The Impact of Task Runtime Estimate Accuracy on Scheduling Workloads of Workflows

Ilyushkin, Alexey; Epema, Dick

DOI
10.1109/CCGRID.2018.00048

Publication date
2018

Document Version
Accepted author manuscript

Published in
18th IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing

Citation (APA)
Ilyushkin, A., & Epema, D. (2018). The Impact of Task Runtime Estimate Accuracy on Scheduling Workloads of Workflows. In *18th IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing* (pp. 331-341) https://doi.org/10.1109/CCGRID.2018.00048

The Impact of Task Runtime Estimate Accuracy on Scheduling Workloads of Workflows

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

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

***Abstract*—Workflow schedulers often rely on task runtime estimates when making scheduling decisions, and they usually target the scheduling of a single workflow or batches of workflows. In contrast, in this paper, we evaluate the impact of the absence or limited accuracy of task runtime estimates on slowdown when scheduling complete workloads of workflows that arrive over time. We study a total of seven scheduling policies: four of these are popular existing policies for (batches of) workloads from the literature, including a simple backfilling policy which is not aware of task runtime estimates, two are novel workload-oriented policies, including one which targets fairness, and one is the well-known HEFT policy for a single workflow adapted to the online workload scenario. We simulate homogeneous and heterogeneous distributed systems to evaluate the performance of these policies under varying accuracy of task runtime estimates. Our results show that for high utilizations, the order in which workflows are processed is more important than the knowledge of correct task runtime estimates. Under low utilizations, all policies considered show good results, even a policy which does not use task runtime estimates. We also show that our Fair Workflow Prioritization (FWP) policy effectively decreases the variance of workflow slowdown and thus achieves fairness, and that the plan-based scheduling policy derived from HEFT does not show much performance improvement while bringing extra complexity to the scheduling process.**

I. Introduction

In workloads of modern computing systems, workflows are often used as a tool to drive complex computations, and their popularity continues to increase [1]. Many of these workflows are usually submitted to the system repeatedly so that (statistical) runtime estimates of their tasks can be derived [2], [3]; alternatively, runtime estimates can be provided by users [4]. However, the accuracy of runtime estimates significantly depends on the estimation algorithm used, or on the user—user runtime estimates can be very unreliable [5], [6]. Most previous work on scheduling workflows [7]–[10] has assumed some (and often even perfect) knowledge of task runtimes. Moreover, often has been considered the *offline* problem of scheduling a single workflow or a batch of workflows (which are all initially present), or periodic submissions with a fixed interval. In most cases, the makespan has been used as the main metric.

However, workflows may be submitted to a system over time according to some arrival pattern, in which case job slowdown is a much more appropriate performance metric. Then, especially when workflows of widely different sizes are submitted, fairness becomes an issue, and an important goal is to reduce the variability of job slowdown. In this paper, we investigate how the accuracy of task runtime estimates affects the quality of scheduling, and we address the issue of fairness, in the *online* case of scheduling complete workloads of workflows. Moreover, we evaluate the system stability to know at which workflow arrival rates the system starts to uncontrollably accumulate waiting workflows. Besides that, we identify the maximal achievable system utilization which guarantees the stability.

We distinguish *dynamic* and *plan-based* policies. Dynamic policies make task placement decisions just-in-time when a processor becomes idle or a new task becomes eligible. Plan-based policies construct a full-ahead plan on every workflow arrival and strictly follow this plan to perform task placements between the workflow arrivals.

Task runtime estimates have been heavily used for different forms of task prioritization by various scheduling algorithms for single workflows and for batches of workflows. The most popular approaches [4], [7], [10], [11] include upward and downward ranking and different forms of list scheduling techniques. Workflow tasks are usually prioritized in ascending or descending order of their runtimes or by their proximity to the entry or exit task. The individual workflows are often prioritized based on the length of their critical path (a longest path from an entry to the exit task). In this situation, inaccurate runtime estimates can significantly affect the task and workflow ranking, as not only the length of the critical path can be affected but even a wrong critical path can be used. Knowing how the quality of estimates affects the performance helps to create better error-resilient scheduling policies and is useful when selecting policies which are less sensitive to incorrect estimates. In our previous work [12], we used a different approach where we scheduled an arriving stream of workflows without using any runtime estimates at all, and completely relied on the structure of the workflows when making scheduling decisions.

To study the influence of the accuracy of task runtime estimation on the performance, in this paper we study a total of seven scheduling policies for *workloads of workflows*, and we simulate their execution on two workloads of realistic workflows. Four of these are existing dynamic workflow scheduling policies, namely Greedy Backfilling (GBF), which came out best of all the policies we proposed [12] in case no runtime estimates are available, Online Workflow Management (OWM) [10], [13], Fairness Dynamic Workflow Scheduling (FDWS) [9], [10], and Rank_Hybd (HR) [8]. We also propose the simple Critical Path Prioritization (CPP) policy and, in order to address the issue of fairness, the Fair Workflow Prioritization (FWP) policy. To check how existing plan-based scheduling algorithms can be applied when scheduling workloads of workflows, we have adapted the Heterogeneous Earliest Finish Time (HEFT) [7] policy to the online case. All policies except GBF require task runtime estimates for their operation.

The main contribution of this paper is threefold:

1) We propose two novel dynamic workflow scheduling policies (CPP and the fairness-oriented FWP), and we adapt the popular HEFT policy to the online case (Section III).
2) We show how inaccurate task runtime estimates affect the performance and fairness of scheduling workloads of workflows (Section V).
3) We demonstrate how the knowledge of task runtime estimates improves the performance at high system utilizations (Section V-A), and how the plan-based approach struggles to deal with workloads of workflows (Section V-C).

II. Problem Statement

This section presents our model for the problem of using runtime estimates for scheduling workloads of workflows and the performance metrics.

A. The Model

We consider large-scale homogeneous and heterogeneous computing systems such as large clusters and datacenters that are subject to an arrival stream of workflows. We only consider processors as the type of system resources that can be controlled by the scheduler. The workflows in the stream arrive according to a Poisson process, and a single workflow in the workload is considered as a job. We suppose that the selected arrival process is representative as our system models a public distributed system which serves multiple independent customers.

We schedule on this system a stream of synthetic workflows where each workflow (WF, Directed Acyclic Graph (DAG)) consists of a set of tasks with precedence constraints among them. Accordingly, the *size* of a workflow is defined as the number of tasks it has. Each workflow task can start its execution only when all of its precedence constraints are satisfied (e.g., when all the required input files are available). We assume that each workflow task requires only one processor to run. The execution time of a task on a processor is proportional to the processor speed. All the considered workflow 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.

Since we focus on the computational properties of workflows, we assume that the data transfer times between workflow tasks in our simulated system can be neglected. This is equivalent to the situation in a real system where the computing nodes are connected to a shared file system so that the input data is available for any task almost immediately after its parents finish.

The runtime estimates are often extracted from historical runs, simulations of workflow executions, or even are obtained from users [5], [14]. However, the quality of such estimates can vary significantly depending on the estimation method. To study the effect of the quality of task runtime estimates on the system performance, we modify the perfect task runtimes obtained from the synthetic workload using a certain pre-defined *error factor* f_e. The error allows to either under-estimate or over-estimate the task runtimes. All of the evaluated schedulers are not aware of under- or over-estimation. They can only derive the error in task runtimes post factum by comparing the given runtime estimates with the actual task runtimes obtained during the execution (as in our FWP policy, Section III-H). We use three methods to introduce the estimation errors:

- *Static error:* Here we multiply the runtime of every task of every workflow by the error factor f_e.
- *Random error I:* Here we multiply the runtime of every task within a single workflow by the same random error factor. This random error factor is independently generated for every workflow in the workload by drawing it from the uniform distribution on the interval $(0, f_e \times 2]$. So on average, task runtimes in the workload are under- or overestimated by a factor of f_e.
- *Random error II:* Here we multiply the runtime of every task by an individually generated random error. The runtime estimate of a task is computed by multiplying its original correct task runtime d_r by a random error factor drawn from the uniform distribution on the interval $(0, f_e/d_r \times 2]$. This is an extreme case of introducing an error, as it changes the distribution of task runtimes to a uniform distribution on the interval $(0, 2f_e]$. Thus, the scheduler operates with estimates which are very far from the original ones.

B. Performance Metrics

In order to compare the implemented scheduling policies, we define a set of metrics and baselines. Scheduling workflows is not work-conserving in that there may be non-eligible tasks waiting in the queue while at the same time, there are idle processors in the system. As a consequence, policies scheduling workloads of workflows may not be able to drive a system up to a utilization of 100%. Therefore, we use the maximal utilization as a system-oriented metric to assess the performance of workflow 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 workflow scheduling policies we use the (average) slowdown, which is defined in steps in the following way:

- The *wait time* t_w of a workflow is the time between its arrival and the start of its first task.
- The *execution time* t_e of a workflow is the sum of the runtimes of all its tasks.
- The *makespan* t_m of a workflow is the time between the start of its first task until the completion of its last task.
- The *response time* t_r of a workflow is the sum of its wait time and its makespan: $t_r = t_w + t_m$.
- The *slowdown* s of a workflow is its response time (in a busy system, when the workflow runs simultaneously with other workflows) normalized by the length c of its critical path: $s = t_r \,/\, c$.

III. Scheduling Policies

In this section, we first provide some definitions and explain the upward rank, which is crucial for task prioritization in almost all of the scheduling policies we consider. Then we present GBF the greedy backfilling dynamic policy which does not use task runtime estimates at all, and five dynamic policies: CPP, OWM, FDWS, HR, and FWP, that require task runtime estimates. Dynamic policies make scheduling decisions whenever a new task becomes eligible or a processor becomes idle. Finally, we present the plan-based WHEFT policy which uses task runtime estimates to construct a full ahead execution plan on every workflow arrival; between arrivals, the execution is completely guided by the precomputed plan. We classify the considered policies by their distinctive properties in Table I. The GBF policy is included in the paper as it has shown good performance in our previous work [12]. The dynamic policies that require task runtime estimates have been selected based on the comparative study by Arabnejad et al. [10]. We choose HEFT as it is one of the most popular algorithms for workflow scheduling and it is often used as a reference [15].

A. Definitions

Scheduling workloads of workflows often operates with the notions of eligible set and level of parallelism. For a workflow, 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. For a workflow 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 workflow can only stay the same or decrease during its execution. The LoP can be computed exactly [16] or approximately [12]. In this paper, we approximate LoP by dividing the total execution time t_e of a workflow by the length c of its critical path.

B. The Upward Rank Computation

The *upward rank* is often used to prioritize tasks in workflows based on their duration and proximity to the exit task (e.g., in HEFT [7]). For each task n_i in a workflow, the *upward rank* r_u is recursively calculated, starting from the exit task, using the following formula:

$$r_u(n_i) = \overline{e_i} + \max_{n_j \in S(n_i)} (\overline{c_{i,j}} + r_u(n_j)),$$

where $\overline{e_i}$ is the average estimated execution time of task n_i, $S(n_i)$ is the set of immediate successors of task n_i, and $\overline{c_{i,j}}$ is the average communication delay between tasks n_i and n_j. Since the exit task has no successors, its upward rank is just equal to its average estimated execution time. The average estimated execution time $\overline{e_i}$ is calculated for each task using the average speed of the processors in the system. The average estimated communication cost $\overline{c_{i,j}}$ is calculated as the average communication start-up time plus the size of the data to be transmitted, divided by the average transfer rate between the processors. The length c of the critical path of a workflow is equal to the maximum value of r_u among all the workflow tasks N:

$$c = \max_{n_i \in N}(r_u(n_i)).$$

By workflow *length* we mean the length of its critical path.

Table I: The distinctive properties of the considered policies.

Property	Policies						
	GBF	CPP	OWM	FDWS	HR	FWP	WHEFT
Proposed in this paper	−	+	−	−	−	+	+
Plan-based	−	−	−	−	−	−	+
Explicit job queue (FCFS)	+	+	−	−	−	−	−
Joint eligible set (one task per WF)	−	−	+	+	−	+	−
Joint eligible set (all eligible tasks)	−	−	−	−	+	−	−
Fairness-aware	−	−	−	+	−	+	−

C. Greedy Backfilling

The simple *Greedy Backfilling* (GBF) policy is an application of greedy backfilling to workflow scheduling which we proposed in our previous work [12]. This policy processes workflows in FCFS order, and does not require task runtime estimates for its operation. In GBF, on every invocation, the scheduler, starting from the head of the queue, selects the first workflow with a non-empty eligible set, randomly picks a task from it, assigns it to the first available fastest processor, and removes it from the set. It continues to do this until the eligible set of the workflow is empty or until there are no more idle processors. When the eligible set is empty but the system still has idle processors, the scheduler takes the eligible set of the next workflow in the queue, and so forth.

D. Critical Path Prioritization

Our *Critical Path Prioritization* (CPP) policy extends our GBF policy. In CPP, on every invocation, the scheduler, starting from the head of the queue, selects the first workflow with a non-empty eligible set, picks the task from it with the highest r_u, assigns it to the first available fastest processor, and removes it from the set. For the rest, the CPP scheduler is similar to GBF.

E. Online Workflow Management

The *Online Workflow Management* (OWM) policy [10], [13] maintains a single joint eligible set which contains only a single eligible task (if any) with the highest r_u from every workflow in the system. At every scheduler invocation, as long as the system has workflows with eligible tasks, the scheduler selects the task with the highest r_u from the joint set. If the idle processors have the same speed, OWM finds the busy processor which will become idle earlier than any other busy processor. If the estimated finish time of the selected task on that busy processor is smaller than EFT on any of the idle processors, the task is postponed (stays in the joint set) until the next scheduler invocation. Otherwise, the task is assigned to any of the idle processors. If the idle processors have different speeds, the task is assigned to the fastest idle processor.

F. Fairness Dynamic Workflow Scheduling

The *Fairness Dynamic Workflow Scheduling* (FDWS) policy [9], [10] maintains a single joint eligible set which is formed in the same way as in OWM. However, within the joint set each task of a workflow j is additionally prioritized with rank r_a (highest first) which considers the fraction of remaining tasks of the workflow and the length of its critical path. The additional rank r_a is defined as follows:

$$r_{a,j} = \left(\frac{m_j}{p_j} \times c_j \right)^{-1},$$

where m_j is the number of unfinished (not yet eligible or eligible) tasks in workflow j, p_j is the total number of tasks in the workflow, c_j is the initial length of the workflow (at the moment of its arrival to the system). The first factor in the formula prioritizes workflows with lower fractions of remaining tasks, while the second factor in the formula gives priority to shorter workflows. There are two versions of the FDWS policy in the literature. The first version considers both idle and busy processors for task allocation. If the selected processor is busy the task is placed in its task queue. The second version considers only idle processors. In both cases the processor allowing the lowest estimated finish time for the task is selected. For better comparability with other considered policies, in this paper we use the version of the FDWS policy [10] without per processor queues.

G. Hybrid Rank

The *Hybrid Rank* (HR, the original name is Rank_Hybd [8]) policy maintains a single joint eligible set of *all* the eligible tasks from all the workflows in the system. On arrival of a workflow, the policy computes r_u for all its tasks. At every scheduler invocation, if the tasks in the joint set belong to different workflows, the scheduler selects the task with the lowest r_u. If the tasks in the joint set are from the same workflow, the algorithm selects the task with the highest r_u. On the one hand, the HR policy tries to achieve fairness by allowing shorter workflows to start their execution earlier. On the other hand, during the execution of a workflow, the length of the remaining part of its critical path decreases as more tasks finish. Although HR could delay longer workflows just after their arrival, the policy gives them more preference when they are about to finish.

H. Fair Workflow Prioritization

We propose the *Fair Workflow Prioritization* (FWP) policy which is similar to OWM and FDWS in the way it forms the single joint eligible set, but which uses a different mechanism to compute task priorities to achieve even better fairness than FDWS. On every workflow completion, by averaging historical slowdowns of previously finished workflows, FWP computes the *target slowdown* which all the workflows in the system are supposed to experience. The workflows are prioritized based on their proximity to the target slowdown. The lower the current slowdown of a workflow than the target slowdown, the lower its priority, the higher the current slowdown than the target slowdown, the higher its priority.

FWP allows to achieve better fairness when scheduling multiple workflows simultaneously, as the acceleration of certain workflows is done at the cost of decelerating others. Thus, the number of possibilities to slow down a certain workflow is limited by the number of workflows present in the system. To achieve the same target slowdown, workflows with longer critical paths should be delayed more compared to workflows with shorter critical paths. If the system does not have enough concurrent workflows, the workflows with longer critical paths will experience lower slowdowns than the target. An alternative solution is to postpone certain workflows by periodically excluding their tasks from the joint eligible set and making them eligible for scheduling after a timeout. However, we keep this improvement for future work.

We calculate the *target slowdown* s_t based on the history of slowdowns s_i of K previously finished workflows by averaging them: $s_t = \sum_{i=1}^{K} s_i / K$. When the history is empty, the system is initialized with $s_t = 1$. For workflow j, its *current slowdown* $\hat{s}_j$ is calculated as:

$$\hat{s}_j = (\hat{t}_{r,j} + \hat{c}_j \xi)/(c_j \xi),$$

where $\hat{t}_{r,j}$ is the current residence time of workflow j from its arrival till now, $\hat{c}_j$ is the length of the critical path of the remaining part of the workflow (which is not yet running), c_j is the length of the workflow, and ξ is the correction coefficient which is required to cope with possibly incorrect estimates. Since FWP depends on critical path length to calculate $\hat{s}$, incorrect task runtime estimates could affect the ranking. Thus, after the completion of each task, the policy stores its actual measured runtime d_m and its estimated runtime d_e, and computes a correction coefficient ξ using information about the M tasks that finished last:

$$\xi = \frac{\sum_{i=1}^{M} d_{m,i}}{M} / \frac{\sum_{i=1}^{M} d_{e,i}}{M}.$$

To prioritize the task from workflow j within the joint eligible set, FWP uses rank r_b which is calculated as:

$$r_{b,j} = \hat{s}_j - s_t,$$

where $\hat{s}_j$ is the current slowdown of workflow j and s_t is the target slowdown.

I. Workload HEFT

The *Workload HEFT* (WHEFT) policy is our adaptation of HEFT policy [7] for scheduling workloads of workflows. In addition to the *scheduler*, WHEFT uses a separate *planner* which maintains a global execution plan for all the workflows in the system. For each non finished task in the workflow, the plan defines the processor where and when the task should run. On every new workflow arrival a completely new global plan is created. Between the workflow arrivals the execution is completely guided by the scheduler using the plan.

Since the workload is an arriving stream of incoming workflows, to be able to apply HEFT it is required to combine the workflows in the system into one. For that, we use an *Alternating DAGs* approach proposed by Zhao and Sakellariou [15] as in the original paper it showed better performance compared to other approaches from the same group. To apply the Alternating DAGs approach, WHEFT planner first combines the workflows in the system by adding a single joint exit node. Then it computes upward ranks r_u for all the tasks within the new combined workflow. Further, using the Hybrid policy [11], WHEFT splits the combined workflow into levels where each level contains only independent tasks. The tasks within each level are grouped according to the original workflow where they belong to. WHEFT switches between the groups in a round robin manner to make a sorted list of tasks in (descending order of their r_u). The plan is created by sequentially traversing the levels and sequentially processing the sorted lists of tasks made from the groups by applying HEFT to them.

The WHEFT scheduler is called after the plan construction and after each task completion. The scheduler sequentially checks the plan and tries to assign non-running tasks from the plan to according processors. The tasks are checked for eligibility in the ascending order of their planned start times. If a task is not yet eligible, the scheduler proceeds to the next processor. When a task finishes, the scheduler removes it from the plan. The scheduler does not perform any task preemption. If, according to the plan, a certain task should be currently started, but the processor where it should run is still busy (as the plan could be incorrect due to erroneous runtime estimates), the task which occupies the processor runs until its completion.

IV. Experimental Setup

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

A. Workloads

In our simulations, we use two workloads with an arrival process of workflows, and a batch of workflows that are all submitted simultaneously. Workload I mixes equal fractions of three representative types of workflows and is generated using the workflow generator [17] presented by Bharathi et al. [18]. The workflows are taken from different application domains, i.e., astronomy (Montage [2], [18]–[20]), physics (LIGO [2], [18], [20], [21]), and bioinformatics (SIPHT [2], [18], [22]). 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) [23] and its mission is to detect gravitational waves predicted by general relativity. SIPHT helps to search for small untranslated bacterial regulatory RNAs.

Montage has the most complicated structure and its size is determined by the number of processed images. A LIGO workflow usually consists of many smaller workflows combined into a single workflow. Similarly to LIGO, the SIPHT workflow combines smaller independent workflows, but with very similar structures. The workflow types are diverse not only in the structure of their DAGs, but also with respect to the processing requirements of their tasks as we will see later in this section.

Workload II solely consists of random workflows generated using an existing random DAG-generator created by Suter et al. [24]. The generator has four configuration parameters: $jump$ sets the maximum number of workflow levels induced by the inter-task dependencies, $regular$ specifies the regularity of the task distribution across workflow levels, fat specifies the width (LoP) of the workflow, and $density$ specifies the numbers of dependencies between tasks of two consecutive workflow levels. The values for these parameters we use are selected uniformly from the following sets: $jump = 1, 2, 3$, $regular = 0.2, 0.8$, $fat = 0.2, 0.8$, and $density = 0.1$. We use only a single and relatively low value for $density$ since for large workflows (with several hundreds of tasks and more), higher densities significantly increase the complexity of finding a critical path. More information on the parameterisation of the random DAG generator can be found in its code repository [24].

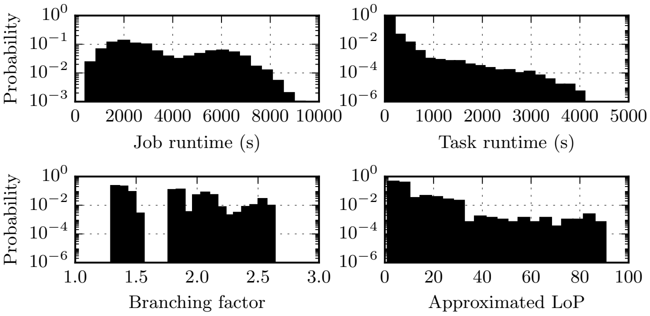


Figure 1: Statistical characteristics of Workload I. The vertical axes have a log scale.

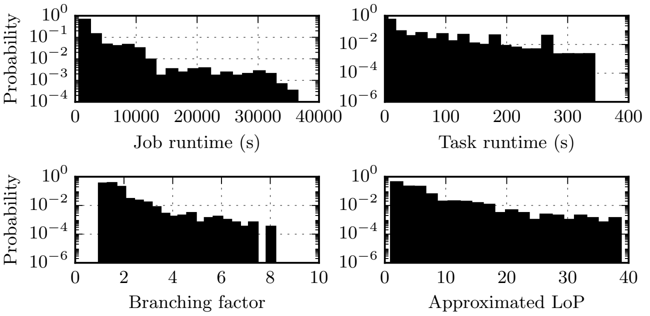


Figure 2: Statistical characteristics of Workload II. The vertical axes have a log scale.

For the total workflow execution time in Workload I, we use a two-stage hyper-Gamma distribution derived from the model presented in [25]. The shape and scale parameters (α, β) of the 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. Figure 1 visualizes this distribution. In Workload II we use the original total execution time distribution obtained from the generator, see Figure 2. In order to obtain simulation results for the different workflow types that can easily be compared, in both workloads we use the same average total execution time of one hour. For every workflow in both workloads, we normalize the generated task runtimes so that its total processing requirement is equal to the corresponding sample of the execution time distribution.

In Figures 1 and 2 we show the distributions of the task runtimes, the branching factors (the total number of inter-task links in a workflow divided by its size), and the approximated LoPs (t_e/c). The two workloads share the phenomenon that they are dominated by short tasks. However, the task runtimes in Workload I are an order of magnitude longer than in Workload II. At the same time, the range of the total job runtimes in Workload II is four times as large as in Workload I. Interestingly, Workload II also shows a higher diversity of branching factors but a twice smaller approximated LoP.

For each utilization level, both workloads consist of three unique sets of workflows with 3,000 workflows each, which allows us to perform three independent simulation runs per utilization level. As with many other workloads in computer systems, in practice, workflows are usually small, but very large ones may exist too [26]. Therefore, in our simulations we distinguish small, medium, and large workflows, defined by sizes that are uniformly distributed on the intervals [30,38], [40,198], and [200,600], respectively (all workflows are assumed to have even sizes). The small, medium, and large workflows constitute fractions of 75%, 20%, and 5% of the workload.

Finally, we use a batch of 1,000 workflows consisting of workflows from Workload I. Accordingly, the statistical characteristics of the batch are similar to those of Workload I.

B. Simulation Environment

We have modified the DGSim simulator [27], [28] for cluster and grid systems to include the workflow scheduling policies we consider. The size of the cluster we use in all of our simulations is 100 single-processor nodes. We vary the accuracy f_e of the estimates using the following values: 0.1, 2, 5, and 10. As in our model the communication overhead is not considered, the r_u values are computed only using the average estimated execution time. We suppose that after a task has been assigned to a processor, it runs there until its completion. For our FWP policy we choose values of $K = 300$ and $M = 1000$ (see Section III-H).

We mostly focus on a homogeneous system where all the processors have an average processing speed of 1 workflow/hour. However, we also perform a set of experiments with a heterogeneous system with two equally sized groups of processors: fast processors with an average processing speed of 1.5 workflow/hour and slow processors with an average processing speed of 0.5 workflow/hour.

For the majority of the simulations we use a system utilization of 98% since all the considered dynamic policies can handle such high utilizations (we show this later in Section V). We only show results when the system is in steady state, i.e., when reporting performance results for workloads, we omit the performance information for the first 1000 workflows and last 1000 workflows in each simulation.

C. System Stability Validation

To be able to clearly distinguish situations when the system is or is not stable in a long-term perspective, we use two methods: the statistical system stability check proposed by Wieland et al. [29], as well as the Lyapunov drift theorem [30]. For every simulation run, we perform both stability tests. The system is considered stable if each method shows at least two stable results out of three.

According the Wieland approach, we take the observed number of workflows in the system $N(t)$ (both queued and partially running) at every moment t, where $0 \leq t \leq \tau$ with τ the total duration of the simulation, and split the observations into b batches, where $b = 10$. Then for each batch $j = 2, \ldots, b$ we compute the time average number of workflows in the system within the batch as:

$$\hat{\lambda}_{N,j} = \int_{(j-1)\tau/b}^{j\tau/b} \frac{N(t)}{\tau/b} dt.$$

Then we compute the difference between the last and second batch observations: $\hat{\lambda}_N = \hat{\lambda}_{N,b} - \hat{\lambda}_{N,2}$, and compute the variance of batch observations σ^2 with $b-1$ degrees of freedom. We conclude whether the system is stable if

$$\frac{\hat{\lambda}_N}{\sqrt{2}\sigma} > t_{1-\alpha,b-2},$$

where $t_{1-\alpha,b-2}$ is the $1-\alpha$ Student-T quantile with $b-2$ degrees of freedom. For the default values of $b = 10$ and $\alpha = 0.05$, $t_{1-\alpha,b-2} = 1.86$, thus, stability is rejected if $\hat{\lambda}_N > 2.63\sigma$.

For the Lyapunov drift-based stability check we compute the mean Lyapunov drift throughout the simulation as follows: $\delta(t) = l(t) - l(t-1)$, where $l(t) = N(t)^2/2$. A low value of the mean Lyapunov drift after the initial transient indicates that the system converges and is stable. For our system we experimentally derive a threshold of 1 for the mean Lyapunov drift; if the mean drift exceeds 1, the system is considered unstable.

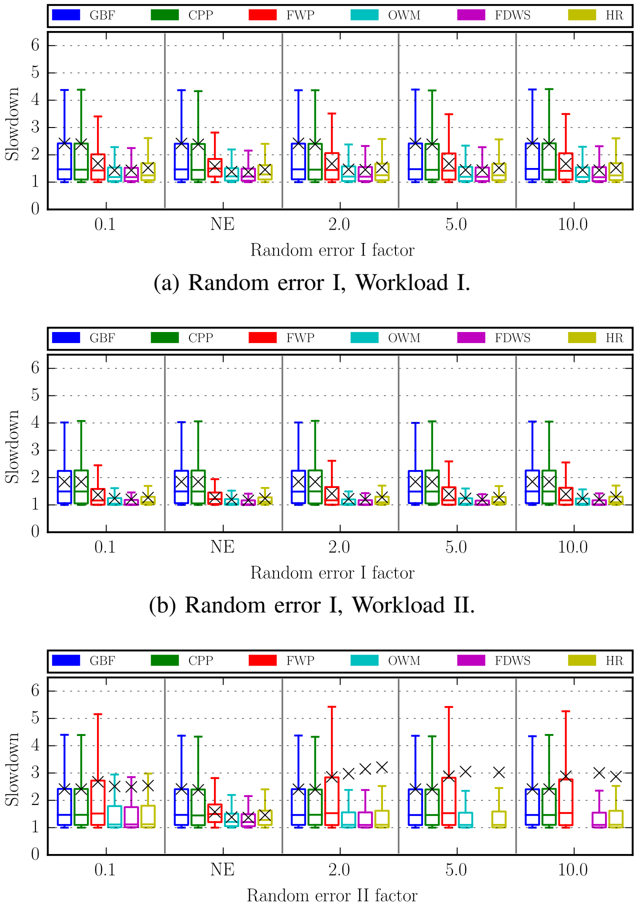
V. Experimental Results

In this section, we present our experimental results. We first investigate how varying the error in task runtime estimates affects the slowdowns of workflows in homogeneous and heterogeneous systems for the six dynamic scheduling policies we consider. Then we show the performance of the plan-based WHEFT policy and the performance when scheduling batches of workflows.

A. Performance of Dynamic Policies

First, we compare the performance of the six dynamic policies we consider in a homogeneous system with the three methods for introducing estimation errors. All of these policies are able to achieve a 98% system utilization without destabilizing. Figures 3a and 3b show the workflow slowdown distribution and standard deviation versus the error factors in our two random error methods for both workloads. We do not show outliers in these figures and set the whisker boundaries within 1.5 times of the interquartile range. Changing the runtime estimates by a static factor does not affect the performance of any of the dynamic policies. The reason is that all the task upward ranks simultaneously scale in the same way, and that as a consequence, the order of selecting workflow tasks for execution is not affected. Therefore, we do not present the results for the Static error method as they are identical to the No Error results in Figures 3 and 4. Without an error, the mean number of workflows (fully or partially running and waiting) in the system throughout the experiment varies from 35 (CPP) to 40 (HR).

For the Random error I method, in Figures 3a and 3b we see that the performance of the policies is largely insensitive to the value of the error factor. We also find that the GBF and CPP policies, which both employ the FCFS principle, exhibit a much poorer mean slowdown and higher percentiles than the other policies. Moreover, the CPP policy exhibits only a slight decrease in the mean slowdown and the standard deviation with Workload I (see Figure 4a) over the GBF policy, which shows that the way in which it uses task runtime estimates is not effective. Notably, for our FWP policy the results for the No Error case are definitely the best. It also achieves lower values of the standard deviation with an error factor of 2.0 for Workload I, at the cost of an increased mean slowdown. In the other cases, FWP shows comparable or slightly higher standard deviation than the other policies, except GBF and CPP.

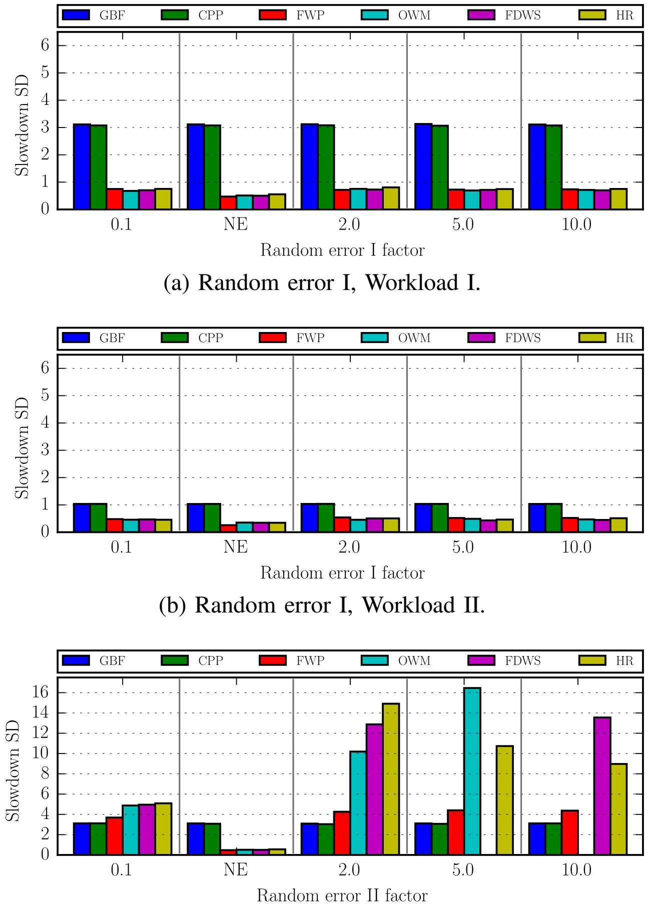


(a) Random error I, Workload I.

(b) Random error I, Workload II.

(c) Random error II, Workload I.

Figure 3: Slowdown versus the error factor at 98% system utilization in a homogeneous system. NE is No Error, means are marked with ×. Missing bars indicate unstable situations.



(a) Random error I, Workload I.

(b) Random error I, Workload II.

(c) Random error II, Workload I. The vertical axis has larger scale.

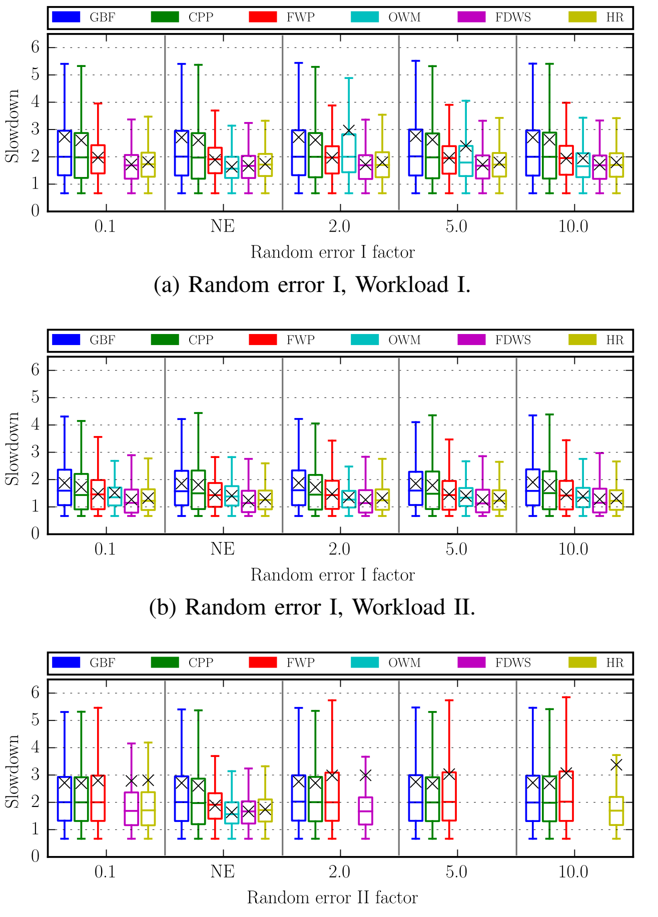
Figure 4: Slowdown standard deviation versus the error factor at 98% system utilization in a homogeneous system. NE is No Error. Missing bars indicate unstable situations.

In contrast, using the Random error II method for varying the estimation error factor (see Figures 3c and 4c) does affect the slowdowns of the workflows in Workload I, creating many outliers and significantly increasing the mean slowdowns and standard deviations for all policies except GBF and CPP. OWM and FDWS even destabilize at high over-estimation factors, but stay stable at lower 97% utilization for all the error factors. However, our FWP policy shows the lowest values of the standard deviation compared to OWM, FDWS, and HR due to its correction mechanism for task runtime estimates (Figure 4c). At the same time, the Random error II method hardly affects Workload II and shows similar results as Random error I. Thus, we omit the results for Random error II with Workload II as they look identical to the Figures 3b and 4b. The statistical characteristics of the workloads (Figures 1 and 2) show the cause of this observation: Workload I has a much higher variability of task runtimes.

B. *Effects of Heterogeneity*

We conduct the same set of experiments with the dynamic policies as in Section V-A in a heterogeneous system and, analogously, we omit the results with static error factor. Interestingly, in a heterogeneous system the dynamic policies stay stable even at 99% imposed utilization with correct runtime estimates. Even though the average service rate of the heterogeneous system is the same as of the homogeneous system, the stream of arriving workflows does not split equally between two processor groups. There are two reasons for this: all the considered policies give priority to faster processors, and faster processors more often lead to scheduler invocations as they simply capable to process tasks faster. Compared to the homogeneous environment, the mean number of workflows in the heterogeneous system during the experiment without an error is higher and ranges from 38 (CPP) to 43 (HR).

For comparability with the results in Section V-A, in Figures 5 and 6 we show the results for the heterogeneous system at 98% utilization. Some policies, e.g., OWM, FDWS, and HR destabilize at certain error factors even more often as in the homogeneous system. However, similarly to the homogeneous system, all the considered dynamic policies are stable at 97% utilization for all the error factors. Comparing Figures 5 and 3 we can see that for all the policies their mean slowdowns increase in the heterogeneous system. At the same time, the values of standard deviation stay comparable to the homogeneous system, and only OWM perform much poorer and even destabilizes at error factor 0.1. The reason is that

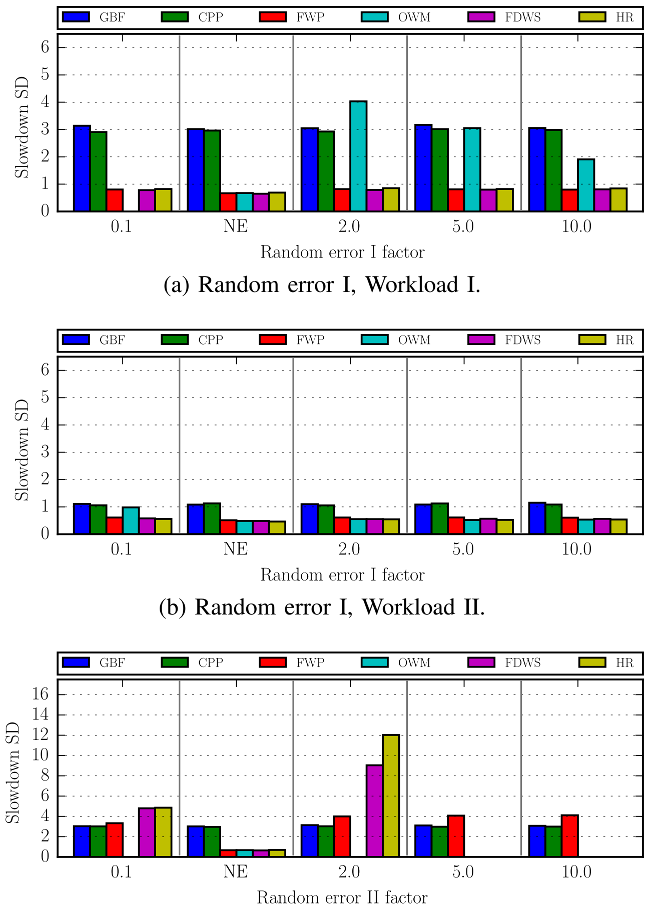


(a) Random error I, Workload I.

(b) Random error I, Workload II.

(c) Random error II, Workload I.

Figure 5: Slowdown versus the error factor at 98% system utilization in a heterogeneous system. NE is No Error, means are marked with ×. Missing bars indicate unstable situations.



(a) Random error I, Workload I.

(b) Random error I, Workload II.

(c) Random error II, Workload I. The vertical axis has larger scale.

Figure 6: Slowdown standard deviation versus the error factor at 98% system utilization in a heterogeneous system. NE is No Error. Missing bars indicate unstable situations.

OWM postpones tasks if there exist better placement in the future on a faster processor, and, of course, it is only applicable to the heterogeneous system. However, when task runtimes estimates are incorrect, OWM starts to make "mistakes" by unnecessarily postponing more tasks. Similar but even worse behavior can be observed in Figures 5c and 6c with Workload I and Random error II where OWM destabilizes for any error factor.

As in Section V-A, all the policies stay stable with Workload II and only exhibit slightly higher mean slowdowns. Only in Figure 6b OWM shows an increase of standard deviation, however, without destabilizing. We do not present results for Workload II with Random error II as they are almost identical to Figures 5b and 6b with the only difference that OWM does not increase the standard deviation at error factor 0.1 and stays in line with FWP, FDWS, and HR.

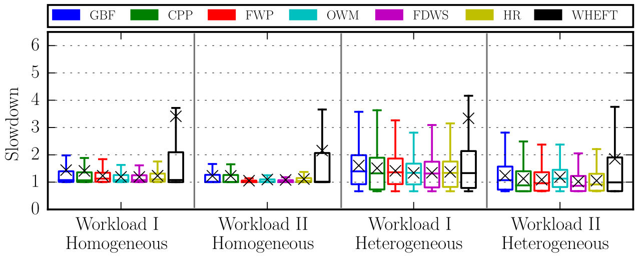
Our FWP policy shows comparable performance to FDWS and HR while showing lower slowdown variability with Random error II and Workload I (Figure 6c) as FDWS and HR simply destabilize. Moreover, CPP policy performs better than GBF, showing that prioritizing tasks with higher upward rank has more effect in a heterogeneous system.

C. *Performance of Plan-based WHEFT*

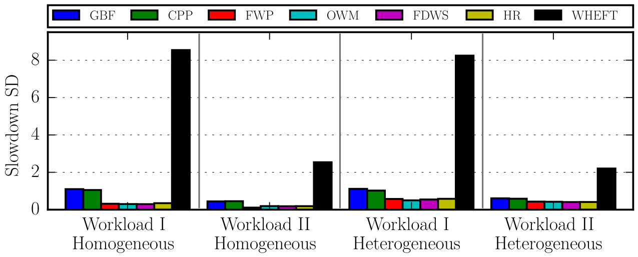
We include only a limited set of results with WHEFT as it simply turns out to be ineffective with workloads of workflows and brings extra complexity by requiring plan construction. Figure 7 shows a performance comparison of WHEFT with the dynamic policies at an imposed system utilization of 97%, as this is the maximum utilization at which WHEFT is stable in the No Error scenario. As a first conclusion from Figure 7, we find that a lower utilization decreases slowdowns and reduces the difference between (the interquartile ranges of) the dynamic policies compared to the results in Figures 3 and 5.

WHEFT is unstable for both error types and all error factors at 97% utilization. We investigated at which utilizations WHEFT stabilizes by decreasing the system utilization with steps of 10% in the presence of task runtime estimation errors. It turns out that WHEFT is only stable for all the considered error factors at a very low utilization of 40%.

The reason why WHEFT is so sensitive to estimation errors is that between workflow arrivals it is completely plan-driven and thus has less flexibility to cope with incorrect runtime estimates. If according to the plan a certain task should be currently scheduled to a processor, but it is not



(a) Slowdown in various system configurations.



(b) Standard deviation of slowdown in various system configurations.

Figure 7: The performance of WHEFT in comparison with the dynamic policies at 97% imposed system utilization without estimation errors. Means are marked with ×.

eligible due to the incorrectly calculated plan, WHEFT just skips it, leaving the processor idle. It thus creates a gap in the schedule and slows down the workflow to which the task belongs. Once a task is wrongly placed in the plan due to incorrect estimates, it can only possibly be relocated to a better position when a new workflow arrives. While WHEFT postpones tasks "unintentionally", OWM postpones tasks on purpose in the hope that finally they will be scheduled on a faster processor. So the reason why WHEFT shows poor results is similar to why OWM becomes unstable in Section V-B.

At a 97% utilization, our simulated system receives 97 workflows per hour (since the workflows have an average total execution time of 1 hour), which means that on average, the plan is recomputed 97 times per hour. So on average, every workflow task has a chance to be relocated to a better position 97 times during the workflow execution. Decreasing the utilization decreases the number of simultaneously scheduled workflows, but at the same time it decreases the number of possible task relocations.

Moreover, the original HEFT policy schedules tasks with higher upward ranks first. For this reason, WHEFT gives priority to longer workflows constituting the joint workflow. Accordingly, WHEFT postpones shorter workflows, which represent the majority of both our workloads. That increases the average slowdown and accumulates more workflows in the system, finally destabilizing it.

For plan-based policies in real-world non-simulated environments, the duration of the plan construction phase is crucial. Newly arrived workflows cannot start their execution until their tasks have been added to the plan. This can additionally increase job slowdown. In the considered simulated environment, from the perspective of workflows plan construction takes zero time, but in real systems it should be much smaller than the average workflow inter-arrival time. In our case, for 1000 simultaneously running workflows (see Section V-D) the plan construction takes 40 minutes for a Python3 implementation running on a DAS-4[1] node (2.4 GHz Intel E5620 CPU, 24 GB RAM). The planning time, however, can be reduced by using a tree structure (e.g., a k-tree [31]) to store the information about the gaps in the plan. It will decrease the time required to find an appropriate gap for a task at the cost of a higher memory consumption.

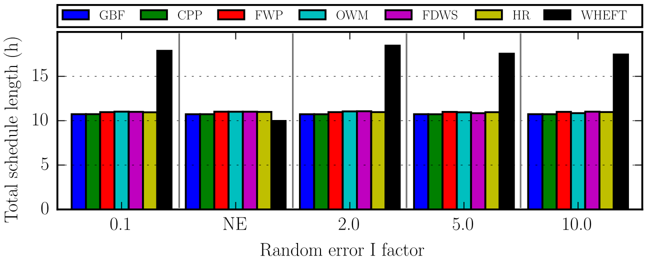


Figure 8: The total schedule length in hours of a batch submission based on Workload I with 1000 workflows in a homogeneous system. NE stands for No Error.

D. Performance of a Batch Submission

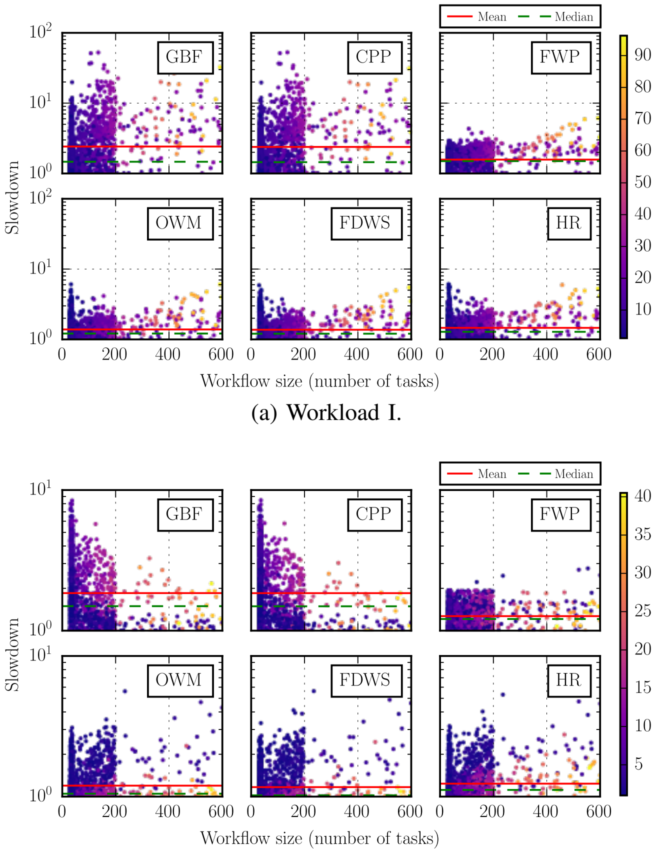
In this section we investigate how the considered policies behave when handling a batch submission. Since this paper mainly focuses on the analysis of workloads of workflows, we only perform a limited set of experiments with a single batch submission of 1000 workflows based on Workload I in a homogeneous system. Figure 8 shows the total schedule length in hours of this batch with variable Random error I. We do not show the results for Random error II as they are comparable. There is a small difference for under- and over-estimating situations, with GBF and CPP producing schedules that are longer by half an hour (5.5%) than the other dynamic policies.

We do not report slowdowns, as we suppose that batch submissions usually come from a single user who is only interested in minimizing the total schedule length rather than achieving fairness among the workflows in the batch. We can clearly see that WHEFT, indeed, constructs a shorter schedule than the dynamic policies. However, it is less resilient to errors as its scheduler postpones tasks which are not eligible or if a target processor is occupied (as in Section V-C). The schedule length created by WHEFT matches the expected length of 10 hours, as we scheduled 1000 workflows with an average total execution time of 1 hour on 100 processors. Surprisingly, GBF and CPP policies produce shorter schedules than the other dynamic policies. Note, that GBF and CPP process workflows in the order in which they are defined in the batch, while all the other policies use various ways of ranking to prioritize workflows.

E. Fairness

To demonstrate that our FWP policy allows to achieve better fairness when scheduling workloads of workflows, in Figure 9 we show scatter plots when running both workloads at 98% utilization in a homogeneous system without runtime estimation errors. We can clearly see that FWP reduces the

[1] Distributed ASCI Supercomputer 4, www.cs.vu.nl/das4



(a) Workload I.

(b) Workload II.

Figure 9: Scatter plots of slowdowns versus the sizes of workflows at 98% utilization in a homogeneous system without an error. The glyph color encodes the approximated LoP, the vertical axes have a log scale.

number of outliers and moves the median slowdown closer to the mean than any other policy, while slightly increasing the mean. Similar behaviour is observed in the heterogeneous system (not shown).

Obviously, Workload I is more challenging for the dynamic policies as it contains more highly parallel workflows (see Figure 1). From one perspective, containing more tasks those workflows have more chances to adjust their ranks during the execution. From another perspective, among those highly parallel workflows some have very short critical paths, which means that their slowdowns, in case of any delay, increase much faster compared to relatively sequential workflows with long critical paths. These "short" but highly parallel workflows are the main reason why FWP does not show even better results with Workload I. None of the considered dynamic policies is able to completely remove such outliers. Including the level of parallelism when computing the rank in FWP might help to solve this problem.

VI. Related Work

Our work is a first study in the field which considers the influence of task runtime estimates on the quality of scheduling for a variety of workflow scheduling heuristics.

Yu ans Shi [8] use Poisson arrivals, but suppose perfect runtime estimates, and do not investigate slowdown variability. Hsu et al. [13] propose the original OWM algorithm and also use Poisson arrivals with a set of experiments dedicated to the impact of inaccurate runtime estimates. Unlike us, that paper only considers one type of random uncertainty and compares OWM only with two other algorithms, including Rank_Hybd (HR) which we implement in this paper. Moreover, the number of scheduled workflows in that paper is only 100, and the system utilization and stability are not taken into account.

In a paper by Arabnejad and Barbosa [32] the authors compare HEFT with FDWS and show that HEFT exhibits the poorest performance. They claim that they modified the original HEFT to use it in an online scenario, but do not clearly explain how. Moreover, the authors do not consider system utilization, just simply submitting relatively few workflows (50) with a fixed interval. In the recent paper by Arabnejad and Barbosa [33] which targets multi-QoS constrains the system utilization is not considered either.

The slowdown-based fairness problem has been addressed before. Zhao and Sakellariou [15] proposed a plan-based policy which targets fairness using a variety of approaches. However, their algorithm has limited applicability for workloads of workflows as it is plan-based. Recently, Wang et al. [34] proposed fairness-aware dynamic FSDP algorithm which, however, does not clearly link the current slowdown and the target average slowdown which should be achieved, and recomputes workflow priorities on every new workflow arrival only. In contrast, our FWP policy recomputes priorities when a task becomes eligible or a processor becomes idle. An algorithm for fairness and granularity control for online scheduling of workflows is also addressed in a paper by Ferreira da Silva et al. [35]. Their approach, though, does not consider system utilization.

Among the algorithms which can operate without task runtime estimates we distinguish PRIO [36] which was successfully implemented in the DAGMan component of the Condor distributed job scheduler [37]. However, we do not include it in the comparison, as it tries to maximize the number of eligible tasks in hope to increase the throughput of the system, while in this paper we focus on upward rank-based policies.

VII. Conclusions and Future Work

In this paper, we have investigated the effect of incorrect task runtime estimates on the performance of dynamic and plan-based scheduling policies in the online scenario of scheduling workloads of workflows.

We can clearly see the benefit of knowing task runtime estimates as we do observe significant performance differences between the considered dynamic policies and large improvements in average job slowdown, but only at extremely high system utilizations. Similarly, the sensitivity to incorrect task runtime estimates increases at higher system utilizations. The order in which the workflows are processed is very important, as it allows to achieve a fairer distribution of slowdowns among workflows in the workload, as in our FWP policy.

Giving priority to workflows with longer critical paths, especially at extremely high utilizations, easily destabilizes the system if the workload has a majority of short workflows, as these short workflows start to accumulate. At lower utilizations, which are very common in real datacenters, simpler backfilling-based policies that do not use task runtime estimates are quite applicable and show comparable performance to more advanced fairness-oriented policies.

The plan-based WHEFT policy shows poor performance with workloads of workflows, but it does construct the shortest schedule for batch submissions. Moreover, WHEFT is quite unstable with incorrect task runtime estimates, running stably only at the relatively low utilization of 40%. We believe that even more complex policies like Hybrid.BMCT [11] would also suffer from this problem. Even though we do not exclude that plan-based approaches could achieve slowdowns comparable to those of dynamic policies, their planning overhead and implementation complexity do not seem to be worth it.

For future work, we are going to further improve the performance of the FWP policy by trying other prioritization approaches to allow even short and extremely parallel workflows to experience comparable slowdowns. Since the performance of the policies which rely on task runtime estimates depends on the type of the random error, it would be interesting to use other error types, e.g., giving more error variability to shorter tasks, as has been observed by Feitelson [5]. Moreover, we plan to additionally validate the considered policies in a real cloud environment with more complex submission patterns.

VIII. Acknowledgments

This research is supported by the Dutch national program COMMIT.

References

[1] Ewa Deelman et al. The future of scientific workflows. *The International Journal of High Performance Computing Applications*, 2017.

[2] Gideon Juve et al. Characterizing and profiling scientific workflows. *FGCS*, 29:682–692, 2013.

[3] Ilia Pietri, Gideon Juve, Ewa Deelman, and Rizos Sakellariou. A performance model to estimate execution time of scientific workflows on the cloud. In *IEEE WORKS*, 2014.

[4] Adán Hirales-Carbajal et al. Multiple workflow scheduling strategies with user run time estimates on a grid. *Journal of Grid Computing*, 10:325–346, 2012.

[5] Dror G. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press, 2015. pp. 399-489.

[6] Cynthia Bailey Lee, Yael Schwartzman, Jennifer Hardy, and Allan Snavely. Are user runtime estimates inherently inaccurate? In *JSSPP*, 2004.

[7] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE TPDS*, 13:260–274, 2002.

[8] Zhifeng Yu and Weisong Shi. A planner-guided scheduling strategy for multiple workflow applications. In *IEEE ICPP-W*, 2008.

[9] Hamid Arabnejad and Jorge Barbosa. Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems. In *IEEE ISPA*, 2012.

[10] Hamid Arabnejad, Jorge Barbosa, and Frédéric Suter. Fair resource sharing for dynamic scheduling of workflows on heterogeneous systems. *High-Performance Computing on Complex Environments*, 2014.

[11] Rizos Sakellariou and Henan Zhao. A hybrid heuristic for DAG scheduling on heterogeneous systems. In *IEEE IPDPS*, 2004.

[12] Alexey Ilyushkin, Bogdan Ghit, and Dick Epema. Scheduling workloads of workflows with unknown task runtimes. In *IEEE/ACM CCGrid*, 2015.

[13] Chih-Chiang Hsu, Kuo-Chan Huang, and Feng-Jian Wang. Online scheduling of workflow applications in grid environments. *FGCS*, 27:860–870, 2011.

[14] Artem M. Chirkin et al. Execution time estimation for workflow scheduling. *FGCS*, 2017.

[15] Henan Zhao and Rizos Sakellariou. Scheduling multiple DAGs onto heterogeneous systems. In *IEEE IPDPS*, 2006.

[16] Selma Ikiz and Vijay K. Garg. Online algorithms for Dilworth's chain partition. Technical report, Parallel and Distributed Systems Laboratory, Department of Electrical and Computer Engineering, University of Texas at Austin.

[17] Gideon Juve et al. *Synthetic Workflow Generators*. www.github.com/pegasus-isi/WorkflowGenerator.

[18] Shishir Bharathi et al. Characterization of scientific workflows. In *Third Workshop on Workflows in Support of Large-Scale Science*, 2008.

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

[20] Ian J. Taylor, Ewa Deelman, Dennis Gannon, and Matthew S. Shields. *Workflows for e-Science: Scientific Workflows for Grids*. Springer-Verlag London Ltd, 2007.

[21] Arun Ramakrishnan et al. Scheduling data-intensive workflows onto storage-constrained distributed resources. In *IEEE/ACM CCGrid*, 2007.

[22] Jonathan Livny. Bioinformatic discovery of bacterial regulatory RNAs using SIPHT. *Bacterial Regulatory RNA: Methods and Protocols*, 905:3–14, 2012.

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

[24] Frédéric Suter and Sascha Hunold. *DAGGen: A synthetic task graph generator*. www.github.com/frs69wq/daggen.

[25] Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *JPDC*, 63:1105–1122, 2003.

[26] Simon Ostermann et al. On the characteristics of grid workflows. In *CoreGRID*, 2008.

[27] Alexandru Iosup, Ozan Sonmez, and Dick Epema. DGSim: Comparing grid resource management architectures through trace-based simulation. In *Euro-Par*. 2008.

[28] Alexandru Iosup et al. Inter-operating grids through delegated matchmaking. In *ACM/IEEE Supercomputing*, 2007.

[29] Jamie R. Wieland, Raghu Pasupathy, and Bruce W. Schmeiser. Queueing-network stability: Simulation-based checking. In *Winter Simulation Conference*, 2003.

[30] Michael J. Neely, Eytan Modiano, and Chih-Ping Li. Fairness and optimal stochastic control for heterogeneous networks. *IEEE/ACM Transactions on Networking*, 16:396–409, 2008.

[31] H.P. Patil. On the structure of k-trees. *Journal of Combinatorics, Information and System Sciences*, 11:57–64, 1986.

[32] Hamid Arabnejad and Jorge G. Barbosa. Multi-workflow QoS-constrained scheduling for utility computing. In *IEEE CSE*, 2015.

[33] Hamid Arabnejad and Jorge G. Barbosa. Multi-QoS constrained and profit-aware scheduling approach for concurrent workflows on heterogeneous systems. *FGCS*, 68:211–221, 2017.

[34] Yuxin Wang et al. Fairness scheduling with dynamic priority for multi workflow on heterogeneous systems. In *IEEE ICCCBDA*, 2017.

[35] Rafael Ferreira da Silva, Tristan Glatard, and Frédéric Desprez. Controlling fairness and task granularity in distributed, online, non-clairvoyant workflow executions. *Concurrency and computation: practice and experience*, 26:2347–2366, 2014.

[36] Grzegorz Malewicz, Ian Foster, Arnold L. Rosenberg, and Michael Wilde. A tool for prioritizing DAGMa jobs and its evaluation. *Journal of Grid Computing*, 5:197–212, 2007.

[37] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency and computation: practice and experience*, 17:323–356, 2005.