Scheduling Workloads of Workflows with Unknown Task Runtimes

Ilyushkin, Alexey; Ghit, Bogdan; Epema, Dick

DOI
10.1109/CCGrid.2015.27

Publication date
2015

Document Version
Accepted author manuscript

Published in
15th IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing

Scheduling Workloads of Workflows with Unknown Task Runtimes

Alexey Ilyushkin
Delft University of Technology
Delft, the Netherlands
a.s.ilyushkin@tudelft.nl

Bogdan Ghiț
Delft University of Technology
Delft, the Netherlands
b.i.ghit@tudelft.nl

Dick Epema
Delft University of Technology
Delft, the Netherlands
d.h.j.epema@tudelft.nl

***Abstract*—Workflows are important computational tools in many branches of science, and because of the dependencies among their tasks and their widely different characteristics, scheduling them is a difficult problem. Most research on scheduling workflows has focused on the offline problem of minimizing the makespan of single workflows with known task runtimes. The problem of scheduling multiple workflows has been addressed either in an offline fashion, or still with the assumption of known task runtimes. In this paper, we study the problem of scheduling workloads consisting of an arrival stream of workflows without task runtime estimates. The resource requirements of a workflow can significantly fluctuate during its execution. Thus, we present four scheduling policies for workloads of workflows with as their main feature the extent to which they reserve processors to workflows to deal with these fluctuations. We perform simulations with realistic synthetic workloads and we show that any form of processor reservation only decreases the overall system performance and that a greedy backfilling-like policy performs best.**

I. Introduction

In many branches of science such as astronomy and bioinformatics, workflows (WFs) are widely used for all kinds of computational and data analysis problems. Because of the dependencies among their tasks and because of the diversity of their structures, sizes, and task runtimes, scheduling WFs efficiently on clusters and datacenters is a difficult problem. Most research on scheduling WFs focuses on the offline problem of minimizing the makespan of single WFs for which estimates of the task runtimes are known. In contrast, in this paper we propose four scheduling policies for online scheduling workloads consisting of arriving WFs with unknown task runtimes, and we perform simulations to evaluate their performance.

The problem of minimizing the makespan of single WFs has been studied very extensively, usually assuming that the task runtimes are known. Well-known approaches for task selection when resources become available are the HEFT policy [1] that schedules each WF task on the processor that minimizes its finish time, and policies that try to optimize the schedule of the critical path of a WF [2]–[4]. Scheduling multiple WFs has received some attention in the past. However, some of this work still considers the offline problem of minimizing the total makespan of a fixed set of WFs, without or with the added requirement of fairness [5]. Other work does consider the online problem with an arrival stream of WFs but translates the problem into scheduling and executing complete batches of WFs before considering later arrivals [6], or builds a single Directed Acyclic Graph (DAG) from the DAGs representing a set of WFs [7]. In this paper we study the online problem of scheduling an arrival stream of WFs, which in addition to the problem of task selection involves the problem of WF selection from which to pick tasks to run.

The WFs that are used in practice are still growing in size, complexity, as well as in the number of dependencies among their tasks [8]–[10]. Even though WFs are popular as an automation tool for e-Science experiments [11], obviously, the workloads of real clusters consist of jobs of different types in addition to WFs, such as parallel applications and Bags of Tasks. When WFs are run very often, either the user or the system may be able to derive reasonable estimates of the runtimes of tasks of WFs. However, in this paper, we take a step back and we address the fundamental question of how well a workload consisting exclusively of WFs of which the task runtimes are not known can be scheduled.

When scheduling WFs from the queue of WFs that have been submitted but that have not yet completed, resources may be available while the WFs towards the head of the queue may not have tasks that are eligible to run. Thus, the main distinguishing feature of the four scheduling policies we propose is to what extent they are greedy in scheduling any task of any WF in the queue versus to what extent they reserve processors for WFs towards the head of the queue in order not to unduly delay these WFs. Our policies range from a very strict reservation-based policy that guarantees no delay to the WF at the head of the queue due to later WFs, to a greedy backfilling policy. The jobs for the simulations we generate are based on real scientific workloads [8]–[11]. In our performance evaluation we report the average job slowdown of the WFs and the maximal utilization—because WF scheduling is not work-conserving due to the precedence constraints among WF tasks, the system may become saturated for utilizations well below 1.0.

For the implementation of the system model and the proposed scheduling policies we use an improved version of the DGSim discrete event simulator [12].

This paper is organized as follows. In Section II we provide our problem statement. Section III motivates and presents our scheduling policies. Section IV discusses our experimental setup and characterizes the synthetic workloads. In Section V we show and explain the obtained results. Section VI contains a survey of related work. Finally, in Section VII we present our conclusions and ideas for future work.

II. The Problem Statement

We consider large-scale computing systems such as large clusters and datacenters that are subject to an arrival stream of WFs. We only consider processors as the type of system resources that can be controlled by the scheduler. Every computing node in the system contains only one processor. Furthermore, we assume that the system is homogeneous, with identical processors and communication links between them. Since we focus on the computational properties of WFs, we assume that the data transfer times between computing nodes in our simulated system can be neglected. This is equivalent to the situation in a real system when all the tasks of a WF write their results to a shared storage so that all the required data are immediately available for any WF task when all of its dependencies are satisfied.

In this paper we assume that each WF task requires only one processor. The *size* of a WF is defined as the number of tasks it has. All the considered WF structures have a single entry node and a single exit node. We guarantee this by adding, if necessary, one or two artificial nodes with zero runtime.

Many WF scheduling algorithms employ user estimates or predictions of task runtimes. It is well known that the estimates provided by users are usually quite inaccurate [13]. At the same time, the runtime prediction approaches are often relatively complex, do not work well for some situations, and can also be inaccurate. Finally, always new or unknown WFs can be submitted to systems. For all of these reasons, we suppose that the runtimes of the tasks of the WFs are unknown to the scheduler.

Scheduling WFs is in itself not work-conserving as there may be idle processors in the system while there is no waiting task with all its dependencies satisfied. In addition, a property of several of our policies is that they reserve processors to WFs in order to deal with their fluctuating resource requirements. As a consequence, policies scheduling workloads of WFs may not be able to drive a system up to a utilization of 1.0. Therefore, we use the maximal utilization as a system-oriented metric to assess the performance of WF scheduling policies. The *maximal utilization* ρ_m is defined as the utilization such that for any ρ_1 with $\rho_1 < \rho_m$ the system is stable (not saturated), and for any ρ_2 with $\rho_2 > \rho_m$ the system is unstable (saturated).

As a user-oriented metric to assess WF scheduling policies we use the (average) slowdown, which is defined in steps in the following way:

- The *wait time* T_w of a WF is the time between its arrival and the start of its first task.
- The *execution time* T_e of a WF is the sum of the runtimes of all its tasks.
- The *makespan* T_m of a WF is the time between the start of its first task until the completion of its last task.
- The *response time* T_r of a WF is the sum of its wait time and its makespan: $T_r = T_w + T_m$.
- The *slowdown* S of a WF is its response time (in a busy system, when the WF runs simultaneously with other WFs) normalized by its makespan T'_m in an empty system of the same size (when the WF has exclusive access to all the processors): $S = T_r \,/\, T'_m$.

The research question we address in this paper is "What are appropriate policies for online scheduling WFs without having knowledge of task runtimes, and what is their performance in terms of the job slowdown as a function of the system utilization, and of the maximal utilization?"

III. Scheduling Policies

In this section we will describe the four policies for scheduling workloads consisting of WFs the simulation results of which we will show later in the paper. Before doing so, we will present some definitions and concepts required for explaining the policies, and the way the scheduler manages the queue of waiting WFs.

A. Definitions

For a WF, at any point in time before or during its execution, its *eligible set* (of tasks) is defined as the set of non-completed tasks of which the precedence constraints have been satisfied. In other words, the eligible set is the set of all tasks that are currently running and those that are waiting but that could run if sufficient resources were available. More generally, we introduce the notion of the *generation-i eligible set* (or the *eligible set of generation i*) at any point in the execution of a WF, which is a potential future eligible set. The *generation-0 eligible set* of a WF is simply equal to its current eligible set. The *generation-$(i+1)$ eligible set* of a WF contains all the tasks that will become eligible when (exactly and only) all the tasks from its *generation-i eligible set* have completed. It is important not to confuse eligible sets with levels in a WF. The *levels* consist of the tasks that have the same distance from the entry task. However, the eligible sets of *generation-i* can differ from the levels, because paths of different lengths can exist in a WF from the entry task to any other single task.

For a WF that has not yet completed, we define its *Level of Parallelism* (LoP) as the maximum number of processors it may ever use at any future point in its execution, which is equal to the maximum number of tasks in any of its potential

future eligible sets. Of course, the LoP of a WF can only stay the same or decrease during its execution. The LoP is used in the definition of some of our scheduling policies.

B. Calculating the Level of Parallelism

In some of our scheduling policies, we will use the LoP of the remaining part of a WF consisting of the tasks that have not yet completed for deciding how many processors to reserve for it. Then, whenever a task of a WF completes, the LoP of the remaining part of the WF has to be recomputed, leading to a very large number of LoP recomputations. The DAG representing the non-completed part of a WF is a sub-DAG of the DAG of the original WF. Such a sub-DAG can have multiple entry nodes but will always have a single exit node. We say that a node a precedes (follows) node b in a DAG when there is a path of precedence constraints from a (b) to b (a). Two nodes a and b are said to be comparable (incomparable) if one (none) of them precedes or follows the other. With the "comparable" relation, a DAG can be considered as a *partially ordered set* (*poset*). The value of the LoP is equal to the width of this poset, which is defined as the cardinality of the maximum set of incomparable elements in the poset. There exist multiple algorithms [14] to compute the exact LoP using Dilworth's chain partition of the original DAG, but they require the creation of an additional comparability graph or a bipartite graph. Dilworth's theorem [15] represents the width of a poset as a partition of the poset into a minimum number of *chains*, where each *chain* is a path from a source to a sink in the directed comparability graph.

To avoid the construction of any auxiliary data structures, we will use a simple LoP approximation algorithm that calculates the LoP in an adequate time even for large WF structures, and, as we will show, it does so with only a relatively small amount of underestimation of the actual LoP for many well-known WF structures. Our approximation algorithm uses the size of the largest generation-i eligible set as the value of the LoP. To find the size of this set the algorithm employs tokens to simulate an execution "wave" in a DAG. Initially, the algorithm places tokens in the entry nodes of the DAG. Then it moves in successive steps the tokens to all the nodes all of whose parents have already received tokens, until the exit node gets a token (Figure 1a). After each such step, the set of tokenized nodes is recorded. At the end of the algorithm, the size of the largest recorded tokenized set is the approximated LoP value. Note that in fact, these tokenized sets coincide with the eligible sets of different generations as defined in Section III-A.

We have compared the results provided by this approximation algorithm with the exact LoPs for different WF sizes for five popular WF structures, LIGO, SIPHT, Montage, Cybershake, and Epigenomics. For each considered size of each WF type, we generate 50 DAGs using the generator from [16]. As can be seen from Figure 2, our method approximates the true LoP value extremely well. We provide the LoPs here only for LIGO, SIPHT, and Montage, since Cybershake and Epigenomics show similar results. For both methods, the LoP values obtained for the 50 DAGs of each WF type are all very close to the mean. Of course, there can be situations where our approach does underestimate the LoP, see for instance the simple example in Figure 1. However, since the algorithm works well for the selected WFs, which are quite popular and representative, we will use the LoPs computed by it when simulating policies that use LoP in their scheduling decisions.

From the results in Figure 2, we can conclude that the tested WFs have rather regular, but still different, structures. First of all, for all three WF structures, the (average) LoP increases superlinearly (and especially for Montage and SIPHT, in a very strong way). Secondly, even for LIGO, but especially for the other two, the LoP is very large in relation to their total size. For LIGO, the LoP is slightly less than 200 for a WF size of 800, but Montage and SIPHT already reach that LoP for WFs of size less than 400.

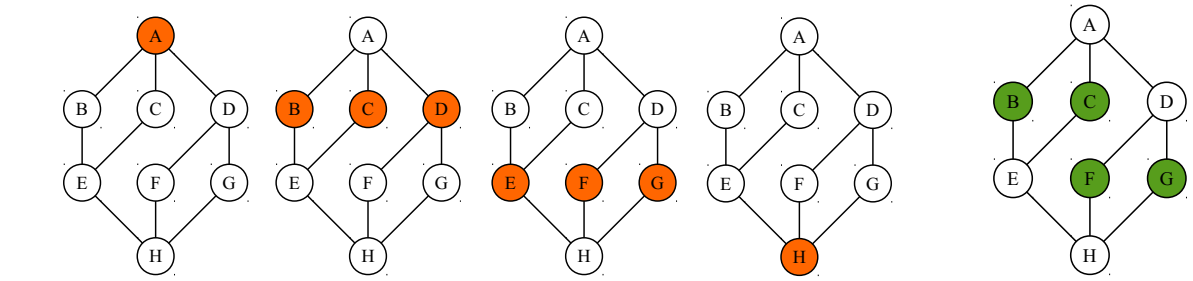
C. Queue management and task selection

We assume that the scheduler maintains a single queue of waiting WFs. Every arriving WF is appended to the tail of the queue, and the scheduler decides which tasks of which WFs in the queue are scheduled when resources become available. For all the policies we will consider, the scheduler is invoked when a task of a WF completes, or when a new WF arrives (possibly to a non-empty queue). The WFs are in principle processed in the order of their arrival, but multiple WFs can be partially in execution while the remainder of their tasks are still waiting (either for a lack of resources or because of precedence constraints between tasks), in which case we still regard them to be present in the queue. A WF only leaves the queue when all of its tasks are finished.

When the scheduler is activated, it selects the WFs in the queue from which tasks are scheduled in a way dictated by the actual scheduling policy. As we assume that all WF tasks require only one processor and that we do not have user estimates or predictions of task runtimes, after the scheduler has selected a WF from which to start a task, it picks a task from its eligible set randomly. As the execution order of the tasks from the current eligible set may influence subsequent eligible sets, and so, the makespan of the WF, this random task selection may not lead to the optimal schedule for the WF, but without knowledge of task runtimes it is difficult to do better. Giving priority to tasks in the eligible set that would enable large numbers of other tasks might improve the makespan of a WF, but this is highly dependent on the task runtimes.

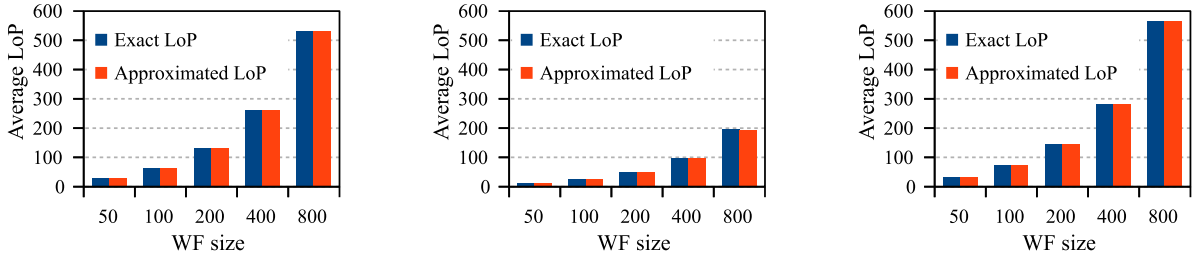
D. The Strict Reservation policy

The most classic and simple general queuing policy is FCFS. When scheduling WFs, the definition of FCFS is not



(a) Execution "waves" (the orange vertices in each graph constitute the sets of tokenized nodes) (b) Exact LoP

Figure 1: An example of LoP approximation (a) versus the exact LoP (b).



(a) Montage (b) LIGO (c) SIPHT

Figure 2: A comparison of the exact and the approximation method for calculating the LoP for different WF structures.

completely straightforward. The idea behind our version of the FCFS policy for WFs can be summarized by the condition that the service to a WF will only be influenced by the WFs ahead of it in the queue and never by WFs behind it in the queue. We enforce this condition by strictly reserving for any WF in the queue sufficient resources so that it will never be delayed by any later WF—hence the alternative name is Strict Reservation (SR) policy.

With the SR policy, when the scheduler is invoked and the WF at the head of the queue has fewer processors allocated to it than its LoP (which may be in use or idle), it allocates additional available processors to the WF at the head of the queue until the WF has LoP processors or until there are no more available processors. When at a later time the scheduler is invoked again while the WF at the head of the queue has not yet been completed, the scheduler recomputes its LoP, keeps LoP processors allocated to the WF, and releases any remaining processors. If any idle processors remain after LoP processors have been allocated to the WF, the scheduler tries to schedule the next WF in the queue—it does so in exactly the same way as if that WF were at the head of the queue.

E. The Scaled LoP policy

Of course, WFs may never attain their LoPs, and especially for large WFs, the SR policy may be very wasteful and lead to a low maximal utilization. Whereas the SR policy executes a WF in the shortest possible time once it starts allocating processors to it, the wait times of the WFs with SR may be excessive. A straightforward solution to this problem is to reduce the reservation of a WF to a number of processors that is lower than its LoP. Thus, the Scaled LoP (SLoP) policy with *scaling factor* $f, 0 \leq f \leq 1$, tries at all times to keep $f \cdot$ LoP processors allocated to a WF. This means that if at some point the size of the eligible set of a WF is smaller than $f \cdot$ LoP, the scheduler will try to keep reserved a number of processors equal to the difference between these two values. If, however, at some point the size of the eligible set of a WF exceeds $f \cdot$ LoP, the SLoP policy will allocate any available processors to eligible tasks of WF. The SLoP policy behaves similar to the SR policy albeit with a lower reservation target, and in the boundary case when $f = 1$ it is equal to it. At the other extreme, as we will see below, our backfilling policy is in fact identical to the SLoP policy with scaling factor equal to 0.

F. The Future Eligible Sets policy

The idea behind reserving processors for WFs is to reduce the delay in placing tasks in the eligible set. Rather than, when reserving processors, taking a worst-case perspective on the number of processors a WF may ever need as we did in the SR policy, we may also try to look into the future of the execution of the WFs. Thus, the Future Eligible Sets (FES) policy with *depth* n tries to allocate to a WF a number of processors that is equal to the size of largest eligible set of any generation from 0 through n. With our approximation of computing the LoP as presented in Section III-B, the FES policy with depth ∞ coincides with the SR policy. At the

other extreme, as we will see below, our backfilling policy is in fact identical to the FES policy with depth equal to 0.

G. The Backfilling policy

Depending on the scaling factor and the depth, the SLoP and FES policies may be very wasteful of resources because of useless reservations. To completely do away with reservations and the waste of resources it entails, we now define the Backfilling (BF) policy that tries to allocate to any WF any number of processors up to the size of its current eligible set. Thus, at every invocation, the scheduler scans the queue from head to tail and from each WF it encounters it places as many tasks from the corresponding eligible set as it can. Thus, the BF policy is greedy as it allows to schedule any task from any eligible set of any WF in the queue. As we assume that each task requires only one processor, there will only be idle processors in the system when all tasks in the eligible sets of all WFs in the queue are actually running. As already remarked above, the BF policy is a special case of the SLoP policy with scaling factor $f = 0$.

Our BF policy for WFs is similar to the backfilling policies that have been introduced for parallel jobs [13], [17], but our policy does not use runtime estimates. However, even so, with our BF policy for WFs and because of our assumption that each task requires one processor, starvation is intrinsically impossible—we always try to schedule as many tasks of a WF as possible before considering the next WF, thus always granting at least some resources to a WF before allowing WFs later in the queue to receive resources. In contrast, unless special measures are taken, starvation is possible when backfilling parallel jobs (and when backfilling WFs with parallel tasks).

Compared to the SR policy, with BF, on the one hand WFs can be delayed by later WFs, thus increasing the makespans and so the job slowdowns. On the other hand, BF allows the WFs in the queue to start their execution earlier, thus decreasing the wait times and so the job slowdowns. From our evaluation we will see which of these two effects is stronger.

IV. Experimental Setup

In this section, we present our simulation environment and we characterise the synthetic workloads we use to analyse the performance of our scheduling policies.

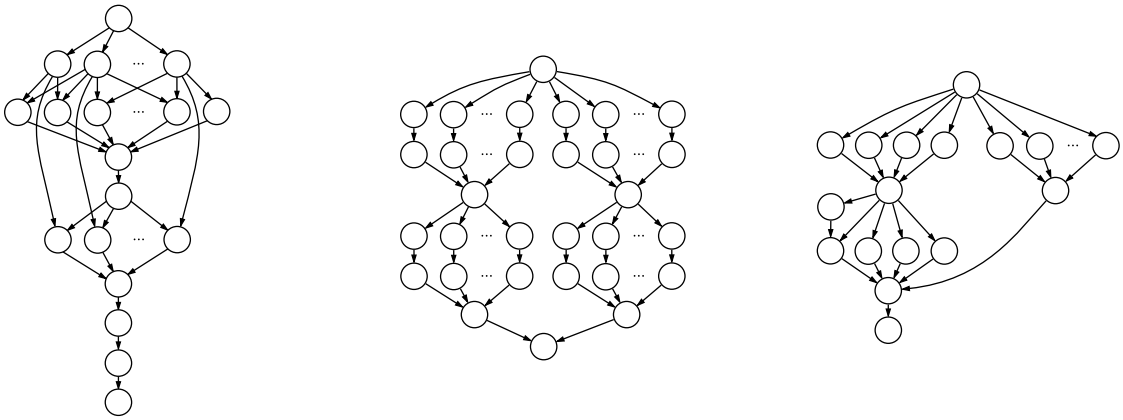
We have modified the DGSim simulator [12], [18] for cluster and grid systems to include our scheduling policies. The only resource modeled in our simulations is the processors. We assume that the WFs submitted to the simulated cluster arrive according to a Poisson process. The size of the homogeneous cluster we use in all of our simulations is 100. For our simulations, we select three representative types of WFs from different application domains, i.e., astronomy (Montage [8]–[10], [19]), physics (LIGO [8]–[10], [20]), and bioinformatics (SIPHT [8], [9], [21]). Montage builds mosaic images of the sky obtained from different telescopes. LIGO is used to process the data from detectors of the Laser Interferometer Gravitational Wave Observatory (LIGO) [22] and its mission is to detect gravitational waves predicted by general relativity. SIPHT helps to search for small untranslated bacterial regulatory RNAs.

In Figure 3 we show the structure of the DAGs of the three WF types. Montage has the most complicated structure and its size is determined by the number of processed images. A LIGO WF usually consists of many smaller WFs combined into a single WF. Similarly to LIGO, the SIPHT WF combines smaller independent WFs, but with very similar structures. The WF types are diverse not only in the structure of their DAGs, but also with respect to the processing requirements of the component tasks as we will see below. Furthermore, we have already analysed the maximum levels of parallelism they can achieve in Section III-B.

We generate four workloads of 3,000 jobs each using the WF generator [16] presented in [8]: one workload per WF type, and an additional workload that mixes equal fractions of the three types. As with many other workloads in computer systems, in practice, WFs are usually small, but very large ones may exist too [11]. Therefore, in our simulations we distinguish small, medium, and large WFs, which constitute fractions of 75%, 20%, and 5% of the workloads. We assume all WFs to have even sizes. The size of the small, the medium, and the large WFs is uniformly distributed on the intervals [30, 38], [40, 198], and [200, 600], respectively.

In order to obtain simulation results for the different WF types that can easily be compared (especially when simulating the mixed workload), we use the same total execution time distribution for all three WF types. This distribution is a two-stage hyper-Gamma distribution derived from the model presented in [23]. The shape and scale parameters (α, β) of two component Gamma distributions are set to (5.0, 501.266) and (45.0, 136.709), respectively. Their proportions in the overall distribution are 0.7 and 0.3. The average total execution time is one hour. Figure 4 visualizes this distribution. For every WF, we normalize the task runtimes generated so that its total processing requirement is equal to the corresponding sample of the execution time distribution. In Figure 5 we show the distributions of the normalized task runtimes for each WF type. The maximum task runtimes in Montage and LIGO are an order of magnitude smaller than the maximum task runtimes in SIPHT, but all three WF types share the phenomenon that they are dominated by short tasks.

In our simulations, we vary the utilization starting at 0.05 with step size 0.05. If for some utilization the system did not become stable in the simulations, we will only show performance results up to that utilization. We will consider the highest utilization for which the system did become stable as the maximal utilization as defined in Section II. We set the scaling factor f in our SLoP policy to 0.2, 0.8, and 0.9,



(a) Montage (b) LIGO (c) SIPHT

Figure 3: The WF structures we use in our simulations.

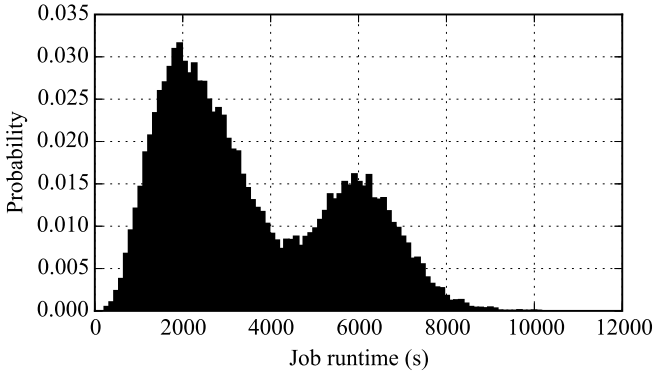


Figure 4: The distribution of the total WF execution times in the workloads.

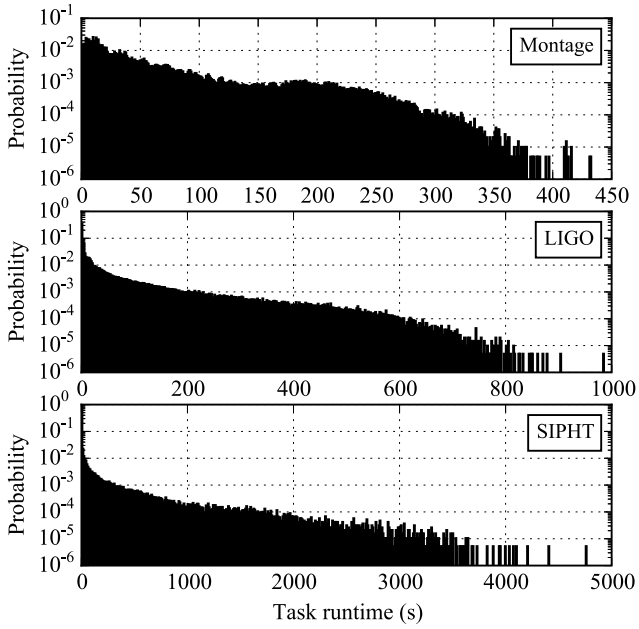


Figure 5: The distribution of the task runtimes for each WF type (the horizontal axes have different scales, and the vertical axes are in log scale).

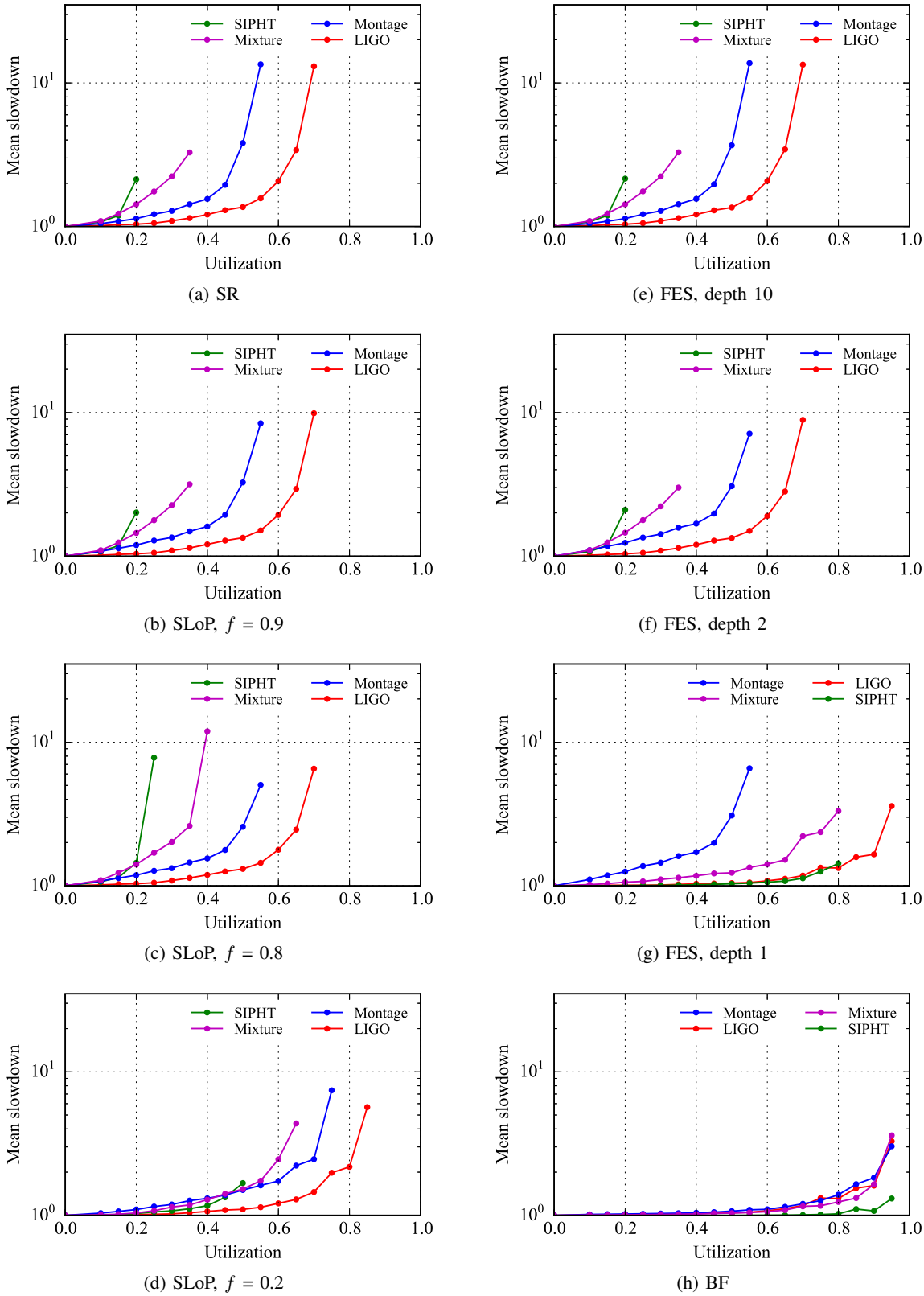
and we evaluate the FES policy with depths 1, 2, and 10. For each experiment, we report the average job slowdown over three repetitions, and we only show results when the system is in steady state. Thus, when reporting performance results, we omit the performance information for the first 1,000 jobs in each simulation and for those WFs that have not completed their execution before the start of the last arriving WF.

V. Experimental Results

In this section we report our simulation results and their analysis. In Figure 6 we show for all the policies and for all four workload types the mean WF slowdown as it depends on the utilization in the system. The last point in all curves is for the highest utilization for which the system is stable in the simulations. In Table I we summarize these maximal utilizations.

As a first general observation, we find that for all policies that do some form of reservation, the maximal utilization is low, or even very low. The worst is a maximal utilization of only 0.2 for the SR policy with the SIPHT workload. Apparently, reserved resources often remain idle, and the benefit of reservations for a short makespan don't balance their negative effect of having long wait times.

Our second general observation is that the performance both in terms of mean slowdown and maximal utilization varies considerably across the three workloads consisting of a single WF type. However, the different WF types almost always have the same relative performance. In particular,



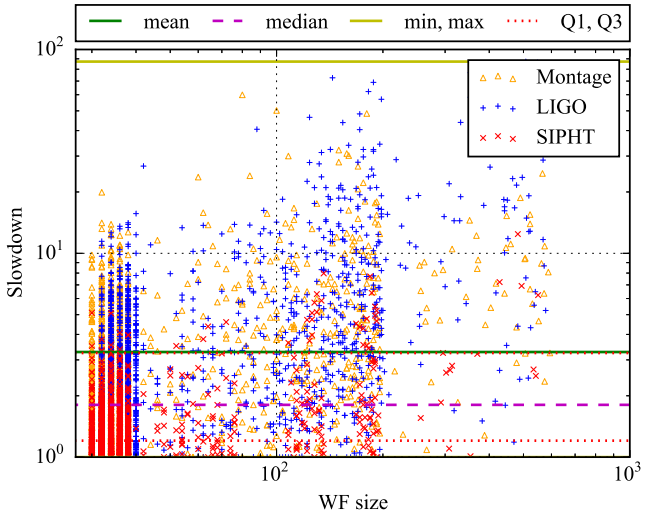
(a) SR (e) FES, depth 10

(b) SLoP, $f = 0.9$ (f) FES, depth 2

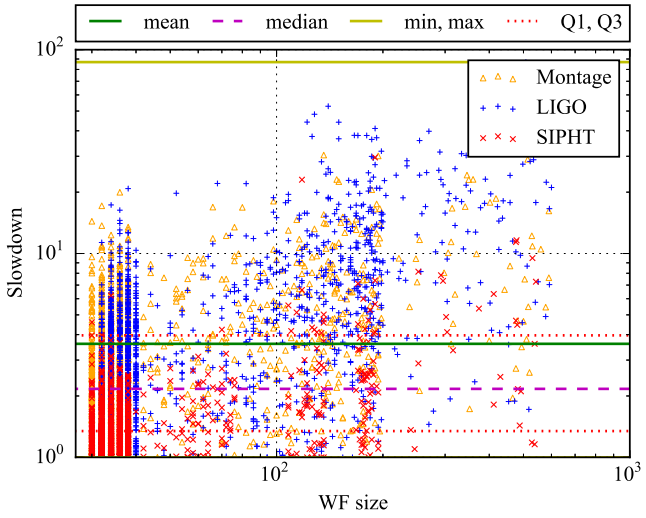
(c) SLoP, $f = 0.8$ (g) FES, depth 1

(d) SLoP, $f = 0.2$ (h) BF

Figure 6: The mean slowdown of WFs as a function of the utilization for the different policies and for each of the four workload types (the vertical axis is in log scale).



(a) SR at a utilization of 0.35



(b) BF at a utilization of 0.95

Figure 7: Scatter plots of the slowdowns versus the sizes of all WFs in the simulations for the mixed workload with the SR and BF policies at their maximal utilizations (both axes are in log scale).

for all the policies except FES with depth 1 and BF, LIGO performs the best and SIPHT performs the poorest. This can be explained from the perspective that the number of reserved processors that is actually used is higher for LIGO than for the other WF types. In contrast, SIPHT and Montage use smaller fractions of the reserved processors.

Thirdly, as can be expected, the performance for the mixture workload is always somewhere in between the performance of the pure workloads. Except for the cases with the FES policy with depth 1 and the BF policy, it has a very low maximal utilization that is well below the maximal

Table I: The maximal utilizations for the considered scheduling policies.

	Workload type			
Policy	Montage	LIGO	SIPHT	Mixture
SR	0.55	0.70	0.20	0.35
SLoP, $f = 0.9$	0.55	0.70	0.20	0.35
SLoP, $f = 0.8$	0.55	0.70	0.25	0.40
SLoP, $f = 0.2$	0.75	0.85	0.50	0.65
FES, depth 10	0.55	0.70	0.20	0.35
FES, depth 2	0.55	0.70	0.20	0.35
FES, depth 1	0.55	0.95	0.80	0.80
BF	0.95	0.95	0.95	0.95

utilizations for the Montage and LIGO workloads.

As to the performance of the SR and SLoP policies, decreasing the scaling factor f improves the performance as the curves for all workloads move to the right in the plots when going down from Figure 6a to Figure 6d. Going from a scaling factor of 1.0 (SR) through 0.9 and 0.8 to 0.2, the maximal utilization for the mixture workload increases from 0.35 (for scaling factors of 1.0 and 0.9) through 0.4 (for a scaling factor of 0.8) to 0.65 (for 0.2). Apparently, when decreasing the scaling factor, the SLoP policy decreases the number of idle but reserved processors.

Our experiments with the FES policy show that increasing the depth decreases the performance because the scheduler reserves ever more processors to each WF. As Figure 6f shows, with a depth of 2 the FES policy already starts to behave like SR, and with a depth of 10 it has almost identical performance to SR. For a depth of 1 the FES policy goes closer to the BF policy, and an interesting effect can be observed. Whereas SIPHT for most policies exhibits the poorest performance, with FES with depth 1 it suddenly achieves a maximal utilization of 0.8. It means that with a depth of 1 the reservation size for SIPHT sharply drops. In contrast, Montage has the same maximal utilization with FES for depths 1, 2, and 10. The reason for this is the structure of the Montage WF, which causes the scheduler to reserve quite a large number of processors even with a depth of 1.

Finally, the BF policy (which is identical to SLoP with a scaling factor of 0.0 and to FES with depth equal to 0) shows by far the best performance results. As can be seen from Figure 6h, it treats all the WF types almost equally, and it is even stable at a utilization 0.95. Apparently, at extremely high utilizations there are always eligible waiting tasks to be found in the queue to start when a task finishes. Overall, we can conclude from Figure 6 that the SR policy is the worst and the BF policy is the best among our policies—reserving processors for WFs with workloads consisting solely of WFs is not a good idea!

In Figure 7 we show scatter plots with the slowdowns versus the sizes of all WFs in the simulations for the mixed

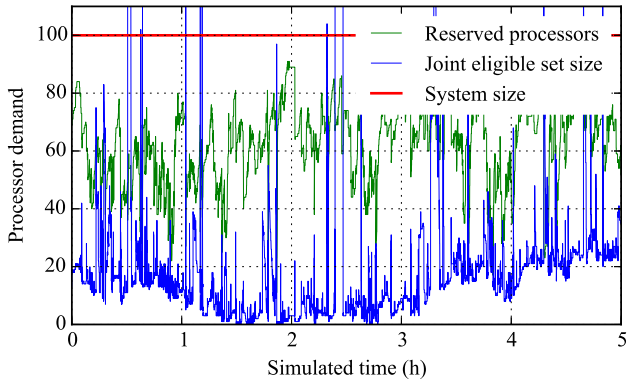


Figure 8: The dynamics of the joint eligible set size and the number of reserved processors during the simulation of the SR policy at a utilization of 0.35 during a 5-hour period.

Table II: The mean wait time and the mean makespan for the small (S), medium (M) and large (L) WFs with the mixed workload for the considered scheduling policies at their maximal utilizations, and in an empty system.

	Mean wait time (s)			Mean makespan (s)		
Policy	S	M	L	S	M	L
Empty system	—	—	—	1136	429	130
SR	636	628	658	1162	484	263
SLoP, f = 0.9	584	578	645	1188	492	249
SLoP, f = 0.8	3195	3121	3329	1174	546	295
SLoP, f = 0.2	879	892	938	1291	597	297
FES, depth 10	637	632	652	1160	480	264
FES, depth 2	526	525	547	1189	503	278
FES, depth 1	596	631	581	1191	530	273
BF	703	707	694	1199	546	271

workload with the SR and BF policies at their maximal utilizations. We show these plots for these two cases as they are the policies that are at the extremes of the spectrum of policies we consider. The most striking thing about these plots is not that they are very different, because in fact, they are not. The striking thing is that they are almost identical at such widely different utilizations (0.35 versus 0.95), exhibiting the huge advantage of using the BF policy. The variation of the density of dots in the horizontal direction is caused by the job size distribution in the used workloads as described in Section IV, with many small jobs of sizes below 40, and much smaller number of medium-sized jobs with sizes between 40 and 200, and a still smaller number of jobs with sizes between 200 and 600. Since we only generated even WF sizes, it also explains the columns for small WF sizes. None of our policies takes into account the size of a WF when processing the queue. Still, in Figure 7 there are somewhat more outliers with slowdowns over 20 among the larger jobs than among the smaller ones, although the difference is not really significant. Among the outliers, most WFs are of the type LIGO, which has the smallest LoP; there are hardly any outliers of type SIPHT.

In Table II we present the mean wait time and the mean makespan for small, medium, and large WFs as defined in Section IV with the mixed workload for all our policies at their maximal utilizations, and in an empty system. As expected, for every policy separately, the mean wait time does not depend on the WF size. The longer mean wait times for the SLoP policy with scaling factor f equal to 0.8 and 0.2 are explained by the step size of 0.05 we use to vary the utilization in our experiments—with a smaller step size to detect the maximal utilization all the mean wait times in Table II would be in the same range. For all policies, the values of the makespans of three groups of WFs are relatively similar. This can be explained from the perspective that when the system is close to its maximal utilization, then despite the length of the queue, the scheduler considers only some set of WFs that are close to the head of the queue. The size of this set is related to the size of the system: the larger the system, the deeper the scheduler should inspect the queue for eligible WF tasks. Another interesting observation is that the larger the WFs, the shorter their makespans. This effect is explained by our usage of the same total execution time distribution for all the WF sizes, and by the fact that smaller WFs have lower levels of parallelism.

Furthermore, while for small and medium WFs the makespans at the maximal utilizations of the policies are almost equal to those in an empty system, large WFs then have makespans that are twice as large. Apparently, even in a busy system small and medium WFs can get a number of processors close to their LoPs, while large WFs suffer more from the presence of other WFs in the system.

Finally, we investigate in more detail how reservations limit the maximal utilization. Figure 8 shows the size of the joint eligible set, which comprises the tasks of the eligible sets of all WFs in the queue, and the number of reserved processors for the SR policy at its maximal utilization during a 5-hour period of the simulation. Obviously, the size of the joint eligible set can exceed the size of the system (in Figure 8 we crop these outliers higher than size 110) and is related to the queue size and the properties of WFs in the queue. The minimum of the two curves gives the number of wasted processors. Apparently, with SR, when the utilization is equal to 0.35, the system capacity spent on reservations approaches 0.65, and the system is saturated.

VI. Related Work

There is an enormous amount of literature on the problem of scheduling WFs. Most of it concentrates on scheduling single WFs in order to minimize the makespan, for which many techniques have been proposed. For instance, the HEFT policy [1] computes the upward rank of a task as the length of

the critical path from that task to the exit task in terms of the time it takes to process the tasks on the path, and schedules the task with the highest upward rank. Previously, we have presented and analyzed a scheduling algorithm for WFs that recursively schedules partial critical paths [4]. There are also several nice overview papers that present and discuss many algorithms for scheduling WFs [24], [25]. Invariably, all of these algorithms assume knowledge of task runtimes.

The problem of scheduling multiple WFs can be split up in the offline problem of scheduling a fixed set of WFs and the problem of online scheduling an arrival stream of WFs as we do in this paper. Two approaches to the offline problem are executing batches with mixtures of multiple WFs [6], and building a single composite WF from multiple WFs and then execute it [5]. In [7], an online stream of WFs is considered, but the authors use a DAG composition approach and task runtime information to prioritize the tasks using HEFT. Also in [26], a stream of arriving WFs is used and the ideas from [7] are extended by considering the critical path for each WF. In [27], the Pegasus planner [28] and the DAGMan [29] batch workflow executor are used. In addition, a scheduling algorithm that allocates resources based on their runtime performance is proposed and real-world experiments are conducted using grid middleware over clusters. Despite the fact that they treat each WF separately without composing a single DAG, their algorithm still uses task runtime information. In [30], a trace of a Teragrid cluster with applications representing (modified) Montage WFs is simulated. Different provisioning policies and priority schemes are considered, with a cap on the amount of resources that can be used by a single WF.

A large body of work proposes backfilling as the main technique to improve the system utilization and reduce the job slowdowns in the area of parallel applications [31]–[35]. Despite the simplicity of the backfilling technique, there exist many variations of the algorithm. For instance, the number of reservations granted by the backfilling algorithm distinguishes two main strategies, conservative and aggressive. The former assigns to each job a reservation when it enters the system and moves smaller jobs forward in the queue as long as no delays are incurred on any of the previously queued jobs. The latter allows any job to be backfilled as long as it does not delay the first job in the queue. The number of queued jobs considered by the backfilling algorithm may have a significant impact on the overall performance. To maximize the utilization, dynamic programming may be employed to find the optimal packing of the jobs [34]. These aspects have been analyzed and incorporated in the Maui project which currently provides a real-world implementation of the general backfill algorithm [35]. The main feature distinguishing backfilling for parallel jobs and WFs is that with WFs, as soon as any number, however low, of resources are available, a WF can make progress. So the concerns about not delaying WFs with backfilling are much less pressing.

VII. Conclusion

In this paper, we have presented a family of four policies for scheduling workloads consisting of arrival streams of WFs with unknown task runtimes. The main distinguishing feature of these policies is to what extent they reserve processors to WFs towards the head of the queue to deal with fluctuations in their level of parallelism. We have simulated these four policies in a cluster with synthetic workloads derived from popular real WFs with as metrics the average WF slowdown and the maximal utilization that can be achieved. Our main conclusion is that any form of processor reservation for WFs without runtime estimates only decreases the overall system performance, leading to low or even to very low maximal utilizations.

As future work, we plan to show the effect of task runtime estimates in our model, we plan to consider workloads consisting of mixes of WFs and other types of jobs, and we plan to include other resource types than processors in our model to assess the performance of I/O-intensive and memory-bound jobs.

References

[1] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 260–274, 2002.

[2] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 506–521, 1996.

[3] M. Rahman, S. Venugopal, and R. Buyya, "A dynamic critical path algorithm for scheduling scientific workflow applications on global grids," in *IEEE International Conference on e-Science and Grid Computing*, 2007.

[4] S. Abrishami, M. Naghibzadeh, and D. H. J. Epema, "Cost-driven scheduling of grid workflows using partial critical paths," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, pp. 1400–1414, 2012.

[5] H. Zhao and R. Sakellariou, "Scheduling multiple DAGs onto heterogeneous systems," in *20th International Parallel and Distributed Processing Symposium*, 2006.

[6] A. Hirales-Carbajal *et al.*, "Multiple workflow scheduling strategies with user run time estimates on a grid," *Journal of Grid Computing*, vol. 10, pp. 325–346, 2012.

[7] Z. Yu and W. Shi, "A planner-guided scheduling strategy for multiple workflow applications," in *International Conference on Parallel Processing-Workshops*, 2008.

[8] S. Bharathi *et al.*, "Characterization of scientific workflows," in *Third Workshop on Workflows in Support of Large-Scale Science*, 2008.

[9] G. Juve *et al.*, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, pp. 682–692, 2013.

[10] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, *Workflows for e-Science*. Springer-Verlag London Limited, 2007.

[11] S. Ostermann, A. Iosup, R. Prodan, T. Fahringer, and D. H. J. Epema, "On the characteristics of grid workflows," in *CoreGRID Workshop on Integrated Research in Grid Computing*, 2008.

[12] A. Iosup, D. H. J. Epema, T. Tannenbaum, M. Farrellee, and M. Livny, "Inter-operating grids through delegated matchmaking," in *ACM/IEEE Conference on Supercomputing*, 2007.

[13] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, pp. 529–543, 2001.

[14] S. Ikiz and V. K. Garg, "Online algorithms for Dilworth's chain partition," Parallel and Distributed Systems Laboratory, Department of Electrical and Computer Engineering, University of Texas at Austin, Tech. Rep.

[15] R. P. Dilworth, "A decomposition theorem for partially ordered sets," *Annals of Mathematics*, vol. 51, pp. 161–166, 1950.

[16] G. Juve *et al.* Synthetic workflow generators. [Online]. Available: www.github.com/pegasus-isi/WorkflowGenerator

[17] D. Talby and D. G. Feitelson, "Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling," in *13th International Parallel Processing Symposium / 10th Symposium on Parallel and Distributed Processing*, 1999.

[18] A. Iosup, O. Sonmez, and D. Epema, "DGSim: Comparing grid resource management architectures through trace-based simulation," in *Euro-Par Parallel Processing*, 2008.

[19] J. C. Jacob *et al.*, "Montage: An astronomical image mosaicking toolkit," *Astrophysics Source Code Library*, vol. 1, p. 10036, 2010.

[20] A. Ramakrishnan *et al.*, "Scheduling data-intensive workflows onto storage-constrained distributed resources," in *7th IEEE International Symposium on Cluster Computing and the Grid*, 2007.

[21] J. Livny, "Bioinformatic discovery of bacterial regulatory RNAs using SIPHT," in *Bacterial Regulatory RNA*, 2012.

[22] B. Abbott *et al.*, "Search for gravitational waves from binary inspirals in S3 and S4 LIGO data," *Physical Review D*, vol. 77, p. 062002, 2008.

[23] U. Lublin and D. G. Feitelson, "The workload on parallel supercomputers: Modeling the characteristics of rigid jobs," *Journal of Parallel and Distributed Computing*, vol. 63, pp. 1105–1122, 2003.

[24] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys*, vol. 31, pp. 406–471, 1999.

[25] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *Journal of Parallel and Distributed Computing*, vol. 59, pp. 381–422, 1999.

[26] C.-C. Hsu, K.-C. Huang, and F.-J. Wang, "Online scheduling of workflow applications in grid environments," *Future Generation Computer Systems*, vol. 27, pp. 860–870, 2011.

[27] K. Lee *et al.*, "Adaptive workflow processing and execution in Pegasus," *Concurrency and Computation: Practice and Experience*, vol. 21, pp. 1965–1981, 2009.

[28] E. Deelman *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, pp. 219–237, 2005.

[29] J. Frey, "Condor DAGMan: Handling inter-job dependencies," Department of Computer Science, University of Wisconsin, Tech. Rep., 2002.

[30] G. Singh and E. Deelman, "The interplay of resource provisioning and workflow optimization in scientific applications," *Concurrency and Computation: Practice and Experience*, vol. 23, pp. 1969–1989, 2011.

[31] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, pp. 529–543, 2001.

[32] D. A. Lifka, "The ANL/IBM SP scheduling system," in *Job Scheduling Strategies for Parallel Processing*, 1995.

[33] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan, "Scheduling of parallel jobs in a heterogeneous multi-site environment," in *Job Scheduling Strategies for Parallel Processing*, 2003.

[34] E. Shmueli and D. G. Feitelson, "Backfilling with lookahead to optimize the packing of parallel jobs," *Journal of Parallel and Distributed Computing*, vol. 65, pp. 1090–1107, 2005.

[35] D. Jackson, Q. Snell, and M. Clement, "Core algorithms of the Maui scheduler," in *Job Scheduling Strategies for Parallel Processing*, 2001.