An Exploration of Online-simulation-driven Portfolio Scheduling in Workflow Management Systems

Jesse McDonald[a,*], John Dobbs[a], Yick Ching Wong[a], Rafael Ferreira da Silva[b], Henri Casanova[a]

[a] *University of Hawai‘i at Mānoa, Information and Computer Sciences Dept., Honolulu, HI, USA*
[b] *National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN, USA*

Abstract

Workflow Management Systems used to automate the execution of scientific workflow applications on parallel and distributed computing platforms must make scheduling decisions at runtime. A large number of workflow scheduling algorithms have been proposed in the literature, but often these algorithms are evaluated based on simplifying assumptions that may not hold in practice. Furthermore, published algorithm evaluation and/or comparison results are necessarily only for a subset of all possible scenarios, and thus may not include scenarios relevant to particular use-cases. Consequently, it is difficult for Workflow Management Systems (WMSs) developers to decide which scheduling algorithm should be implemented. To obviate this difficulty, one possible approach is to implement a portfolio of scheduling algorithms and select the most effective algorithm at runtime. One method for performing this selection is to run an online simulation for each algorithm in the portfolio. The algorithm that leads to the best performance, in simulation, is selected for future use.

The above simulation-driven portfolio scheduling (SDPS) approach has been proposed in a few parallel and distributed computing contexts. The main objective of this work is to evaluate the feasibility and potential merit of SDPS if implemented in WMSs. We perform this evaluation using simulated WMS executions, where the simulations are instantiated from real-world platform and workflow configurations. Our main finding is that SDPS is on par with or outperforms an approach in which a single algorithm is used, where this algorithm is the one that performs best on average across all our experimental scenarios. Furthermore, we find that SDPS remains an attractive proposition even in the presence of high levels of simulation error and for simulators with relatively low levels of sophistication. In many of our experimental scenarios we find that mitigating simulation error at runtime can further improve performance. Finally, we show that simulation overhead can be made sufficiently low for SDPS to be feasible in practice.

Keywords: Scientific Workflows, Workflow Management Systems, Portfolio Scheduling, Simulation

1. Introduction

Scientific workflow applications have been used by computational scientists to support some of the most significant discoveries of the past several decades [1], and are executed daily to serve a wealth of scientific domains. Many workflows have high computational demands and are executed in production on platforms that range from single HPC clusters to federations of such clusters and clouds. Setting up, orchestrating, monitoring, and optimizing workflow executions on these platforms is managed by runtime systems often called Workflow Management Systems (WMSs) [2, 3]. These systems automate application execution including (i) selecting hardware resources; (ii) picking application configuration options (e.g., numbers of cores to be used by multi-threaded tasks); (iii) allocating resources to application activities; and (iv) monitoring application executions. It is widely accepted that the scheduling decisions should be made by using appropriate *scheduling algorithms* so as to optimize performance, monetary cost, energy consumption, reliability, etc. The past decade has witnessed a proliferation of WMSs [4], but there is no consensus on which scheduling algorithms should be implemented in these systems.

Scheduling problems are generally NP-hard, and thus most proposed algorithms employ non-guaranteed heuristics. The design of scheduling algorithms for workflow applications has received an enormous amount of attention [5–13]. Most of the proposed algorithms reuse ideas and principles from the extensive DAG (Directed Acyclic Graph) scheduling literature [14]. Yet, when examining existing WMSs, there is a clear disconnect between research and practice. Given the complexity of hardware platforms on which workloads are to be executed and

[*]Corresponding address: University of Hawai‘i at Mānoa, Information and Computer Sciences Department, POST Building, Rm 317, 1680 East-West Road, Honolulu, HI 96822, USA

[**]This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The publisher acknowledges the US government license to provide public access under the DOE Public Access Plan (`http://energy.gov/downloads/doe-public-access-plan`).

Email addresses: `jamcd@hawaii.edu` (Jesse McDonald), `jmd386@hawaii.edu` (John Dobbs), `wongy@hawaii.edu` (Yick Ching Wong), `silvarf@ornl.gov` (
Rafael Ferreira da Silva), `henric@hawaii.edu` (Henri Casanova)

the complexity of workflow applications themselves, scheduling research results are typically obtained relying on simplifying assumptions so that scheduling problems are rendered more formalizable and tractable. Furthermore, published evaluation results cannot cover all relevant situations a WMS could encounter in practice. The literature is thus rife with scheduling algorithms that have been evaluated within the scope of their underlying assumptions, but whose potential effectiveness in practice for particular use-cases remains unquantified. Given the above, there is little incentive for developers of WMSs to pay close attention to scheduling research. Our own observation of current and popular WMSs has shown that often naive, and thus possibly vastly suboptimal, scheduling algorithms are implemented, such as simple greedy algorithms.

A way to resolve the disconnect between scheduling research and scheduling practice is to obviate the challenge of picking a particular scheduling algorithm to implement in a WMS. Instead, one can implement a set of scheduling algorithms, estimate how each algorithm would fare for the particular use-case at hand at runtime, and select the most desirable algorithm. This approach has been termed "portfolio scheduling" [15]. In this work, we consider WMSs that automate the execution of workflow applications that perform I/O, communication, and computation operations that use and contend for various hardware resources. In this context, determining how a scheduling algorithm would perform using an analytical model to compute application execution metrics is challenging, due to the need to account for complex phenomena such as disk bandwidth contention, network sharing, network protocol effects, overlap between computation, I/O, and network communication activities, etc. An alternative is to use *discrete-event simulation* by which the behavior of a particular workflow application when executed on a particular hardware platform emerges from simulated discrete events. Relevant metrics can then be easily computed from the produced event trace. This does not, however, mean that such a simulation is necessarily perfectly (or even moderately) accurate.

In this work we make the following contributions:

- We propose to use simulation-driven portfolio scheduling (SDPS) as part of Workflow Management Systems;
- We investigate the following research questions (RQ): RQ#1: What is the potential improvement of SDPS over the traditional one-algorithm approach in which a single scheduling algorithm is used throughout the execution? RQ#2: What level of simulation error is tolerable? RQ#3: Is it useful to mitigate simulation error at runtime? RQ#4: What is the impact of the sophistication of the simulation? RQ#5: Is SDPS effective for handling dynamic resource availability? RQ#6: Is simulation overhead sufficiently low for SDPS to be practical?;
- We develop an experimental methodology to answer the above questions, which entails implementing SDPS as a part of a simulated WMS that can quickly simulate the execution of arbitrary workflow specification on arbitrary multi-cluster platform specifications.
- We find that SDPS can be on par with, and often outperform, the one-algorithm approach even in the presence of significant simulation inaccuracy.

A shorter version of this work appeared in [16]. This works improves on the experimental methodology, summarizes and/or complements findings for RQ#1, RQ#2 and RQ#6, and answers questions RQ#3, RQ#4 and RQ#5 which were not considered in [16]. Section 2 discusses related work. Section 3 describes SDPS, which we evaluate via the experimental methods described in Section 4. Section 5 presents experimental results. Finally, Section 6 summarizes our contributions and discusses future work.

2. Related Work

2.1. Portfolio Scheduling

Several authors have proposed to select, at runtime, a scheduling algorithm or the parameter values that define a particular instantiation of a scheduling algorithm. Approaches for performing this selection have been investigated in various parallel and distributed computing domains and are often termed "portfolio scheduling". The key question is that of the mechanism that should be employed to select the best algorithm among those included in the portfolio. Authors have proposed different answers to this question by using one of three approaches, which we review in the next sections: online performance monitoring, offline simulations, or online simulations.

2.1.1. Online Performance Monitoring

Some previously proposed approaches perform algorithm selection via *online performance monitoring* of the algorithms in the portfolio. In [17] the authors use such an approach in the context of batch scheduling for HPC clusters to dynamically select one among 12 different batch queue reordering policies. Their approach keeps track of how well each policy has fared over previous time windows in terms of job wait times, and selects the policy for the next time window using reinforcement learning. Specifically, an ϵ-greedy strategy is employed to achieve both exploitation (select with high probability the policy that has worked best in the past) and exploration (select with low probability a policy at random). Reinforcement learning is also used in [18] for selecting one among 10 loop scheduling algorithms in the context of iterative distributed-memory parallel applications. The authors define a metric that captures the robustness of a scheduling algorithm to variations in the execution time of the loop iteration. Using this metric as a reward they then evaluate the effectiveness of several reinforcement learning techniques for selecting the loop scheduling algorithm. In [19], the authors propose an approach for selecting one among 12 scheduling algorithms in the context of OpenMP parallel loops. Based on online monitoring of the load imbalance after each iteration they propose several methods for selecting the algorithm to use for the next iteration, including a method based on fuzzy logic. These works are for different application contexts but they all rely on the same assumption: observing how algorithms have fared over time windows or application steps in the near past makes it possible to determine which algorithm is best for the near future. This is clearly a reasonable assumption for the iterative applications considered in [18, 19]. In [17] this assumption is more questionable since batch scheduling decisions can have long-term implications, which explains why the results therein show that a simulation-based approach leads to better results. In this work we focus on workflow applications for which it is difficult to assess the effectiveness of a scheduling algorithm based on its past performance. The particular task graph structure of a workflow can cause early scheduling decisions to have significant long-term (positive or negative) impact on the overall execution time. As a result, and as observed in [16], although a particular scheduling algorithm may seem efficient (e.g., in terms of amount of computational work performed per time unit) during an early phase of the execution, other algorithms that are similarly or less efficient during that phase can lead to significantly shorter execution times overall. This provides a clear motivation in this work to use simulation to assess future algorithm performance instead of online monitoring of previously observed performance.

2.1.2. Offline Simulations

Several authors have proposed using results from *offline simulations* to drive portfolio scheduling decisions at runtime. In [20], the authors focus on batch scheduling for HPC clusters and study the problem of picking the batch queue ordering policy at runtime. They generate large numbers of synthetic workload datasets and run offline simulations of their executions for permutations of the job scheduling order. The goal is to determine a score that, based on a job's characteristics, quantifies the impact of scheduling that job first. They then use a machine learning technique (nonlinear regression) to derive a model of that score. At runtime, the queue ordering policy is decided by predicting the score for each job in the queue and picking the job with the best predicted score. Also in the context of batch scheduling for HPC clusters, in [21] the authors conduct offline simulations of the execution of several real-world workload datasets using two different queue ordering schemes. The simulation results show that the average job parallelism can be used to decide on which scheme to select at runtime. The proposed approach consists in defining a time window length, computing the average job parallelism over that time window, and using the computed value to pick which of the two queue ordering schemes should be used for the next time window. In [22], the authors propose to use a supervised learning approach for selecting which dynamic loop scheduling algorithm to use at runtime. The training dataset consists of simulated executions for various problem sizes, hardware platform specifications, and workload and resource variability ranges. At runtime, the trained model can then be used to decide which algorithm in the portfolio (which in the presented results consists of eight algorithms) should be selected. The main challenge for these approaches is to determine what kind and what amount of training data should be used to obtain good results in practice. In some cases, the setting is simple enough that only a few parameters are to be considered (e.g., 4 parameters in [22]). It is then reasonable to expect that even a relatively small training set can produce good results. But this is not always the case. For instance, in the context of batch-scheduling, there are many kinds of possible workloads with different distributions of job arrival times, levels of parallelism, requested durations, actual durations, and correlations between all these features. It is thus likely that in [20, 21] better results could be achieved with larger and/or more diverse datasets.

This work focuses on portfolio scheduling to be implemented in Workflow Management Systems. These systems must support applications that have a large space of possible configurations in terms of task graph structures (number of tasks, number of levels, distribution of parallelism across these levels, density of dependency edges), task compute volumes, and data compute volumes; and execute these applications on platforms that also have a large space of possible configurations in terms of scale, heterogeneity, network topologies, and storage architectures. As a result, only a very large training dataset could potentially lead to good results. But obtaining a sufficiently large real-world dataset is prohibitively difficult in the general case. An alternate approach could be to generate a synthetic dataset from large numbers of offline simulations. In this work we employ an online simulation approach instead.

2.1.3. Online Simulations

Using online simulation for portfolio scheduling can be seen as a compromise between online performance monitoring and offline simulation approaches. Like the former it does not require any a-priori model of which algorithms in the portfolio will perform well in particular situations, and like the latter it uses simulation to capture long-term impact of scheduling decisions. In [23] the authors propose to simulate periodically at runtime the execution of iterative parallel applications when using one of 13 possible dynamic loop scheduling algorithms, and pick the best algorithm to use for the next time period. Their main objective is to adapt to changing platform conditions. The key difference between this work and that in [23], besides the targeted application domain, is that the authors therein assume that simulation error is low (below 2%). This assumption makes sense in their context due to the focus on compute-intensive applications for which the influence of possibly complex memory hierarchies, network protocols and network topologies are small. In this work, instead, we consider the execution of distributed applications for which network and I/O usage can drive the overall performance. As a result, simulation error can be higher since simulating complex network and I/O behaviors accurately is challenging. Furthermore, unlike in [23], and as explained at the end of Section 2.1.1, we must simulate the application execution until its completion, meaning that simulation errors are likely to accumulate. For this reason, in this work we consider simulation error up to 100%, and one of our key results is that portfolio scheduling is feasible even with high simulation error.

Most previous works that have employed online simulation for portfolio scheduling are in the context of batch scheduling for HPC clusters. The works in [15, 17, 21, 24, 25] all propose similar approaches by which a portfolio of standard batch scheduling algorithms is considered. At runtime, while scheduling is done using a particular scheduling algorithm during the current time period, all other algorithms are also used to also compute the schedule during that time period. At the end of the time period, the schedules produced by all algorithms are compared using metrics such as average job wait time or bounded slowdown, and the best performing algorithm is selected for the next time period. The work in [26] proposes to use online simulation to tune the parameters of a single batch scheduling algorithm at runtime, using genetic algorithms to determine optimal parameter values. The fitness function evaluation entails running simulations of the recently executed workload. In all these works, because jobs are submitted to the HPC cluster by users, at the time the next algorithm must be chosen the future workload is unknown. Therefore, online simulations are used to simulate the past workload, assuming that it is representative of the future workload. In this work we use online simulation to simulate the future workload. This is because the future workload is known (as the workflow application is fully specified) and because, as explained at the end of Section 2.1.1, the performance of a scheduling algorithm in earlier phases of the execution may not be indicative of its performance overall. Another major difference between this work and [15, 17, 21, 24–26] is that in those works the "simulation" is not a discrete-event simulation but is instead merely the execution of each algorithm to compute a schedule. That is, there is no actual simulation of the hardware platform, but merely a computation of a Gantt chart of the schedule based on job sizes (number of compute nodes) and durations (requested by the user). The only source of inaccuracy in this computation is the job durations requested by the users. This does not, however, correspond to the standard notion of simulation inaccuracy, which stems from the fact that the simulation is only an approximating of a real-world system. Instead, the inaccuracy in these works is in the input to the simulation, which is no different from the inaccuracy of the input to the real system. The one exception among the works cited above is [17], in which after the schedule has been computed so as to estimate the wait time of all the jobs, then a random uniformly distributed noise (up to 20%) is added to each job wait time. In this work we also inject random noise to experiment with various levels of simulation accuracy. In our context, unlike in the HPC context, simulation inaccuracies arise because the simulation cannot perfectly capture the behavior of a complex system in which an application workload performs communication, I/O, and computation activities that use and contend for distributed hardware resources. Furthermore, information on the current state of the execution, on the platform configuration, and on the application's behavior, which are all needed to instantiate a simulation, is not perfect. As a result, simulation models are inherently biased and simulation inaccuracy can be large. For this reason, and unlike all the aforementioned works, we also investigate the possibility of mitigating simulation error at runtime.

The authors in [27] use online simulation for the purpose of workflow scheduling. The authors propose to use simulation to improve the decisions made by a single scheduling algorithm. A workflow scheduling algorithm makes decisions regarding which task should be scheduled next based on task finish time estimates. These estimates are traditionally obtained based on analytical models of task compute, I/O, and/or communication times. Given the complexity of current platforms, developing accurate analytical models is a steep challenge, especially if needing to account for contention and network protocol effects. The authors in [27] propose to obtain network transfer time estimates at runtime using discrete-event simulation, thereby developing in essence a simulation-enhanced scheduling algorithm. A major difference with this work is that the purpose is not to perform portfolio scheduling. But both this work and that in [27] share a key motivation for using simulation for scheduling workflows on distributed computing platforms: the ability afforded by simulation to estimate network (and in our case also I/O) times more accurately than would be possible with only analytical models.

Finally, note that the idea of driving scheduling decisions using online simulations has been explored by authors outside the field of parallel and distributed computing, such as for manufacturing and logistics (e.g., to solve versions of the job shop scheduling problem for cyber-physical systems). A cursory review of that literature shows that simulation-driven scheduling has been proposed and used since at least the early 1990's. To take just one recent example, the work in [28] proposes simulation-driven scheduling in the context of wind farms operation.

2.2. Simulation of parallel and distributed systems

Many simulation frameworks have been developed for parallel and distributed computing domains [29–39]. The objective of these frameworks is to ease the development of simulators in these domains. To do so they implement and provide convenient interface to discrete-event simulation models. The particular implementations of these models in each framework achieve different compromises between accuracy and speed. At one extreme are fine-grained models that capture "microscopic" behaviors of hardware/software systems (e.g., packet-level network simulation, block-level disk simulation, cycle-accurate CPU simulation), which favor accuracy over speed. At the other extreme are coarse-grained models that capture "macroscopic" behaviors via mathematical formulations. While these models lead to fast simulation, they must be developed carefully if high levels of accuracy are to be achieved [40]. This work is completely agnostic to the user's implementation of the simulator. As part of our experimental methodology, described in Section 4, we implement a simulator using the SimGrid [37] and WRENCH [39] frameworks. We selected SimGrid because it is has a large user base (it has been used to obtain simulation results for over 630 research publications to date [41]), because its simulation models have been extensively validated, and because these models are macroscopic and thus enable fast simulations of large platforms and applications. WRENCH builds on SimGrid to provide high-level simulation abstractions that make it possible to implement simulators of complex systems, including WMSs that execute workflows on distributed computing platforms, in only a few hundred lines of code, especially for simulators of workflow executions [39].

3. Proposed Approach

Consider a distributed computing platform with hardware resources (compute, storage, network) accessible via various software services for starting computations, storing data, and moving data. A runtime system is used to automate the execution of some application workload, for which it must make decisions regarding the allocation of application activities to the hardware resources in time and space. The goal is to optimize user-defined metrics of performance. In this work we focus on Workflow Management Systems (WMSs) that are runtimes systems for executing workflow applications.

We propose simulation-driven portfolio scheduling (SDPS) by which the application execution is simulated at runtime for each scheduling algorithm in a portfolio. A description of the application and of the available hardware resources is constructed based on (likely imperfect) available information, so as to instantiate a simulator of the upcoming application execution. Simulations are then executed for each algorithm in the portfolio, assuming in each case that the considered algorithm is used until application completion. These simulations are executed on the host on which the runtime system itself executes (typically some multi-core host that orchestrates the application execution on other "remote" resources). Each simulation produces a trace of time-stamped simulated discrete events, from which execution metrics can be computed. The scheduling algorithm that achieves the best such metrics (in simulation) is then selected and used for making scheduling decisions at runtime from then on.

In what follows we discuss implementation considerations, discuss when SDPS should be used at runtime, and state the specific research questions that we investigate in this work.

3.1. Implementation Considerations

For a WMS to implement SDPS it needs to: (i) make it possible to implement and activate different scheduling algorithms; (ii) have access to information regarding the platform, the application and the current state of the execution to instantiate a reasonable (but not necessarily perfectly accurate) simulation of the remainder of the execution; and (iii) include an implementation of a simulator of this execution instantiated based the available information. We discuss these three requirements in the next three sections.

3.1.1. Scheduling algorithms

All WMSs provide ways for users to specify workflow tasks and task control- and data-dependencies. Information provided to the WMS about each task can include resource requirements (in terms of compute cores, memory, and disk space), amounts of CPU and I/O work, and/or traces from previous executions. To execute a workflow WMSs are configured with or can discover the resources provided by the platform on which the workflow is to be executed. At runtime, WMSs keep track of which tasks are ready, running, or not ready, and can also keep track of where application data is stored. All the above information can be used to automate resource management decisions and to implement scheduling algorithms (the more information is available the larger the number of algorithms that can conceivably be implemented). Hundreds of WMSs have been developed over the last decades [4] and a comprehensive survey is outside the scope of this work. Our own observation of some current and popular WMSs has shown that often only baseline naive scheduling algorithms are implemented (such as random greedy algorithms). However, in some cases scheduling algorithms can be implemented by users as external plug-ins or modules (e.g., StreamFlow [42], Dask [43]). In some other open-source projects, we have determined that replacing the default scheduling algorithm would be straightforward (e.g., Parsl [44], Pegasus [45]). Consequently, in these WMSs it is feasible to implement a portfolio of algorithms. SDPS can then be implemented as a new scheduling algorithm that, whenever invoked by the WMS, either invokes the currently selected algorithm from the portfolio or selects a new algorithm and invokes it.

3.1.2. Information to instantiate a simulation

Instantiating a simulation of a workflow execution requires a workflow description, an execution platform specification, and the current state of the execution. As explained above, most WMSs are configured with and/or maintain such information. At a minimum, the information necessary to instantiate a simulation for SDPS includes: task dependencies, estimates of CPU and I/O work for each task, hardware specifications or benchmark results for compute, network, and I/O resources, set of completed, ongoing, and to-be-executed tasks, locations of application data file replicas. The challenge is that this information is never perfectly accurate nor complete. Particular values may not be perfectly known and only rough estimates may be available (e.g., CPU speeds, disk bandwidths). Structural information could also be incomplete. For instance, the precise network topology of the platform is typically not known to the WMS. Instead, the WMS may only be provided with information regarding network interface bandwidth hardware specifications, or previously observed data transfer rates on that topology. In this case, a simulation will necessarily abstract away the real-world network topology, for instance as a single network link with some well-chosen bandwidth. The challenge of simulation instantiation is well-recognized and can be alleviated by performing simulator calibration based on ground-truth data [46]. That is, high simulation accuracy can be achieved by computing a simulation instantiation that minimizes simulation error with respect to logs of previous and representative workflow executions. In this work we do not assume high accuracy and instead evaluate the usefulness of SDPS in the presence of simulation inaccuracy.

3.1.3. Simulator implementation

The onus of implementing and maintaining the simulator used for SDPS falls on the WMS developers as only they have all the necessary knowledge about the functional behavior of their system. Developing a simulator of distributed computing systems, applications, and platforms from scratch is a tall order. For this reason, simulation frameworks have been developed that make it possible to implement such simulators relatively easily (see Section 2.2). The simulator can be provided to the WMS as a stand-alone program or implemented directly in the WMS using the simulation framework's API. In this work we use WRENCH [39], which was originally designed for the specific purpose of simulating WMSs. As a result, the core of the simulator used in this work, which corresponds to what a WMS developer would have to implement, is under 300 lines of C++ code (code available on GitHub [47]).

3.2. *When to apply* SDPS

SDPS necessarily executes a round of simulation at the onset of the application to pick a scheduling algorithm, but other rounds of simulations could be conducted throughout the execution. One reason for doing so is to handle dynamic behavior of the workflow or the platform, which may make different scheduling algorithms better suited at different times.

Dynamic workflow changes occur when new tasks are created at runtime based on the output generated by other tasks. Note that in our previous work [16] we have found that, even for static workflows, using more than one algorithm during the workflow execution can sometimes bring marginal benefits. This is because different workflow levels can have different characteristics (e.g., different ratios of data to compute volumes) and can be better served by different scheduling algorithms. Dynamic platform changes occur for shared platforms on which resources that can be used for executing the workflow exhibit variable performance or availability (e.g., due to external load, to resources being reclaimed or released by a provider, to resources being acquired by the user). To handle these dynamic changes, rounds of simulations could be performed continuously throughout the execution (in the extreme at each scheduling decision) provided the frequency at which they are performed is feasible given the simulation overhead (see Section 5.5).

Another reason for running at least one other round of simulations, even when there are no dynamic workflow and platform changes, is that simulations, and in particular those executed at the onset of the execution, are necessarily inaccurate. During the execution, it becomes possible to compare the real execution to its previously simulated counterpart, determine causes of simulation inaccuracies, and thus improve the simulation's instantiation for better future accuracy. We term this process simulation error mitigation.

3.3. *Research Questions*

Our objective in this work is to determine whether SDPS has merit in the context of WMSs. To this end, we seek to answer the following research questions (RQs):

RQ#1: What is the potential improvement over the traditional approach? We wish to quantify the improvement that SDPS can afford over using a single scheduling algorithm.

RQ#2: What level of simulation accuracy is necessary? No simulation is perfectly accurate and we wish to determine the level of accuracy needed for SDPS to outperform or at least be comparable to the traditional approach.

RQ#3: Is it useful to mitigate simulation inaccuracy at runtime? At runtime it is possible to apply corrective measures to mitigate simulation error and apply SDPS again to select a scheduling algorithm. We wish to determine whether performing such simulation error mitigation is worthwhile.

RQ#4: What is the impact of the sophistication of the simulation? Simulators range in their sophistication as they can opt to use more or less naive models for simulating components of the target real-world system. We wish to determine whether a less sophisticated simulator still allows SDPS to rank candidate algorithms effectively.

RQ#5: Is SDPS effective for handling dynamic resource availability? Dynamic resource availability can occur in shared platforms (e.g., due to external load, to resources being reclaimed/released by a provider), in which case using different scheduling algorithms at different times could be beneficial.

RQ#6: Is simulation overhead sufficiently low? For SDPS to be feasible in practice, the simulation overhead should be sufficiently low when compared to the workload execution time.

4. Case Study

We evaluate the effectiveness of SDPS via a broad case study. We consider the execution of scientific workflow applications on a *multi-cluster* platforms, i.e., platforms that comprise one or more commodity clusters (with all computes nodes within the same cluster being homogeneous) connected over a wide-area network. The scheduling objective is to minimize overall execution time, or *makespan*.

In what follows we describe our experimental methodology (which relies exclusively on simulations), the platform configurations and workflow instances that we use to drive our simulations, and the scheduling algorithms we include in our portfolio.

4.1. *Experimental Methodology*

To perform a sound evaluation of SDPS we need: (i) an implementation of a WMS that executes workflows on multi-cluster platforms; and (ii) an implementation of a simulator of these executions that can be invoked at runtime by the WMS. We face two main technical difficulties. First, to answer RQ#2 in Section 3, we need to experiment with different levels of simulation (in)accuracy, including the best-case 100% accuracy. This would not be possible with a real-world implementation since a given simulator is necessarily inaccurate and its inaccuracy is typically unknown or at least uncharacterized. Second, we wish to evaluate SDPS on a large spectrum of workflows, platforms, and algorithms. For instance, in this work we consider a portfolio of 48 individual algorithms, plus SDPS, for $3 \times 9 = 27$ experimental scenarios (3 platform configurations, 9 workflow instances), for a total of 1,323 different application executions. Furthermore, we evaluate many different versions of SDPS and different random samples so that, overall, we obtain results for over 375,000 application executions. It would be prohibitive to obtain all these experimental results in a real-world setting, not only in terms of time and energy consumption, but also in terms of repeatability.

Given the above, we perform our experiments entirely in simulation. We implement a simulator of a WMS that executes workflows on multi-cluster platforms as an analog of a production WMS implementation for which SDPS can be implemented. That is, during its simulated execution, the WMS runs a simulation of its own future execution for each algorithm in the portfolio. The `fork` system call is used to clone child processes that each simulate the full remaining execution of the workflow using a particular algorithm and report the observed makespan to the parent process, i.e., the WMS. The WMS then picks the algorithm that achieved the lowest simulated makespan for future use. Once the workflow execution completes the simulator outputs the workflow makespan.

Our simulator is implemented using the WRENCH [48] (v2.1) and SimGrid [49] (v3.32) simulation frameworks. Simulator code, raw experimental data, and scripts to analyze and plot the data are publicly available on GitHub [47]. It takes as input *a platform configuration* and a *workflow instance*, as described in the next two sections. Although we base our platform configurations and workflow instances on real-world use cases, not all required information is available. As a result, we augment the available information using reasonable assumptions, as described in the next two sections. This, however, does not invalidate our evaluation results since our ground truth is based on simulated executions using this same augmented information.

4.2. Platform Configurations

We consider platforms that comprise commodity clusters with different numbers and types of compute nodes. Each cluster is homogeneous and its compute nodes are connected via a 100GbE interconnect. Each cluster is connected to the Internet via a network path with some bottleneck bandwidth. The compute nodes at each cluster have access to shared storage local to the cluster (network-attached storage, parallel file system, etc.) with some bounded aggregate I/O bandwidth. Whenever a compute node in a cluster needs to write application data, it writes it to the cluster's shared storage. Whenever a compute node in a cluster needs to read application data, it does so from the cluster's shared storage if possible. Otherwise, the data is read from a remote location (the user's machine, where all input data is located initially, or another cluster's storage) and then cached locally. We assume that storage capacity at each cluster is large enough to hold all application data if necessary. This is the case in practice when executing the workloads described in the Section 4.3. Considering storage capacity constraints would simply amount to removing some clusters from consideration when using the scheduling algorithms described in Section 4.4. Finally, in most our experiments we assume that the platform is dedicated to the application's execution and delivers constant performance throughout this execution (e.g., no external load or transient behaviors). In practice, this is achieved by reserving resources in some shared platform (starting virtual machine instances in cloud platforms, submitting pilot jobs to batch-scheduled clusters, etc.).

In [16], we conducted simulation experiments for 9 synthetic, arbitrarily generated platform configurations that comprised 1, 2, or 3 clusters. The preliminary results therein showed no clear trends or discernible patterns depending on the specific platform performance characteristics. The only observable trends were for the number of clusters in the platform (e.g., with a single cluster in the platform, algorithms that only differ in the way they select a cluster will necessarily be indistinguishable). Given those preliminary results, in this work instead, we base our platform configurations on an actual hardware platform, the Grid5000 [50] testbed.

Grid5000 comprises many clusters distributed over a wide-area network, and we consider three of these clusters: Ecotype, Dahu, and Neowise. Table 1 lists the clusters' hardware characteristics, most of which are derived based on advertised hardware specifications in [50]. Core speeds are from benchmark results obtained for an N-body physics simulation executed on each cluster's particular core type (Intel Xeon E5-2630L v4 for Ecotype, Intel Xeon Gold 6130 for Dahu, and AMD EPYC 7642 for Neowise). We do not have benchmark information regarding storage system bandwidths at these clusters, and we simply assume 100 Gbps for all clusters, which is typical of real-world platforms. Finally, these clusters are deployed on a wide-area network with 10Gbps end-to-end bandwidths. The bandwidths shown in Table 1 correspond to a particular observation on the Grid5000 platform (which provides a real-time network weather map) in which there is some background network traffic to/from the Dahu and Neowise clusters.

We consider 3 platform configurations, P_1, P_2, and P_3, where P_x corresponds to using the first x clusters from left to right in Table 1 (see the bottom row). These configurations correspond to different test cases for our approach in terms of platform heterogeneity and scheduling decision complexity going from P_1 (fully homogeneous) to P_3 (most heterogeneous).

4.3. Workflow Instances

We drive our simulations using 9 workflow specifications based on real-world scientific applications, as listed in Table 2, 8 of which are from the Bioinformatics domain, in which workflows (or "pipelines") are common-place. The Cycles workflow comes from the Agroecosystem domain. These workflow instances are provided by the WfCommons project [51] and were constructed based on logs from actual executions [52]. Each workflow instance defines a set of tasks, with specified execution times, and a set of files, with specified sizes. Each file can be input to and/or output from tasks, thus creating data dependencies. The metrics shown in the table show that our workflow instances are diverse, with different structures and different computation-data ratios. Overall, we expect that different scheduling algorithms will fare differently across these workflow instances. [1].

[1] In previous work [16], we used a WfCommons instance of the Montage workflow, but have removed it from this study as that particular instance had many tasks with sub-second execution times, making execution on a distributed multi-cluster inadvisable in the first place unless task aggregation is performed (in this work we execute each WfCommons workflow instance as is).

Table 1: Cluster configurations

Cluster	Ecotype	Dahu	Neowise
Number of nodes	48	32	10
Number of cores per node	10	16	48
Core speed (Gflop/sec)	3.21	4.01	6.48
Storage r/w bandwidth (Gbps)	100	100	100
Internet bandwidth (Gbps)	10	7	8
Used in platform configurations	P_1, P_2, P_3	P_2, P_3	P_3

Table 2: Workflow instances, indicating for each the application name ("Name"), the number of tasks ("#Tasks"), the sequential compute time on a single 3.21Gflop/sec core ("Work"), the number of data files ("#Files"), the sum of all data file sizes ("Footprint"), the length of the longest path in the workflow's task graph in number of tasks ("Depth"), and the maximum number of tasks that can be executed in parallel ("Max Width").

Workflow	Name	#Tasks	Work	#Files	Footprint	Depth	Max Width
W_1	1000Genomes	328	6h02m	352	24.72GB	3	216
W_2	BLAST	303	8h45m	907	471.79KB	3	302
W_3	BWA	1004	3h44m	3012	56.79MB	3	1002
W_4	Cycles	874	5h13m	6930	6.18GB	4	652
W_5	Epigenomics	1095	5h40m	1370	8.26GB	9	542
W_6	RNA-Seq	197	0h43m	680	290.80MB	10	121
W_7	SoyKB	156	6h47m	321	2.83GB	11	121
W_8	SRA-Search	22	5h16m	48	16.51GB	3	21
W_9	Viralrecon	203	0h42m	877	270.79MB	18	52

WfCommons workflow instances specify task execution times in seconds, and give the specifications of the processors on which tasks where executed. None of these processors correspond to the processors of the clusters in the Grid5000 clusters described in the previous section. Furthermore, some of the platforms on which workflows were executed are heterogeneous, meaning that different tasks ran on different kinds of processors. In this work, we arbitrarily assume that the execution time for each task in the WfCommons workflow instances is for execution on a compute node of the Ecotype cluster, the cluster with the slowest nodes in our platform configurations (see Table 1). In our experiments, if a task is executed on the Dahu, resp. Neowise, cluster, then its execution time is scaled down by a factor 4.01/3.21 = 1.25, resp. 6.48/3.21 = 2.02, i.e., the task execution time is proportional to the core speed. Scaling the execution times by different factors would lead to different data-intensiveness of the workflows (but our workflows already span a spectrum of data-intensiveness).

WfCommons workflow instances do not include information about the execution of workflow tasks on multiple cores, but only give a single execution time t. Due to this lack of information, we assume that this time t was measured for a single-threaded task execution on a single core, and we assume an Amdahl's Law parallel speedup behavior [53]: a task that executes in time t on 1 core executes in time $\alpha \cdot t/n + (1 - \alpha) \cdot t$ on n of these cores. We sample α uniformly between 0.5 and 0.9 for each workflow task. This ensures that our experiments include a broad range of parallel multi-core performance behaviors. Note that this model is general enough to correspond to any notion of a processing elements with a parallel speedup behaviors, such as GPUs for instance. Although our experiments are for platform configurations that comprise multi-core hosts and workflow executions on such platforms, our results are not specific to the underlying processor architecture.

4.4. *Algorithms*

As discussed in Section 2.1.3, scheduling driven by online simulations has been proposed in the context of batch scheduling for HPC clusters. In that specific context, a few classical algorithms are prevalent and lead to a natural portfolio. By contrast, in the the context of scientific workflow scheduling a very large number of scheduling algorithms have been proposed over the years, which has prompted many survey articles [5–13, 54]. The vast majority of algorithms proposed for makespan minimization perform list-scheduling [55] for selecting ready tasks and compute resources. The algorithm is invoked whenever there is at least one ready task and at least one available compute resource. Given our case study's application and platform models list-scheduling consists in:

1. Selecting a ready task using some criterion (C_1);
2. Selecting a cluster with at least one idle core using some criterion (C_2);
3. Selecting a number of cores for the task execution using some criterion (C_3);

4. Scheduling the selected task on the selected cluster using the selected number of cores.

Many options have been proposed in the literature for defining the above criteria, resulting in an enormous number of possible algorithms that could be included in a portfolio. An yet, as discussed in Section 3.1.1, most of these algorithms are not implemented in production WMSs. One of the reasons is that many proposed algorithms are difficult to implement in production systems because they rely on performance models to be designed and implemented by WMS developers. For instance the classic Heterogeneous Earliest Finish Time (HEFT) algorithm prioritizes tasks by an estimate of their earliest finish times. Implementing this algorithm thus requires that a performance model be implemented to compute this estimate. Developing accurate performance models is challenging due to the need to account for a range of complex phenomena (e.g., network contention effects, data locality). The effectiveness of sophisticated scheduling algorithms implemented based on inaccurate performance models is questionable. Furthermore, the information necessary to implement accurate performance models is not necessarily available in the first place. Given all the above, for this case study we have opted to build a portfolio of list-scheduling algorithms that can be implemented in current production WMSs solely based on available information about the workflow specification, the platform, and the current state of the execution, without relying on any performance model. This is the same information necessary to instantiate a simulation for SDPS, as explained in Section 3.1.2. Specifically, we consider the following options for each of the above criteria:

- Criterion C_1 (task selection):
  - 0: Pick the task with the largest bottom-level (i.e., prioritize tasks on the critical path);
  - 1: Pick the task with the largest number of children tasks;
  - 2: Pick the task with the largest amount of input and output data (in total bytes);
  - 3: Pick the task with the largest computational load.
- Criterion C_2 (cluster selection):
  - 0: Pick the cluster with the fastest cores.
  - 1: Pick the cluster with the larger number of idle cores;
  - 2: Pick the cluster with the most idle compute capacity (number of idle cores multiplied by the core speed);
  - 3: Pick the cluster that holds the largest amount of task input data (in total bytes) in its shared storage;
- Criterion C_3 (number of cores selection):
  - 0: Pick as many cores as possible while ensuring that the task's parallel efficiency is above 90%;
  - 1: Pick as many cores as possible while ensuring that the task's parallel efficiency is above 50%;
  - 2: Pick as many cores as possible.

We denote each algorithm as A_x, where $x = 12 \times C_1 + 3 \times C_2 + C_3$, which gives us 48 different algorithms (A_0 to A_{47}). All above criteria, or variations thereof, have been proposed time and again in the scheduling literature. For instance, the first option for criterion C_1 corresponds to the classic idea of prioritizing tasks on the critical path [56] and the second option corresponds to the classic idea of generating as much parallelism as quickly as possible [57]. Although many other options could be considered, these 48 algorithms provide us with a sufficiently large and diverse algorithm portfolio (see Section 5.1).

We include these 48 algorithms in the portfolio for all experiments and for all our platform/workflow combinations, and the same portfolio is used throughout the whole execution. We note that the portfolio could be reduced for some of these combinations. For instance, for platform P_1 there is no need for different cluster selection schemes (criterion C_2) since there is a single cluster. But the key idea of portfolio scheduling is that algorithms that perform poorly will simply not be used, and so we always include all 48 algorithms in the portfolio.

5. Results

5.1. Diversity of the Algorithms in the Portfolio

In Section 4, we claim that our experimental scenarios (workflow instances and platform configurations) would lead the algorithms in the portfolio to exhibit a range of behaviors. In this section, we verify this claim quantitatively. Figure 1 shows, for each experimental scenario (i.e., a workflow and platform combination) the relative difference, in percentage, between the makespan achieved by each algorithm and that achieved by the best algorithm for this scenario. This percentage is typically termed "degradation from best" or *dfb*. In other terms, if for a particular experimental scenario each algorithm i achieves a makespan m_i, then the *dfb* of algorithm j is defined as:

$$dfb(j) = 100 \times \frac{m_j - \min\limits_i m_i}{\min\limits_i m_i} .$$

$dfb(j) = 0\%$ means that algorithm j achieves the lowest makespan for that experimental scenario. $dfb(j) = 100\%$ means that algorithm j achieves a makespan twice as long as the makespan achieved by the best algorithm.

In Figure 1, the scenarios are sorted by increasing value of the maximum *dfb* value over all algorithms. This maximum *dfb* value ranges from 5.11% to 424.80%, and is above 100% for 16 of the 27 experimental scenarios. This means that there is more than a 2x difference between the makespan achieved by the best algorithm and that achieved by the worst algorithm for that experimental scenario for more than 50% of our experimental scenarios. We conclude that our experimental scenarios are sufficient to highlight the diversity of our 48 scheduling algorithms.

Although the above results indicate diversity, one may wonder whether some (or perhaps just one?) algorithm is always best, in which case, one should just use that algorithm. The answer to RQ#1 in Section 3.3 would then be that SDPS has little potential improvement over the traditional one-algorithm approach. Computing each algorithm's average *dfb* over all experimental scenarios, we find that algorithm A_8 achieves the lowest average *dfb* at 12.63%. Algorithm A_8 prioritizes tasks

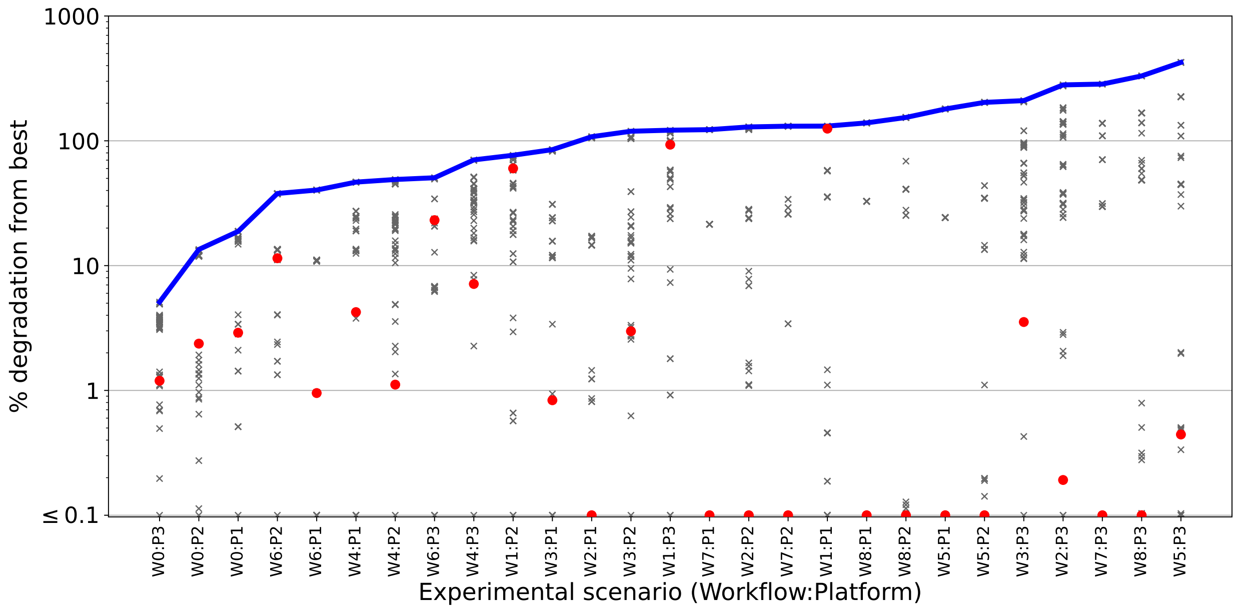


Figure 1: Degradation from best (*dfb*) of each algorithm in the portfolio vs. experimental scenarios sorted by increasing maximum *dfb* (shown as a blue solid line). Data points for algorithm A_8, which has the lowest average *dfb*, shown as red dots.

with largest bottom-level ($C_1 = 0$), selects the cluster with most idle compute capacity ($C_2 = 2$), and uses as many cores as possible on a compute node ($C_3 = 2$). While algorithm A_8's average *dfb* is relatively low, it is not always a good choice. It happens to be the best (or within 1% of the best) choice for only 14 of our 27 scenarios. It has a *dfb* higher than 10% for 5 of the remaining 13 scenarios, and as high as 125.16% for the W_1:P_0 experimental scenario, as seen in Figure 1. We conclude that no single algorithm is best. Although algorithm A_8 is the best on average it can perform relatively poorly for some experimental scenarios.

In all that follows, we consider as our competitor a runtime system that implements and uses algorithm A_8 as its only scheduling algorithm. Since algorithm A_8 has the best average *dfb*, it corresponds to the best choice a runtime system developer could make if asked to pick one algorithm to implement in their system, at least in the scope of our experimental settings. It is unclear how a developer would identify this algorithm, short of conducting an extensive experimental study. In fact, they could very well pick another algorithm, in which case all results hereafter would be more favorable (and often drastically so) for SDPS. For simplicity, we call this competitor OneAlg.

5.2. *Impact of Simulation Error*

With 100% accurate simulations SDPS would necessarily pick the best algorithm for each experimental scenario. In this section, to answer RQ#2 in Section 3.3, we quantify the sensitivity of SDPS to simulation inaccuracy.

5.2.1. *Simulation Error Injection Method*

To the best of our knowledge, there is no model or characterization of the simulation error behavior of simulators of parallel and distributed computing platforms and applications. But we note that simulation error is not fully random as it typically stems from simulation models under- or over-estimating the performance delivered by hardware resources when executing application activities. That is, for each such resource, the simulation suffers from some constant bias. This bias can be due to a simulation model being inherently biased or to an incorrect instantiation of the model's configuration parameters (e.g., due to imperfect knowledge about the hardware resource available).

When SDPS performs its round of simulations we introduce simulation error by injecting random perturbations in the platform's resource speeds, i.e., each cluster's internet bandwidth, storage system bandwidth, and core compute speed. If the actual value of a speed parameter is x, in the simulation it is set to value $\mathcal{U}(\max(0, x \times (1 - e)), x \times (1 + e))$, where $\mathcal{U}(a, b)$ denotes the uniform random distribution on the (a, b) open interval and e denotes the magnitude of the error range. Since simulations are conducted with erroneous values of these metrics, they report erroneous makespans, based on which a scheduling algorithm is selected. Note that even small simulation errors can cause drastically different scheduling decisions. As e increases there is a higher probability for SDPS to select an algorithm that is not (and is possibly much worse than) the best algorithm for the upcoming application execution.

5.2.2. *Using a Single Round of Simulations*

In this section we assume that algorithm selection is based on a single round of simulations at the onset of the workflow execution, and that the selected algorithm is then used throughout the execution.

We conducted experiments for our 9 workflow instances and 3 platform configurations for $e \in \{0.1, 0.2, \ldots, 1.0\}$, with 100 trials (i.e., 100 different random number generator seeds)

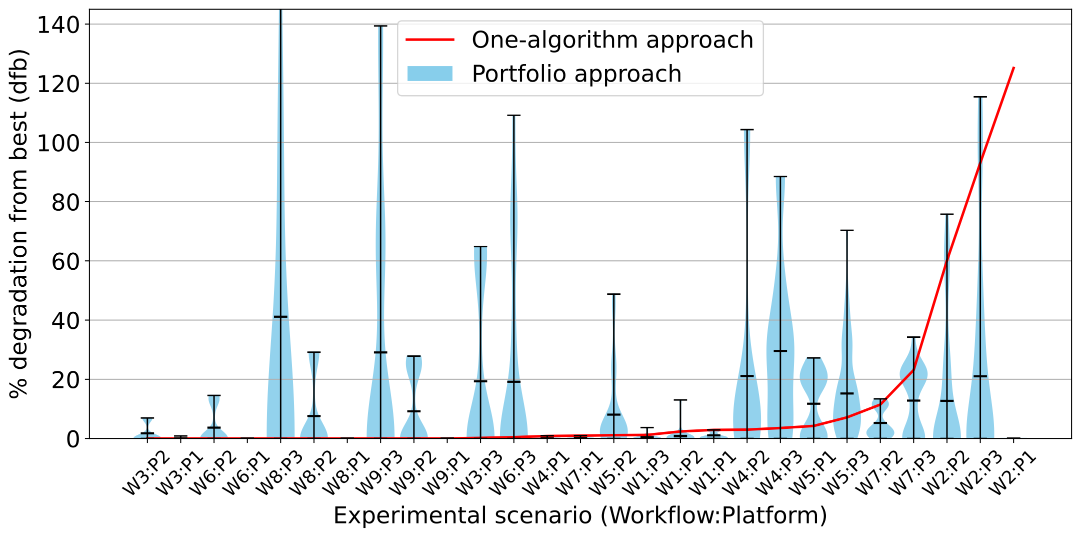


Figure 2: *dfb* vs. experimental scenarios, for simulation error magnitude $e = 1.0$.

for each e value. Figure 2 shows *dfb* results for $e = 1.0$ for all $9 \times 3 = 27$ experimental scenarios. The scenarios are sorted by increasing *dfb* for OneAlg. Results for OneAlg are shown as a single line since its performance is not affected by simulation error (as it does not use simulation). Results for SDPS are shown as a violin plot for each experimental scenario, which depicts the distribution of 100 data points. The bottom, resp. top, horizontal bar of the violin plot shows the minimum, resp. maximum, value, while the middle horizontal bar shows the average. As seen in Figure 1, OneAlg is the best for many of these scenarios. However, for some scenarios, seen on the right of the horizontal axis in Figure 2, its *dfb* can be large.

The main observation from Figure 2 is that even with high simulation error SDPS still performs relatively well for many scenarios. Its average *dfb* is below 5% for 12 of the 27 experimental scenarios, and above 20% for only 5 of them. Although there are cases in which SDPS is largely outperformed by OneAlg, the converse is also true. For some scenarios the *dfb* distribution of SDPS has a high maximum value. These correspond to cases in which simulation error causes SDPS to pick an algorithm that does not perform well. We note that these cases occur mostly for scenarios with the 3-cluster platform configuration (P_3), which has the largest scheduling decision space. Conversely, we see that for the 1-cluster platform configuration (P_1), SDPS is more tolerant to simulation error.

Figure 3 shows results for $e = 0.3$. Expectedly, the results are vastly improved, with SDPS's average *dfb* at most 18.54%. Overall, SDPS is on par with or vastly preferable to OneAlg.

Figure 4 shows the distribution of the *dfb* values achieved by SDPS over all experimental scenarios for all values of e. As expected, SDPS's *dfb* improves as e decreases, with the average converging to zero. The figure also depicts the *dfb* of OneAlg, which is 12.63%. We see that SDPS leads to improvements over OneAlg with relatively high probability even when simulation error is 100%. At simulation error 50% or lower, SDPS outperforms OneAlg in more than 90% of the cases.

To better explain the previous results, we compute the rank of the algorithm picked by SDPS in the portfolio (i.e., algorithm with rank 0 is the best, algorithm with rank 1 is the second-best, etc.). Figure 5 shows the cumulative distribution of the rank of the algorithm selected by SDPS over all experimental scenarios, for all values of e. That is, a point at coordinate (x, y) in the plot means that SDPS selected an algorithm with rank at least x for y% of the experimental scenarios. Even with high error ($e = 1.0$), SDPS selects the best algorithm in more than 50% of the cases, and one of the top 5 algorithms in about 70% of the cases. However, we see that algorithms with high rank (i.e., the worst algorithms in the portfolio) are still selected occasionally, which explains the high maxima for some of the violin plots shown in this section. Expectedly, results improve as e decreases. For instance, for $e = 0.3$, SDPS selects the best algorithm in more than 70% of the cases and a top-5 algorithm in more than 90% of the cases.

We conclude that even with high simulation error SDPS rarely picks a "bad" algorithm, and as a result compares favorably to OneAlg.

5.2.3. *Simulation Error Mitigation via Another Round of Simulations*

Simulators developed using SimGrid and WRENCH, such as the one developed in this work, have been reported to achieve simulation error well below 20% [39, 58], which, as seen in the previous section, would have little negative impact on the efficacy of SDPS. Unfortunately, while it is certainly possible to develop low-error simulators, doing so is not a given. This is because one must calibrate the simulator (i.e., pick values for all its configuration parameters) so as to match particular real-world applications and platforms. Simulator calibration is typically a non-trivial, labor-intensive, and manual process [46]. It is thus reasonable to expect that many simulators could exhibit relatively high error in practice.

One approach to reduce simulation error is to perform *sim-*

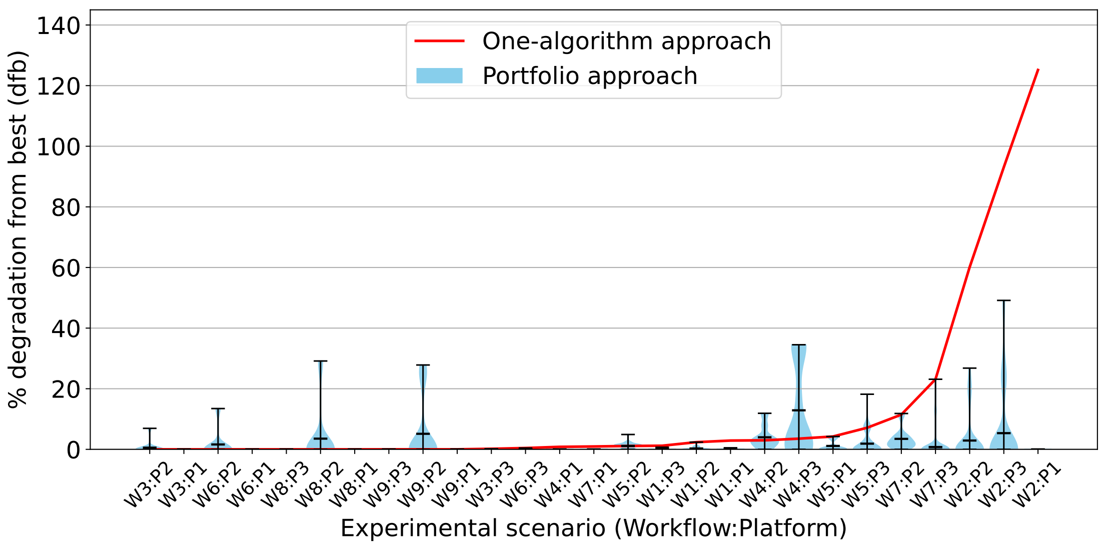


Figure 3: *dfb* vs. experimental scenarios, for simulation error magnitude $e = 0.3$.

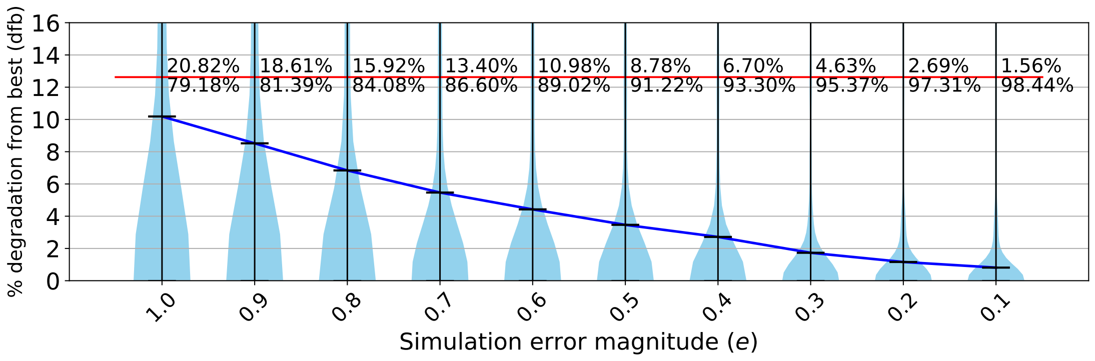


Figure 4: *dfb* vs. simulation error (e). Percentages denote fractions of SDPS values above/below OneAlg's value.

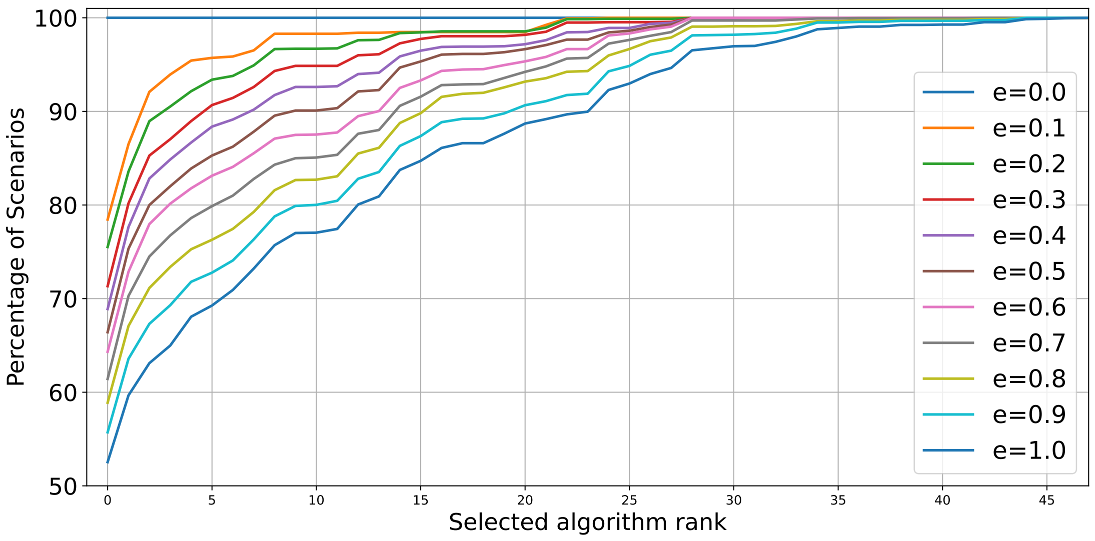


Figure 5: Cumulative distribution of the rank of the algorithm selected by SDPS for each simulation error magnitude value (e).

ulation error mitigation when using SDPS. The runtime system can keep a record of the simulated execution for the algorithm that ends up being selected after a round of simulations, and then compare this execution to what actually happened in the real execution. The goal is to identify sources of simulation error, and adjust the instantiation of the simulator before the next round of online simulations. For instance, comparing the real and the simulated execution could reveal that some network bandwidth has been underestimated by some factor. A new round of simulations could be conducted after applying a correction factor to that bandwidth, so as to pick a possibly different scheduling algorithm for the rest of the application execution.

To answer RQ#3 in Section 3.3 we repeat the experiments in the previous section but assuming that simulation mitigation is performed. That is, after a task completion occurs and 10% of the application's total work has already been executed, then a second round of simulations is conducted with error $e' < e$. Assuming that the actual value of a platform's performance metric is x, in the first round of simulations the simulations use value $x' = \mathcal{U}(\max(0, x \times (1-e)), x \times (1+e))$. In the second round of simulations, a value $x'' = x + (x' - x) \times (e'/e)$ is used, which is closer to x by a factor $e'/e < 1$. In this manner, we can evaluate how SDPS fares for ranges of initial and mitigated errors.

Figure 6 is similar to Figures 2 and 3, but shows results for $e = 1.0$ and $e' = 0.3$. Error mitigation leads to improvements. Specifically, over the 27 experimental scenarios, the maximum *dfb* was improved for 19 scenarios and the average *dfb* was improved for 22 scenarios. In terms of average *dfb* across all scenarios, SDPS achieves a *dfb* of 10.18% with $e = 1.0$ and no mitigation, and a *dfb* of 4.48% with $e = 1.0$ and mitigation down to $e' = 0.3$, while with an initial error $e = 0.3$ SDPS achieves an average *dfb* value of 1.73%.

One observation when comparing Figure 6 to Figure 2, which holds for all other e and e' values, is that error mitigation is effective only for some workflows. To illustrate this behavior, let us consider the execution of workflows W_6 and W_8 on platform P_3 (results are similar, but not as pronounced for P_1 and P_2 because the scheduling decision space is smaller). Figure 7 shows average *dfb* results when SDPS is used to execute workflow W_6. The horizontal axis shows e' values (i.e., mitigated error) and each curve is for a different e value (i.e., initial error). Each curve shows a decreasing trend until $e' = 0.3$, at which point the curve flattens out. Furthermore, the curve for a particular e value is for the most part below the curves for higher e values. For this workflow, error mitigation is useful, but, expectedly, a lower initial error value e is preferable. Trends are similar for the W_1, W_3, W_4, W_5, W_7, and W_9 workflow instances, but with different slopes and final plateaux.

Figure 8 shows results for the W_8 workflow (results for W_2 are similar). For this workflow error mitigation provides little improvement. This is due to the structure of this workflow, which consists of a first level of compute-heavy tasks that account for the majority of the total work. That is, once initial scheduling decisions have been made for these tasks, scheduling decisions for subsequent tasks have almost no effect. Recall that we perform error mitigation only after a task completion occurs. The rationale is that, in real-world systems, logging information is typically not available before a task completes, and there is thus no available real-world execution data to perform simulation error reduction. For this workflow, by the time the first task completion occurs, more than 95% of the total work has already been scheduled. The effectiveness of SDPS is thus entirely driven by the initial simulation error.

We conclude that simulation error mitigation is useful provided it can be performed before the bulk of the total work has begun executing.

5.3. Impact of Simulator Sophistication

To answer RQ#4 in Section 3.3 we quantify the extent to which SDPS is sensitive to the sophistication of the simulator. In particular, we want to determine whether a simulator that employs simplistic, or even naive, simulation models can still be useful for SDPS. To answer this question we repeat experiments described in previous sections but we disable features of our simulator so as to reduce its level of sophistication. First, we disable the simulation of network and I/O contention since many simulators neglect the simulation of contention effects on network and I/O devices. They do so because simulating contention accurately is challenging [40]. Furthermore, simulators that do not simulate contention are often used in the context of scheduling, since ignoring contention greatly simplifies the design and evaluation of scheduling algorithms [59]. Second, we disable the simulation of realistic multi-core parallel speedup. Our simulator uses an Amdahl's law parallel speedup model for estimating the execution of a workflow task on multiple cores. This model requires benchmark information about the workflow tasks, which is not always readily available. Instead, a less sophisticated option is to assume that every task has 100% parallel efficiency.

Together these 2 simplifications let us explore 4 possible simulators to use for scheduling decisions, each at a different level of sophistication, which we denote as CA, $C\overline{A}$, $\overline{C}A$, $\overline{CA}$, where C denotes contention, A denotes Amdahl's law, and a bar on the letter means that feature is disabled in the simulator. So, for instance, $\overline{C}A$ denotes our simulator with no contention simulation but with Amdahl-based parallel speedup simulation. All the results in previous section were for the CA simulator.

Figure 9 shows the cumulative *dfb* distribution across all experimental scenarios for two simulation error magnitudes, $e = 0$ and $e = 0.3$. A point at coordinate (x, y) means that for y% of the experimental scenarios a *dfb* value better than x is achieved. That is, the faster a curve approaches the $y = 100\%$ line, the better. Simulation error magnitude $e = 0.3$ was selected as a reasonable level of error in a calibrated simulator, but results are similar for all other e values (full results available in [47]).

Expectedly, using the CA simulator leads to the best results overall. Using the $\overline{C}A$ simulator leads to results only slightly worse, which indicates that simulating contention, for our experimental settings, is not critical for SDPS to be effective. This is because our workflow instances do not correspond to highly data-intensive scenarios. Using simulators $C\overline{A}$ or $\overline{CA}$ leads to noticeably worse results. Interestingly, using $\overline{CA}$ often outperforms using $C\overline{A}$ despite it being less sophisticated. This

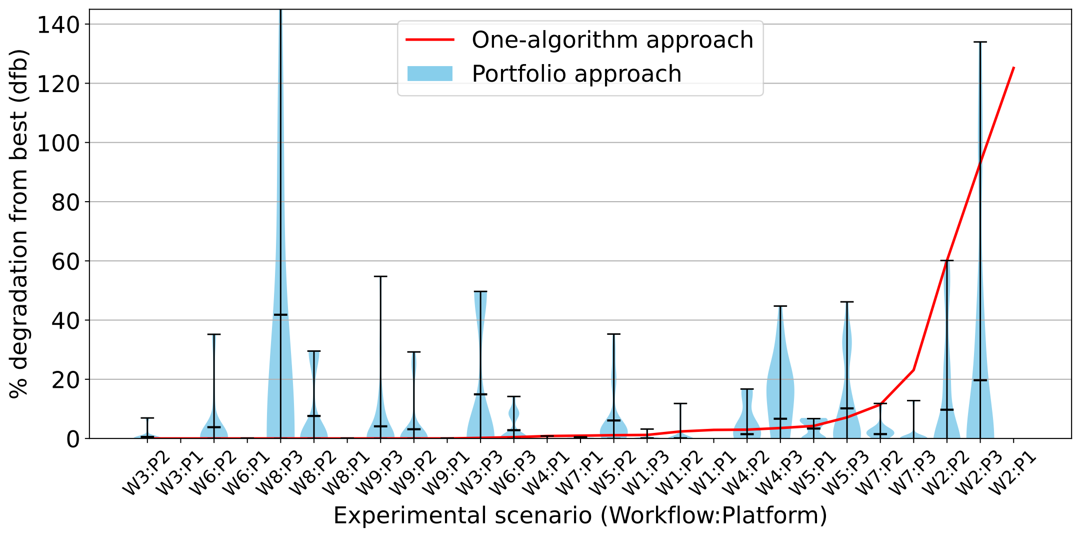


Figure 6: SDPS's *dfb* vs. experimental scenarios, for simulation error magnitude $e = 1.0$ and mitigated error magnitude $e' = 0.3$.

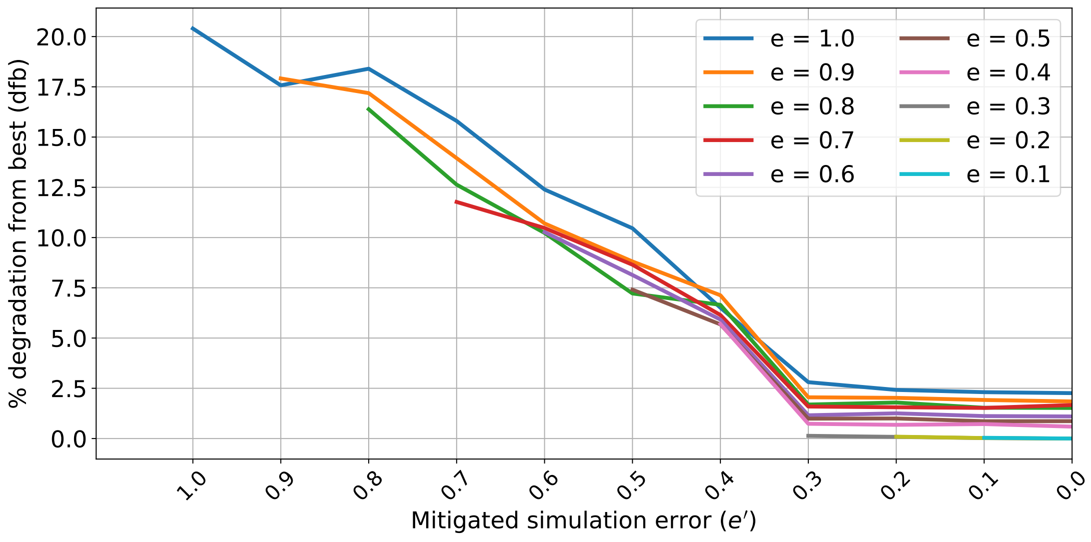


Figure 7: SDPS's average *dfb* vs. mitigated error (e') for each initial error magnitude (e), for workflow instance W_6.

is because the ranking of candidate scheduling algorithms for a particular scenario strongly depends on the communication-to-computation ratio of the execution. $\overline{CA}$, unlike $C\overline{A}$, underestimates both communication times (because it ignores contention) and computation times (because it assumes perfectly parallelizable workflow tasks). It thus ends up with a more accurate ranking of the candidate scheduling algorithms than the more sophisticated $C\overline{A}$.

The results in Figure 9 are aggregated over all workflows, but different workflows show different trends. Table 3 shows, for each workflow and for each simulator sophistication level, the percentage of experimental scenarios for which a *dfb* value under 10% is achieved, for a simulation error magnitude $e = 0.3$. The main observation is that for some workflows, like W_1, simulator sophistication has no impact, but for others, like W_2, using more sophisticated simulators yields dramatic improve-

Table 3: Percentage of *dfb* values below 10% for each workflow when using each of the four simulation sophistication level, for simulation error magnitude $e = 0.3$.

Workflow	CA	$\overline{C}A$	$C\overline{A}$	$\overline{CA}$
W_1	100.00	100.00	100.00	100.00
W_2	100.00	100.00	0.00	0.00
W_3	100.00	100.00	66.67	100.00
W_4	100.00	66.67	66.67	100.00
W_5	100.00	66.67	33.33	100.00
W_6	100.00	100.00	66.67	100.00
W_7	100.00	100.00	100.00	66.67
W_8	100.00	100.00	66.67	100.00
W_9	100.00	100.00	66.67	100.00

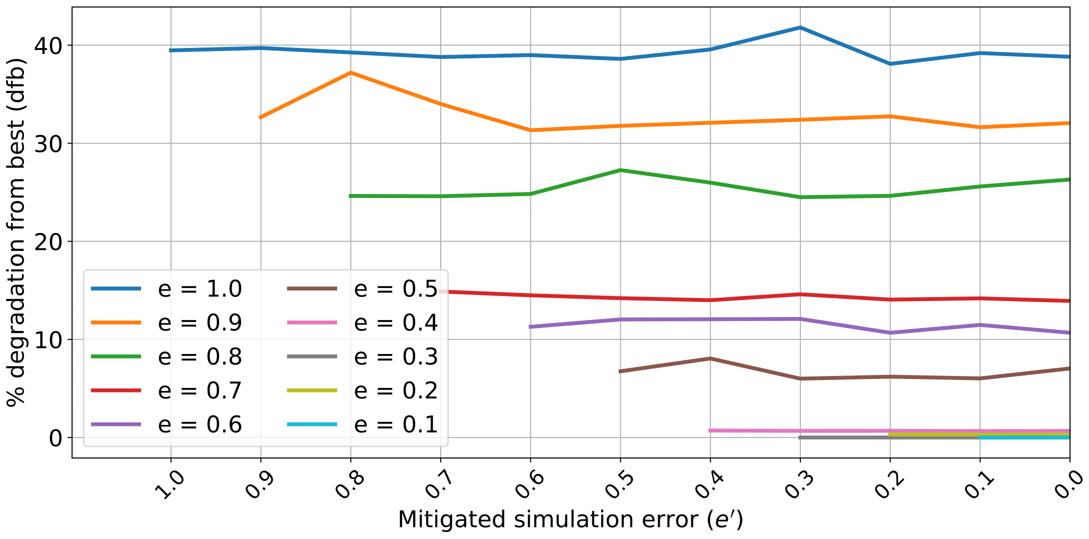


Figure 8: SDPS's average *dfb* vs. mitigated error magnitude (e') for each initial error (e), for workflow instance W_8.

ment. In the case of W_2, simulating parallel speedups accurately is critical. For some workflows, using a less sophisticated simulator can yield marginally better results than using the most sophisticated simulator (*CA*). This is because particular simulation errors can occasionally cause a less correct simulator to produce a more accurate ranking of the candidate scheduling algorithms. But overall, using the most sophisticated simulator yields low *dfb* values across all workflows.

Expectedly, a higher level of simulator sophistication is better. However, lower levels of simulator sophistication can still yield good results for SDPS. In the scope of our experimental settings, for instance, not simulating network and I/O contention typically leads to only marginal makespan degradation. Unfortunately, the needed level of sophistication depends on the specifics of the platform and application scenarios. In our results, a less sophisticated simulator can lead to almost no degradation for some workflows and to large degradation for others. If these specifics are known in advance, it may be possible to determine the needed level of sophistication a priori.

5.4. *Adapting to Dynamic Platform Changes*

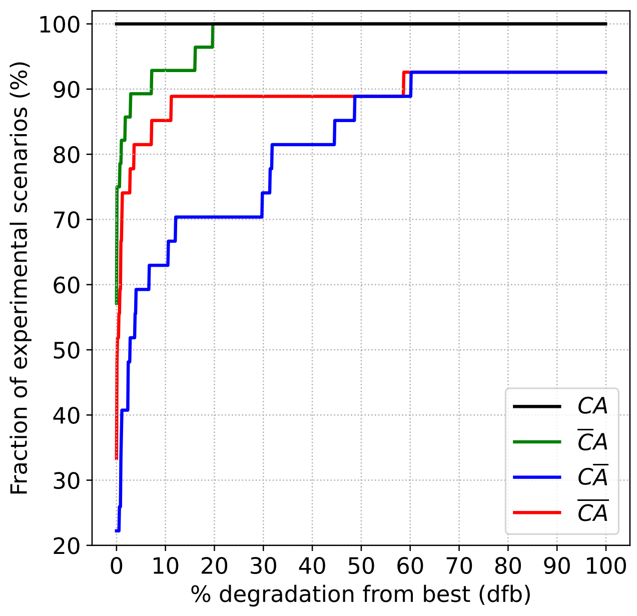
All experiments in the previous sections assume that the workflow executes on dedicated resources reserved for that execution. In many practical use cases, however, resources allocated to a workflow's execution can fluctuate (e.g., on clouds where virtual machine instances can be created and can expire dynamically, on batch-scheduled HPC platforms when using pilot jobs [60]). SDPS can then be used periodically through the execution to adapt to these changes and select different scheduling algorithms for different platform conditions. We have conducted experiments with a simple dynamic resource availability model to answer RQ#5. Our main finding is that while significant improvements from these adaptations can sometimes be achieved, the results are workflow-dependent (similarly to the findings in Section 5.2.3). We refer the reader to [47] for supplementary material that provides full details.

5.5. *Simulation Overhead*

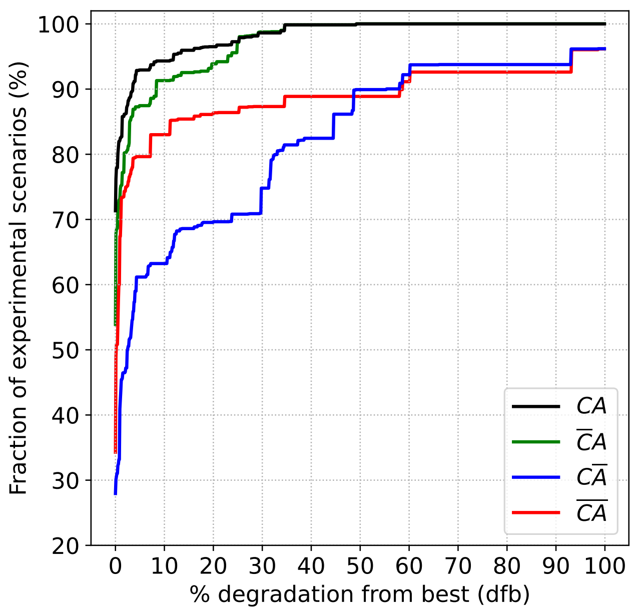
To answer RQ#6 in Section 3.3, we quantify simulation overhead and discuss possible approaches to mitigate it. Recall that in this work we consider that simulations are executed on the host on which the WMS itself executes. For SDPS to be practical, simulation time must be low relative to the makespan since a round of simulation needs to be executed at the onset of the execution. Furthermore, additional rounds of simulation may be useful throughout the execution, for simulation error mitigation purposes or for adapting to dynamic changes of the workflow and/or platform. In practice, simulations can be executed concurrently with the application so that the simulation overhead is fully hidden [23]. However, a high simulation overhead can still be harmful as it would delay the time at which the algorithm selection decision is reached.

Table 4 shows results obtained when simulating the full execution of each of our workflow instances on platform configuration P_3 using algorithm A_8 (the best algorithm on average). We find that all peak memory footprints are low (at most 102.06MB) and that, for most workflow instances, the ratio of simulated makespan to simulation time is large. This is because for discrete-event (as opposed to discrete-time) simulation, computational complexity depends on the number of events to simulate and not on the length of time being simulated. The lowest ratio is for the W_4 workflow, for which the ratio is "only" 36.43x. Overall, we find that in general the time to simulate the execution is small, and often negligible, when compared to the real execution.

The results in Table 4 are for the simulation of one algorithm, but SDPS runs one simulation per algorithm in the portfolio. These simulations are independent and can be executed concurrently on multiple cores, which is feasible due to low memory footprint. For instance, running 48 concurrent simulations for workflow W_3, which causes the largest memory footprint, requires less than 5GB of RAM. Running these 48 simulations concurrently on a machine with a 2.40GHz Intel Xeon



(a) Cumulative *dfb* distribution for simulation error magnitude $e = 0.0$



(b) Cumulative *dfb* distribution for simulation error magnitude $e = 0.3$

Figure 9: Cumulative *dfb* distribution for all experimental scenarios for two simulation error magnitude values.

Table 4: Simulated makespan, simulation time, ratio thereof, and peak memory footprint when simulating the execution of each workflow on platform P_3 with algorithm A_8. Results obtained on a 2.80GHz core (Intel Xeon Gold 6242 CPU), averaged over 10 samples.

Workflow	Simulated Makespan (sec)	Simulation Time (sec)	Ratio	Peak Memory Footprint (MB)
W_1	10573.78	0.57	18457.90	29.82
W_2	117.16	0.55	213.39	32.83
W_3	455.60	3.67	124.11	102.06
W_4	106.53	2.92	36.43	86.80
W_5	265.60	1.66	159.79	47.40
W_6	113.49	0.26	44.96	25.92
W_7	10347.80	0.83	12,396.92	51.45
W_8	602.17	0.03	17,295.35	19.93
W_9	90.99	0.26	343.77	26.18

Gold 6240R 48-core processors takes 10.03 seconds, while running only the slowest of these simulations (simulations take different amounts of time depending on the scheduling algorithm) takes 8.27 seconds. We conclude that although many simulations need to be executed, they can be executed concurrently on multiple cores with high parallel efficiency.

One option to reduce simulation time is to not fully simulate the execution to completion. We investigated this option in previous work [16] but found that its effectiveness is highly workflow-dependent. For some workflows simulating only a small fraction of the upcoming execution has almost no negative impact on SDPS, but for others the negative impact is large. This is because different workflows have different task- and data-dependency structures, and in some cases the impacts of initial scheduling decisions are not felt until the later phases of the execution.

If the algorithm portfolio is too large, one possibility is to prune the set of candidate algorithms. For instance, in [15] it is proposed that algorithms be placed in different categories depending on their past simulated performance, and that a bounded amount of simulation time be allocated to each category. This approach could be used in our context, with the caveat that the performance of an algorithms in earlier phases of the execution may be misleading [16]. Another approach, which we leave for future work, is to train a ML-based surrogate model, such as a neural network, that learns the simulator's behavior and can be used as a fast approximation of the simulator.

6. Conclusion

In this work, we have assessed the potential merit of using simulation-driven portfolio scheduling (SDPS) in runtime systems that automate the execution of scientific workflow applications on parallel and distributed computing platforms. Our results show that SDPS outperforms the one-algorithm approach, even when this approach happens to use the algorithm that performs best on average across all experimental scenarios considered in this work. Crucially, SDPS still performs well in the presence of relatively large simulation error, i.e., much larger than what state-of-the-art simulators have been reported to exhibit. Furthermore, simulation error mitigation at runtime can be effective. We also found that, for some execution scenarios, even unsophisticated simulators can be used by SDPS and achieve good results. Finally, simulation overhead is sufficiently low for SDPS to be used in practice.

In our results we have compared SDPS to the best possible choice a runtime system developer could make for implementing the one-algorithm approach in the context of our study, i.e., pick algorithm A_8. The main motivation for this work is that it is not clear how the developer could identify this algorithm in practice (short of conducting a full experimental case study as done in this work). Were the developer to pick the median algorithm, algorithm A_{22}, which has a relatively low average *dfb* at 29.38% (A_8 is at 12.63% and the worst algorithm is at 124.21%), all results presented in Section 5 would be improved. For instance, with high simulation error $e = 1.0$ and no error mitigation, SDPS would outperform the one-algorithm approach on average for 25 of the 27 experimental scenarios (as opposed to only 9 of them as seen in Figure 2).

We conclude that, portfolio scheduling, because it obviates the challenge of picking a particular scheduling algorithm to implement in a WMS, thus has the potential to resolve the disconnect between scheduling research and scheduling practice in the context of scientific workflows. A clear future work direction is the development of simulation forensics techniques for detecting and mitigating simulation error at runtime. Another future direction is using SDPS for optimizing other application execution metrics (e.g., energy consumption, monetary cost) and for addressing notoriously difficult multi-objective scheduling problems that consider multiple such metrics (e.g., satisfying multiple QoS requirements, minimizing makespan given an energy consumption budget).

Acknowledgments

This work is funded by National Science Foundation awards #2106059 and #2103489. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. We thank the National Science Foundation Chameleon Cloud for providing time grants to access their resources. The technical support and advanced computing resources from University of Hawaii Information Technology Services Cyberinfrastructure, funded in part by NSF awards #2201428 and #2232862 are gratefully acknowledged.

References

[1] M. Atkinson, S. Gesing, J. Montagnat, I. Taylor, Scientific workflows: Past, present and future, Future Generation Computer Systems 75 (2017) 216–227.

[2] R. Ferreira da Silva, R. Filgueira, I. Pietri, M. Jiang, R. Sakellariou, E. Deelman, A Characterization of Workflow Management Systems for Extreme-Scale Applications, Future Generation Computer Systems 75 (2017) 228–238.

[3] R. Ferreira da Silva, H. Casanova, K. Chard, I. Altintas, R. M. Badia, B. Balis, T. a. Coleman, F. Coppens, F. Di Natale, B. Enders, T. Fahringer, R. Filgueira, G. Fursin, D. Garijo, C. Goble, D. Howell, S. Jha, D. S. Katz, D. Laney, U. Leser, M. Malawski, K. Mehta, L. Pottier, J. Ozik, J. L. Peterson, L. Ramakrishnan, S. Soiland-Reyes, D. Thain, M. Wolf, A Community Roadmap for Scientific Workflows Research and Development, in: 2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS), 2021, pp. 81–90.

[4] List of Workflow Systems, `https://s.apache.org/existing-workflow-systems` (2024).

[5] J. Liu, E. Pacitti, P. Valduriez, M. Mattoso, A Survey of Data-Intensive Scientific Workflow Management, J. Grid Comput. 13 (4) (2015) 457–493.

[6] L. Versluis, A. Iosup, A survey of domains in workflow scheduling in computing infrastructures: Community and keyword analysis, emerging trends, and taxonomies, Future Generation Computer Systems 123 (2021) 156–177.

[7] J. Liu, S. Lu, D. Che, A Survey of Modern Scientific Workflow Scheduling Algorithms and Systems in the Era of Big Data, in: 2020 IEEE International Conference on Services Computing (SCC), 2020, pp. 132–141.

[8] R. Nallakumar, K. Sruthi Priya, A Survey on Deadline Constrained Workflow Scheduling Algorithms in Cloud Environment, International Journal of Computer Science Trends and Technology 2 (5) (2014) 44–50.

[9] L. K. Arya, A. Verma, Workflow scheduling algorithms in cloud environment - A survey, in: Proc. of Conf. on Recent Advances in Engineering and Computational Sciences, 2014.

[10] L. Singh, S. Singh, A Survey of Workflow Scheduling Algorithms and Research Issues, International Journal of Computer Applications 74 (15) (2013) 21–28.

[11] M. A. Rodriguez, R. Buyya, A taxonomy and survey on scheduling algorithms for scientific workflows in iaas cloud computing environments, Concurrency and Computation: Practice and Experience 29 (8) (2017) e4041.

[12] A. Gupta, R. Garg, Workflow scheduling in heterogeneous computing systems: A survey, in: 2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN), IEEE, 2017, pp. 319–326.

[13] M. Adhikari, T. Amgoth, S. N. Srirama, A survey on scheduling strategies for workflows in cloud environment and emerging trends, ACM Computing Surveys (CSUR) 52 (4) (2019) 1–36.

[14] O. Sinnen, Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing), Wiley-Interscience, USA, 2007.

[15] K. Deng, J. Song, K. Ren, A. Iosup, Exploring portfolio scheduling for long-term execution of scientific workloads in IaaS clouds, in: Proc. International Conference on High Performance Computing, Networking, Storage and Analysis, 2013, pp. 1–12.

[16] H. Casanova, Y. Wong, L. Pottier, R. Ferreira da Silva, On the Feasibility of Simulation-driven Portfolio Scheduling for Cyberinfrastructure Runtime Systems, in: Proc. of the 25th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), 2022.

[17] É. Gaussier, J. Lelong, V. Reis, D. Trystram, Online Tuning of EASY-Backfilling using Queue Reordering Policies, IEEE Transactions on Parallel and Distributed Systems 29 (10) (2018) 2304–2316.

[18] A. Boulmier, I. Banicescu, F. M. Ciorba, N. Abdennadher, An Autonomic Approach for the Selection of Robust Dynamic Loop Scheduling

Techniques, in: 2017 16th International Symposium on Parallel and Distributed Computing (ISPDC), 2017, pp. 9–17.

[19] A. Mohammed, J. H. M. Korndörfer, A. Eleliemy, F. M. Ciorba, Automated Scheduling Algorithm Selection and Chunk Parameter Calculation in OpenMP, IEEE Transactions on Parallel and Distributed Systems 33 (12) (2022) 4383–4394.

[20] D. Carastan-Santos, R. Y. de Camargo, Obtaining Dynamic Scheduling Policies with Simulation and Machine Learning, in: Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, Association for Computing Machinery, New York, NY, USA, 2017.

[21] D. Talby, D. Feitelson, Improving and stabilizing parallel computer performance using adaptive backfilling, in: Proc. 19th IEEE International Parallel and Distributed Processing Symposium, 2005.

[22] N. Sukhija, B. Malone, S. Srivastava, I. Banicescu, F. M. Ciorba, Portfolio-Based Selection of Robust Dynamic Loop Scheduling Algorithms Using Machine Learning, in: Proc. IEEE International Parallel Distributed Processing Symposium Workshops, 2014, pp. 1638–1647.

[23] A. Mohammed, F. M. Ciorba, SimAS: A simulation-assisted approach for the scheduling algorithm selection under perturbations, Concurrency and Computation: Practice and Experience 32 (2019).

[24] A. Streit, The self-tuning dynP job-scheduler, in: Proc. 16th International Parallel and Distributed Processing Symposium, 2002.

[25] K. Deng, R. Verboon, K. Ren, A. Iosup, A Periodic Portfolio Scheduler for Scientific Computing in the Data Center, in: Job Scheduling Strategies for Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 156–176.

[26] D. Feitelson, M. Naaman, Self-tuning systems, IEEE Software 16 (2) (1999) 52–60.

[27] A. Nazarenki, O. Sukhoroslov, Using Simulation to Improve Workflow Scheduling in Heterogeneous Computing Systems, in: Proc. of Russian Supercomputing Days, 2017, pp. 480–490.

[28] E. Pérez, A simulation-driven online scheduling algorithm for the maintenance and operation of wind farm systems, SIMULATION 98 (1) (2021) 47–61.

[29] M. Tikir, M. Laurenzano, L. Carrington, A. Snavely, PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications, in: Proc. of the 15th Intl. Euro-Par Conf. on Parallel Processing, no. 5704 in LNCS, Springer, 2009, pp. 135–148.

[30] T. Hoefler, T. Schneider, A. Lumsdaine, LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model, in: Proc. of the ACM Workshop on Large-Scale System and Application Performance, 2010, pp. 597–604.

[31] R. Buyya, M. Murshed, GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing, Concurrency and Computation: Practice and Experience 14 (13-15) (2002) 1175–1220.

[32] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, R. Buyya, CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms, Software: Practice and Experience 41 (1) (2011) 23–50.

[33] A. Núñez, J. Vázquez-Poletti, A. Caminero, J. Carretero, I. M. Llorente, Design of a New Cloud Computing Simulation Platform, in: Proc. of the 11th Intl. Conf. on Computational Science and its Applications, 2011, pp. 582–593.

[34] G. Kecskemeti, DISSECT-CF: A simulator to foster energy-aware scheduling in infrastructure clouds, Simulation Modelling Practice and Theory 58 (2) (2015) 188–218.

[35] A. W. Malik, K. Bilal, K. Aziz, D. Kliazovich, N. Ghani, S. U. Khan, R. Buyya, Cloudnetsim++: A toolkit for data center simulations in omnet++, in: Proc. of the 2014 11th Annual High Capacity Optical Networks and Emerging/Enabling Technologies (Photonics for Energy), 2014, pp. 104–108.

[36] T. Qayyum, A. W. Malik, M. A. Khan Khattak, O. Khalid, S. U. Khan, FogNetSim++: A Toolkit for Modeling and Simulation of Distributed Fog Environment, IEEE Access 6 (2018) 63570–63583.

[37] H. Casanova, A. Giersch, A. Legrand, M. Qinson, F. Suter, Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms, Journal of Parallel and Distributed Computing 75 (10) (2014) 2899–2917.

[38] C. D. Carothers, D. Bauer, S. Pearce, ROSS: A High-Performance, Low Memory, Modular Time Warp System, in: Proc. of the 14th ACM/IEEE/SCS Workshop of Parallel on Distributed Simulation, 2000, pp. 53–60.

[39] H. Casanova, R. Ferreira da Silva, R. Tanaka, S. Pandey, G. Jethwani, W. Koch, S. Albrecht, J. Oeth, F. Suter, Developing Accurate and Scalable Simulators of Production Workflow Management Systems with WRENCH, Future Generation Computer Systems 112 (2020) 162–175.

[40] P. Velho, L. Mello Schnorr, H. Casanova, A. Legrand, On the Validity of Flow-level TCP Network Models for Grid and Cloud Simulations, ACM Transactions on Modeling and Computer Simulation 23 (4) (2013).

[41] SimGrid Use by Others , Available at `https://simgrid.org/usages.html` (2024).

[42] I. Colonnelli, B. Cantalupo, I. Merelli, M. Aldinucci, StreamFlow: Cross-Breeding Cloud With HPC, IEEE Transactions on Emerging Topics in Computing 9 (4) (2021) 1723–1737.

[43] M. Rocklin, Dask: Parallel computation with blocked algorithms and task scheduling, in: Proceedings of the 14th Python in science conference, no. 130-136, 2015.

[44] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, K. Chard, Parsl: Pervasive Parallel Programming in Python, in: 28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC), 2019, babuji19parsl.pdf.

[45] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, K. Wenger, Pegasus: a Workflow Management System for Science Automation, Future Generation Computer Systems 46 (2015) 17–35.

[46] J. McDonald, M. Horzela, F. Suter, H. Casanova, Automated Calibration of Parallel and Distributed Computing Simulators: A Case Study, in: Proc. of the 25th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC), 2024.

[47] Reproducible Research for FGCS manuscript, `https://github.com/wrench-project/fgcs2024_manuscript_reproducible_research` (2024).

[48] The WRENCH Project, `http://wrench-project.org/` (2024).

[49] The SimGrid Project, `http://simgrid.org/` (2024).

[50] The Grid'5000 Testbed, `https://www.grid5000.fr` (2022).

[51] WfCommons: Community Framework for Enabling Scientific Workflow Research and Development, `https://wfcommons.org` (2024).

[52] T. Coleman, H. Casanova, L. Pottier, M. Kaushik, E. Deelman, R. Ferreira da Silva, Wfcommons: A framework for enabling scientific workflow research and development, Future Generation Computer Systems 128 (2022) 16–27.

[53] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the April 18-20, 1967, spring joint computer conference, 1967, pp. 483–485.

[54] Y.-K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Comput. Surv. 31 (4) (1999) 406–471.

[55] J. Y.-T. Leung, Handbook of Scheduling : Algorithms, Models, and Performance Analysis, Chapman & Hall/CRC, 2004.

[56] R. Graham, E. Lawler, J. Lenstra, A. Kan, Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey, in: P. Hammer, E. Johnson, B. Korte (Eds.), Discrete Optimization II, Vol. 5 of Annals of Discrete Mathematics, Elsevier, 1979, pp. 287–326.

[57] R. Hall, A. L. Rosenberg, A. Venkataramani, A Comparison of Dag-Scheduling Strategies for Internet-Based Computing, in: 2007 IEEE International Parallel and Distributed Processing Symposium, 2007, pp. 1–9.

[58] M. Horzela, H. Casanova, M. Giffels, A. Gottman, G. Quast, S. Rissi Tisbeni, A. Streit, F. Suter, Modelling Distributed Heterogeneous Computing Infrastructures for HEP Applications, in: 26th International Conference on Computing in High Energy & Nuclear Physics (CHEP), 2023.

[59] L. Eyraud-Dubois, A. Legrand, The Influence of Platform Models on Scheduling Techniques, in: Y. Robert, F. Vivien (Eds.), Introduction to Scheduling, CRC Press, 2009, Ch. 11, pp. 281–309.

[60] M. Turilli, M. Santcroos, S. Jha, A Comprehensive Perspective on Pilot-Job Systems, ACM Comput. Surv. 51 (2) (2018) 43:1–43:32.