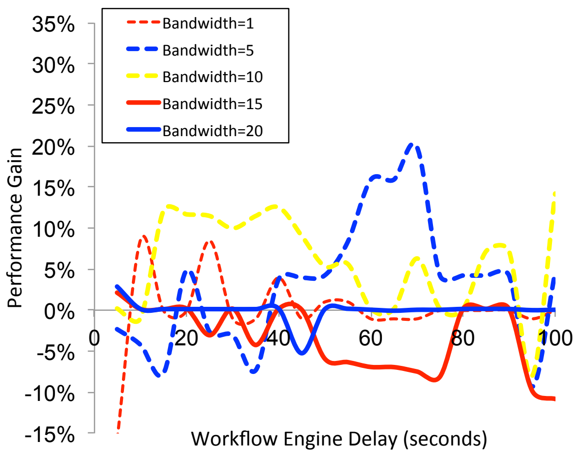


Fig. 8: Broadband

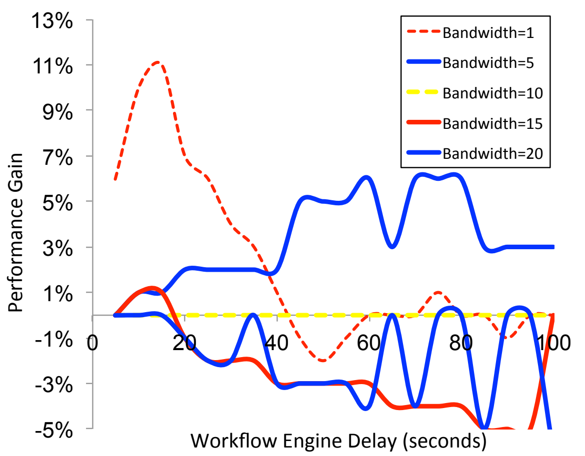


Fig. 9: CyberShake

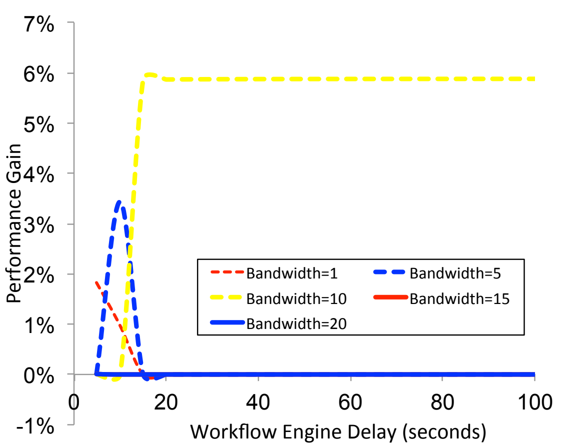


Fig. 10: Montage

Experiment 1: Fig. 8, 9, 10 show the *Performance Gain* of MAXMIN over MINMIN for the three workflows. We expected to see most *Performance Gain* $> 0$ if MAXMIN is a overhead friendly heuristic compared to MINMIN. However, except for the Montage workflow, the *Performance Gain* is not significant for all of parameter settings, which concludes that MAXMIN is not globally overwhelming MINMIN in terms of overhead robustness. The reason we believe is that the *Performance Gain* in this comparison highly depends on the ratio of task runtime, workflow engine delay, number of resources and bandwidth.

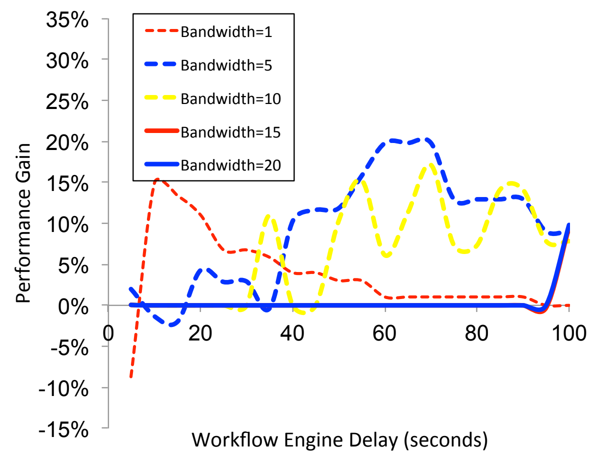


Fig. 11: Broadband

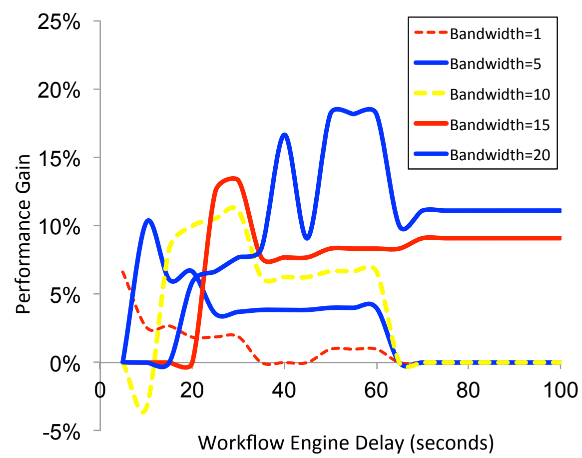


Fig. 12: CyberShake

Experiment 2: Fig. 11, 12, 13 shows the *Performance Gain* of BFS over DFS for the three workflows. We observe that most *Performance Gain* $> 0$ and thus BFS performs better than DFS in terms of overhead robustness. What is more, Fig 12 shows with the increase of average bandwidth, the *Performance Gain* is more significant. We can also see that when the average bandwidth is high, the *Performance Gain* increases with the average workflow engine delay. This suggests us in a real environment with a large overhead, we

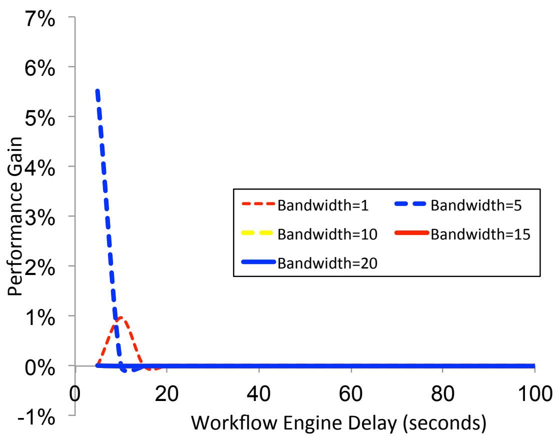


Fig. 13: Montage

should use BFS instead of DFS.

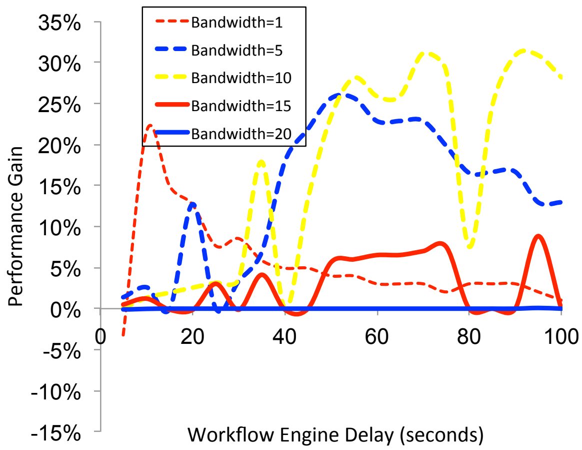


Fig. 14: Broadband

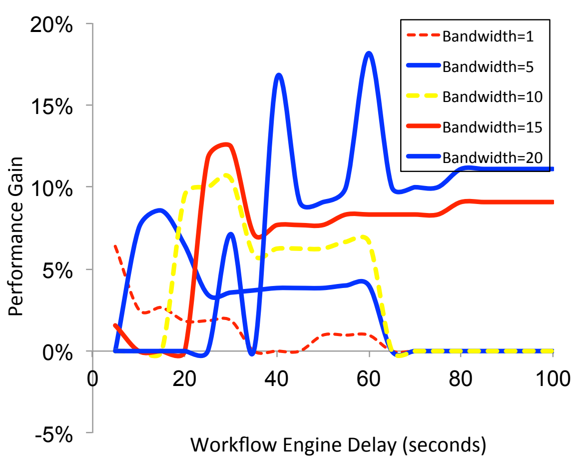


Fig. 15: CyberShake

Experiment 3: Fig. 14, 15, 16 shows the *Performance Gain*

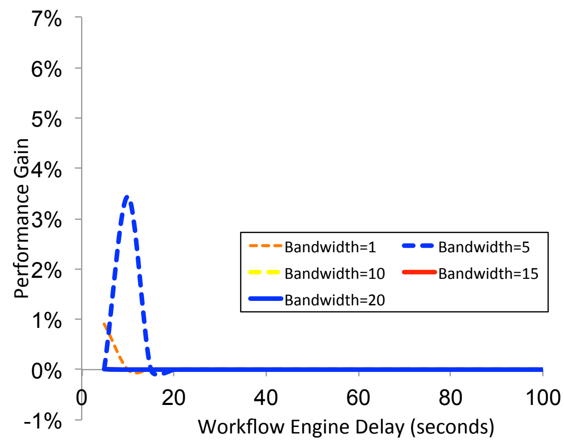


Fig. 16: Montage

of FFS over UFFS for the three workflows. We observe that most *Performance Gain* $> 0$ and thus FFS performs better than UFFS in terms of overhead robustness, which is similar to Experiment 2. Comparing Fig. 11 and Fig. 14 we can see the *Performance Gain* of FFS over UFFS is more significant (30%) than that of BFS over DFS (20%).

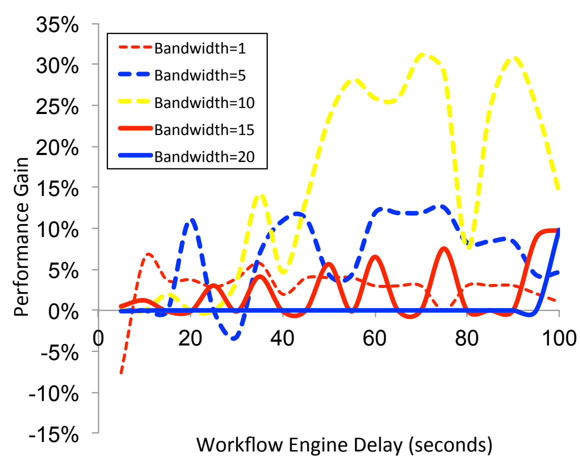


Fig. 17: Broadband

Experiment 4: Fig. 17, 18, 19 shows the *Performance Gain* of IFS over UIFS for the three workflows. We observe that most *Performance Gain* $> 0$ and thus IFS performs better than UIFS in terms of overhead robustness, which is similar to Experiment 3. The reason is that for these workflows, IF based heuristics can produce similar schedule as the heuristics based on the number of children. Most of the workflows used in this paper is not irregular enough and thus we are not able to show the difference of IF based heuristics and the heuristics based on the number of children.

## VII. Conclusion

## References

[1] W. Chen and E. Deelman, "Workflow overhead analysis and optimizations," in *Proceedings of the 6th workshop on Workflows in support of*

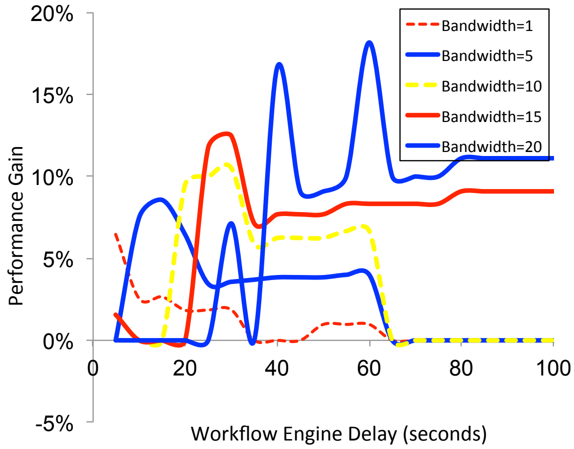


Fig. 18: CyberShake

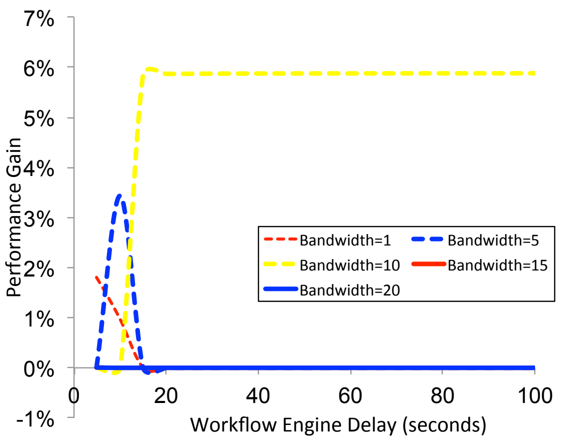


Fig. 19: Montage

*large-scale science*, ser. WORKS '11. New York, NY, USA: ACM, 2011, pp. 11–20.

[2] F. Dong, J. Luo, A. Song, and J. Jin, "Resource load based stochastic dags scheduling mechanism for grid environment," in *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, 2010, pp. 197–204.

[3] L. Yang, J. M. Schopf, and I. Foster, "Conservative scheduling: using predicted variance to improve scheduling decisions in dynamic environments," 2003, pp. 1–16.

[4] I. Ahmad, M. K. Dhodhi, and R. Ul-Mustafa, "Dps: Dynamic priority scheduling heuristic for heterogeneous computing systems," *Computers and Digital Techniques, IEE Proceedings-*, vol. 145, no. 6.

[5] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real-Time Systems 2.3*, 1990.

[6] R. Sakellariou and H. Zhao, "A low-cost rescheduling policy for efficient mapping of workflows on grid systems," *Sci. Program.*, vol. 12, no. 4, pp. 253–262, Dec. 2004.

[7] Y. Zhang, C. Koelbel, and K. Cooper, "Hybrid re-scheduling mechanisms for workflow applications on multi-cluster grid," in *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, 2009, pp. 116–123.

[8] S. Ali, A. Maciejewski, H. Siegel, and J.-K. Kim, "Measuring the robustness of a resource allocation," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 15, no. 7, pp. 630–641, 2004.

[9] L.-C. Canon and E. Jeannot, "Comparative evaluation of the robustness of dag scheduling heuristics," in *Grid Computing*. Springer, 2008.

[10] T. D. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. A. Hensgen, and R. F. Freund, "A comparison of eleven static heuristic for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, 2001.

[11] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, and et al., "Task scheduling strategies for workflow-based applications in grids," in *IN CCGRID*. IEEE Press, 2005, pp. 759–767.

[12] J. Yu and R. Buyya, "A taxonomy of workflow management systems for grid computing," *Journal of Grid Computing*, vol. 3.

[13] 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.

[14] E. Deelman, G. Singh, M.-H. 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," *Sci. Program.*, vol. 13, no. 3, pp. 219–237, Jul. 2005.

[15] 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, Feb. 2005.

[16] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe, "Taverna: lessons in creating a workflow environment for the life sciences: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 18, no. 10, pp. 1067–1100, Aug. 2006.

[17] W. Chen and E. Deelman, "Workflowsim: A toolkit for simulating scientific workflows in distributed environments," in *The 8th IEEE International Conference on eScience*, Oct. 2012.

[18] Amazon.com, Inc., "Amazon Web Services," http://aws.amazon.com. [Online]. Available: http://aws.amazon.com

[19] "FutureGrid," http://futuregrid.org/. [Online]. Available: http://futuregrid.org/