

Enabling Workflow Aware Scheduling on HPC systems

Gonzalo P. Rodrigo*, Erik Elmroth, Per-Olov Östberg, Lavanya Ramakrishnan⁺

Dept. Computing Science, Umeå University, SE-901 87, Umeå, Sweden

Lawrence Berkeley National Lab, 94720, Berkeley, California⁺

{gonzalo,elmroth,p-o}@cs.umu.se

lramakrishnan@lbl.gov

ABSTRACT

Workflows from diverse scientific domains are increasingly present in the workloads of current HPC systems. However, HPC scheduling systems do not incorporate workflow specific mechanisms beyond the capacity to declare dependencies between jobs. Thus, when users run workflows as sets of batch jobs with completion dependencies, the workflows experience long turn around times. Alternatively, when they are submitted as single jobs, allocating the maximum requirement of resources for the whole runtime, they resources, reducing the HPC system utilization.

In this paper, we present a workflow aware scheduling (WoAS) system that enables pre-existing scheduling algorithms to take advantage of the fine grained workflow resource requirements and structure, without any modification to the original algorithms. The current implementation of WoAS is integrated in Slurm, a widely used HPC batch scheduler. We evaluate the system in simulation using real and synthetic workflows and a synthetic baseline workload that captures the job patterns observed over three years of the real workload data of Edison, a large supercomputer hosted at the National Energy Research Scientific Computing Center. Finally, our results show that WoAS effectively reduces workflow turnaround time and improves system utilization without a significant impact on the slowdown of traditional jobs.

1 INTRODUCTION

In recent years, we have seen an increase in the processing of large amounts of scientific data and high-throughput processing at HPC centers. These scientific workloads are changing the landscape of software ecosystems on HPC centers that have traditionally supported large communication-intensive MPI jobs. The scientific workloads are increasingly composed as scientific workflows with complex dependencies.

The HPC batch schedulers still operate with a job-centric view and do not account for the complexities and dependencies of scientific workflows. Scientific workflow tools present in HPC centers often run workflows as chained jobs (jobs with dependencies) or as a pilot job (a single job containing the entire workflow). These approaches both have their drawbacks. Workflows run as chained jobs have very long and unpredictable turnaround times as they include the intermediate wait times for each job in the critical path. Workflows run as pilot jobs are likely to have shorter turnaround time, since the intermediate tasks do not wait for resources. However, they allocate the maximum required resources over the length

of the workflow for its entire runtime. Thus, resources are wasted as they are allocated but idle in parts of the workflow.

Scientific workflows also provide a unique opportunity for future HPC scheduling and resource management systems. Scientific workflows include detailed knowledge of the complex pipeline and dependencies between the tasks that can be used to gain efficiency in the system. In this work, we present the design and implementation for extending existing batch schedulers with workflow awareness. Our workflow aware scheduling system (WoAS) takes advantage of dependencies to achieve short turnaround times while not wasting resources. WoAS enables pre-existing scheduling algorithms to be aware of the fine grained workflow resource requirements and structure, without any modification to the original scheduling algorithms. WoAS uses the knowledge of the workflow graph to schedule individual tasks while minimizing the wait times for the jobs.

We implement WoAS within Slurm, a common HPC workload manager. Execution of diverse workflow workloads was simulated over a model of a real system (NERSC's Edison) [18], [1]. In our evaluation, we compare its performance with the chained and pilot job approaches. The experiment set is composed of 271 scenarios, covering different workflow types and submission patterns. Simulated time accounts for 253,484 hours (29 years) of system time and 3.8 million compute core-years.

Our experiments show that in most workloads run with WoAS, workflows show significantly shorter turnaround times than the chained job and single job approaches without wasting resources. The impact on non-workflow jobs was minimal except for workloads heavily dominated by very large workflows where performance limitations of the backfilling (queue depth limit) interfered with their scheduling.

Specifically, in this paper, our contributions are:

- We design a workflow aware scheduling system and algorithms that produce turnaround times with almost no intermediate wait times and wastage of resources.
- We present the WoAS implementation and its integration with Slurm. The WoAS system will soon to be made available open source.
- We evaluate and present the results of a detailed comparison of the workflow performance and system impact of WoAS, the pilot job, and the chained job approaches for diverse workflow workloads simulated over the model of Edison, a supercomputing system at the National Energy Research Scientific Computing Center (NERSC).

The rest of the paper is organized as follows. In Section 2, we present the life cycle of workflows and current scheduling approaches. The

* Work performed while working at the Lawrence Berkeley National Lab.

HPDC '17, Washington D.C., USA

2016. 123-4567-24-567/08/06...\$15.00

DOI: 10.475/123.4

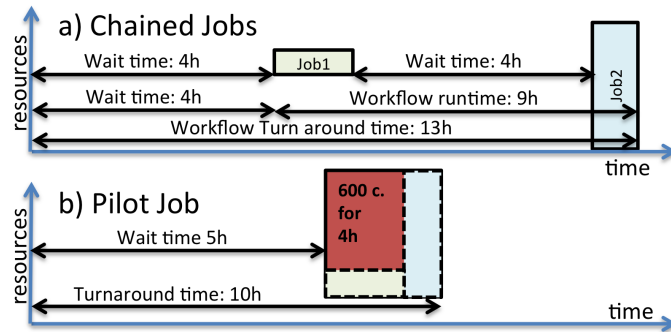


Figure 1: Cybershake workflow executed using chained and pilot job approaches. The chained jobs approach increases execution time due to the wait times. In the single job, there are no intermediate wait times but it wastes 600 cores for 4h.

Workflow Aware Scheduling technique (WoAS) is discussed in Section 3. We present the methodology to compare WoAS to the existing workflow scheduling methods in Section 4 and experimental results in Section 5. We discuss our conclusions in Section 6.

2 BACKGROUND

In this section, we describe the state-of-art and challenges of managing scientific workflows on HPC systems and discuss related work.

2.1 A workflow's life-cycle

Workflows are represented as Directed Acyclic Graphs (DAG) i.e., each vertex represents one or more work tasks, and the edges express control or data dependencies between the (vertices) tasks.

The first step to run a workflow on an HPC system is to map the DAG into one or more batch jobs, while respecting the data and control dependencies expressed by the edges. Users manually do the mapping or rely on workflow managers [26], which might also automate the submission and control of the workflow job(s). There are different mapping techniques governed by targets such as minimizing cost [27], minimizing runtime and turnaround time [4, 25], or tolerating faulty, distributed resources [17].

Once the execution plan is defined as a list of jobs and dependencies, users usually follow one of two strategies to submit a workflow. The strategies balance between lower resource consumption (as *chained jobs*) or shorter turn around time (as a *pilot job*). Figure 1 illustrates these approaches for the Cybershake workflow ([3]), which is used to simulate geological structures to characterize earthquake hazards in a region.

2.1.1 Workflow as chained jobs. In this approach, one batch job is submitted per execution plan job. Current batch schedulers allow users to specify dependencies between batch jobs. The scheduler then forces jobs to wait to start until the completion of its dependencies. Alternatively, users or their workflow engines might submit a job when the ones it depends upon have completed. Each job receives the exact amount of resources required to run and no allocated hardware resources are intentionally left idle. However, the workflow runtime will include the wait times endured by each of the jobs in the workflow's critical path. As described in Section 2.2, job priority systems do not consider a job until its

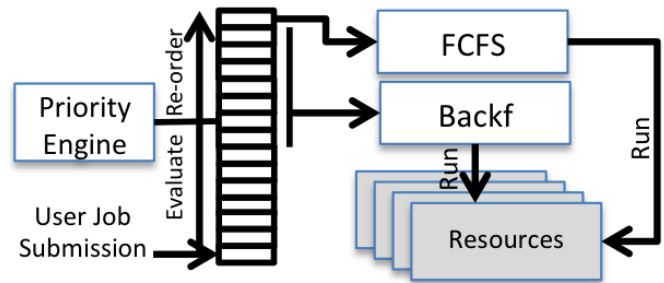


Figure 2: Classical batch scheduler with the waiting job queue in its center, which jobs are ranked by the priority engine and scheduled by FCFS and backfilling.

dependencies are resolved. As a consequence, job wait times for explicit dependencies are equal to the ones observed by submitting jobs when their dependencies are solved.

Figure 1a shows Cybershake's execution plan submitted as chained jobs. Even if both jobs are submitted at the same time, *Job2* has to wait four hours from the point *Job1* is completed, as its priority (and thus position in the waiting queue) does not increase from its initial value until its dependencies are resolved. The result is that the workflow runtime (i.e., start of first job to completion) is nine hours, wait time is four hours (44%), while the turnaround time (i.e., from job submission to completion) is 13 hours, eight hours accounts for wait time (61%).

2.1.2 Workflow as a pilot job. In this approach, the execution plan is submitted as a single job. The job's time limit is set to the expected runtime of the critical path with no intermediate job wait time. The job's resource request is the maximum resource allocation needed at any point in the workflow. As a consequence, the runtime of the workflow is the minimum possible, but some allocated resources might be idle, and thus wasted.

Figure 1b presents Cybershake's execution plan submitted as a pilot job. The workflow wait time is larger than the one faced by the chained job approach. However, the runtime is the minimum possible as there is no intermediate wait time. In this case, the wait time for the pilot job to start is smaller than the wait time for the two jobs in the chained job approach. However, during the first four hours of the workflow 600 CPU cores are allocated to the workflow but left unused, totaling 2400 idle core hours. This is the caveat of this approach, turnover is better but for a higher cost of consumed resources over the same work. This approach would work well for workflows where the difference between the minimal and maximum width of the workflow is not significant.

2.2 Classic HPC scheduling systems

Figure 2 presents the schema of a classic HPC scheduler with a queue of waiting jobs in its center: a) Jobs are inserted in a queue when users submit them. b) Scheduling algorithms select which jobs should start running and extract them from the queue. A classical HPC batch scheduler incorporates at least the following scheduling algorithms: FCFS (First-Come, First-Served) [6], running the first job of the queue if enough resources are available; and backfilling [11], scanning jobs in the queue in order to run them if enough resources are available and if they would not delay the start time of

previous jobs in the queue. Combined, they achieve high utilization and a reasonable job turnaround times in HPC systems.

Schedulers also include internal prioritization engines to manage turnaround time by ranking waiting jobs following some administrator set policy. Backfilling algorithms try to schedule sooner those jobs with a higher rank (priority).

Batch schedulers also understand job dependencies (relationships between jobs), and, among them, the most common one, enforces that a job cannot start until n previous jobs end successfully. Dependencies can be used to express the structure of the jobs within a workflow. However, we observed that in existing HPC schedulers dependencies affect the job priority calculation. For example, job age priority engines consider that a job age starts when its dependencies are resolved. In such schema, the submission time of the job will be its dependency resolution time, no matter when it was submitted. In Figure 1a we show the impact of this policy on workflow's turnaround time.

2.3 Related work

Complex experiments in scientific fields like high-energy physics, geophysics, climate study, or bioinformatics, require distributed resources as computational devices, data-sets, applications, and scientific instruments. The orchestration of such processes is organized as scientific workflows: collections of tasks structured by their control and data dependencies [26]. Distributed scientific workflows have been explored in detail in the last few years. Due to location specific or large resource requirements, a large portion of the workflows are distributed [16], i.e. their tasks run and data stored in different compute centers. In such environment, their execution depends on user inputs, specific resource characteristics, and run-time resource availability variations [9].

In other cases, workflows are run within the same compute facilities (single site workflows), however their tasks might be too large, or require too different resource sets that force to run them as different entities. This work focuses on the scheduling of the jobs of single site workflows.

Scheduling, automation, and execution systems for scientific workflows has been largely studied. Pegasus [4], Askalon [5], Koala [14], and VGrADS [17] are examples of Grid workflow managers that includes different approaches to workflow mapping, meta-scheduling, execution, task management, monitoring, and fault tolerance. However, they do not propose specific solutions to schedule the jobs of a workflow inside each of the Grid site, which is responsibility of the site scheduler.

There is also work on specific Grid workflow scheduling algorithms like: *Myopic* [25], *Min-Min* [13], *Max-Min* [13], *Sufferage* [13], Heterogeneous-Earliest-Finish-Time (*HEFT*) algorithm [24], or *Hybrid* [20]. These algorithms schedule jobs within, between, or across workflows under different strategies and objective functions. However they rarely schedule regular jobs and workflow jobs together, which is the main focus of this work.

Scientific workflow management systems for high throughput application have become more popular in the last years. Fireworks [8], QueueDO (QDO) [2], Falkon [15], and Swift [29] offer tools for workflow composition and management, execution, job packing of tasks (serial, OpenMP, MPI, and hybrid), and monitoring. These

systems may deploy their own execution frameworks or run their task in workers packed inside HPC jobs which, in the end are submitted as a pilot job or chained jobs.

Finally, data intensive and streaming workflows have become very important for the data processing within large IT companies. Frameworks like Hadoop [22], Spark [28], or Heron [10] offer workflow composition, management, and automation.

Clusters with batch jobs and services present the challenge of scheduling different workloads which metrics that cannot be compared. In that context, multilevel scheduling approaches have appeared allowing independent schedulers for different workloads (Mesos [7]), smart resource managers (Omega [21]), or cloud inspired two level scheduling for HPC systems (A2L2 [19]).

3 WORKFLOW AWARE SCHEDULING

Workflow Aware Scheduler (WoAS) provides an interface that allows users to submit *workflow jobs*. As illustrated in Figure 3, a user submits a *workflow job* that is a batch job that includes a manifest describing its internal workflow structure (e.g., two task jobs in the example). The workflow job is stored in the system's waiting queue.

There are three separate threads that work on the waiting queue - WoAS, the scheduler, and the priority engine. WoAS is always activated between the scheduler and priority. Thus the order of execution would be [WoAS, Scheduler, WoAS, Priority Engine]. When WoAS acts before the scheduler, it substitutes each workflow aware job by the task jobs described in its manifests, configuring the corresponding dependencies, and placing them in the same position of the queue as the original job. The resulting version of the queue is the *scheduler view* of the queue.

Once the queue is transformed, the scheduling algorithms act on it. In our example, backfilling selects and starts the first task job of of the example workflow, allocating exactly the resources that it requires. After the scheduling phase is over, WoAS transforms the waiting queue, removing the task jobs of workflows that have not started and restoring the corresponding workflow-aware jobs. The current state of the queue is the *priority view* of the queue. The priority engine periodically processes the waiting queue, calculating the priority of each job and ordering jobs accordingly.

The system continues repeating the cycle of a) recalculating the jobs priority b) transforming the queue into its scheduler view c) doing a scheduling pass (scheduling the second job of the example workflow when the first had completed) d) restoring the queue to its priority view.

In this section, we describe the steps of this process in detail.

3.1 Workflow job submission

In WoAS, users submit a workflow as a *workflow aware job*. This is an extension of the way workflows are represented in the pilot job approach. Users submit a job allocating the maximum resources required in the workflow for the minimum duration of its critical path, similar to a pilot job. However, a manifest describing the workflow is attached to the batch script.

Figure 4 is an example workflow description in JSON format for the LongWide workflow (defined in Table 2). It contains the definition of all the tasks within the workflow, including their

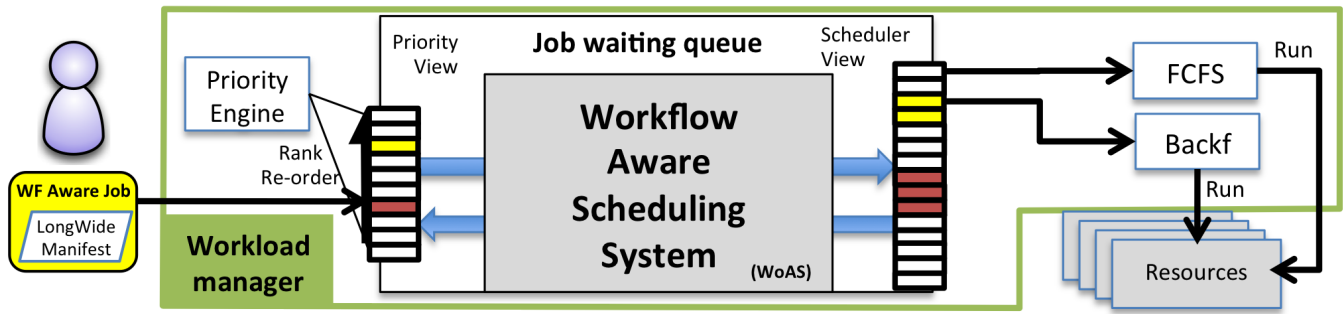


Figure 3: A workflow in WoAS scheduling model from its submission to execution start.

```

1 {"tasks": [
2   {"id": "SLong", "cmd": "./SLong.py",
3     "cores": 48, "runtime": 14400.0},
4   {"id": "SWide", "cmd": "./SWide.py",
5     "cores": 480, "runtime": 3600.0,
6     "deps": [{"SLong"}]}

```

Figure 4: LongWide workflow manifest in JSON format.

resource allocation requirement (allocated CPU cores for an estimated runtime), command or application to be executed (cmd), and dependencies with other tasks (deps, where SWide depends on completion of SLong). This manifest information is used by WoAS to transform a workflow aware job (priority view) into its task jobs (scheduler view).

3.2 Workflow Aware Scheduling system

The Workflow Aware Scheduling system (WoAS) is a job waiting queue model to bring workflow awareness to an HPC scheduler by offering different job lists (views) depending on the scheduler element that is interacting with the queue. The dual-view enables WoAS to enforce general scheduling behaviors such as the ones in Section 3.3, without requiring to change the code of the scheduler elements interacting with the queue.

WoAS controls the access to the waiting queue, and depending on the scheduler component interacting it presents two views:

The priority view. In this view, each workflow aware job is presented as a single job (the one submitted by the user). This view is the one presented to the priority engine. As a consequence, the priority and queue position of each workflow is based on the workflow aware job characteristics (submission time, geometry). All tasks in a workflow have the same priority or start with the same priority?

The scheduler view. In this view, each workflow aware job is present in the waiting queue through instances of its internal task jobs (and corresponding dependencies) placed in the same position of the queue where the original workflow aware job was. This view is the one presented to the scheduler algorithms, so they can schedule the workflow task jobs individually.

3.3 Workflow awareness in WoAS

In this section, we discuss the impact of the views model on the workflow awareness in the scheduler. Specifically, there are three behaviors. First, workflow task job level scheduling results in the allocated resources are the minimum possible. Second, the intermediate wait times are minimized to avoid the ones observed in the chained job approach. Finally, this minimizes system gaming

where users don't have to ask for strange resource requests to make sure the tasks in their workflows get the correct priority.

Workflow awareness is consequence of the interaction of the scheduler with the views and the way job's priority information is transferred when the queue is transformed between views.

3.3.1 Workflow task level job scheduling. In WoAS, the scheduling algorithms schedule workflow task jobs, assigning the precise required resource allocation to each step of the workflow.

This is possible because the scheduling algorithms act on the scheduler view provided by WoAS. In opposition to the pilot job approach, even if workflows are submitted as a single job, WoAS ensures that the scheduling algorithms will see the workflows as their task jobs.

This characteristic is what allows WoAS not to waste resources to run workflows, even if they are submitted as single job.

3.3.2 Minimization of the intermediate wait times. With WoAS, when the first task job of a workflow is started during the scheduling view, the rest of the workflow task jobs remain related by their dependencies in the waiting queue. This situation is similar to how task jobs stay in the queue for the chained job approach.

However, in the chained job approach, intermediate wait times might be very long. Classical schedulers consider jobs with non resolved dependencies as not "submitted", so their priority does not increase as they wait. From the moment that jobs they depend on are completed, task jobs have to wait as they had been submitted, even it had been waiting in the waiting job queue for much longer time.

WoAS reduces intermediate wait times by propagating the priority attributes of the original workflow aware job to all its tasks. All the tasks have the same geometry priority factor and submit time as the original workflow aware job. As a consequence, under the scheduling view, all the tasks in a workflow are positioned in adjacent positions in the waiting queue. If the first task job starts, its queue position should be close to the top. As all the workflow task jobs have similar queue positions, once the first task job is completed, the following one will still be in a good queue position to be started. In such situations, the intermediate wait time should be close to the time until adequate resources for that task job are available. This time can be significantly shorter than the time that it would take for that task job to progress from the bottom to the top of the priority queue in a highly utilized system.

Propagation of the priority information is performed by a combination of the views and operations within WoAS.

A workflow aware job's priority information is set during the priority calculation under the priority view. In our system, the priority of a job depends on two factors:

1) Job's geometry factor, (smaller job, higher priority). It is calculated only once in the life of a job, in the first priority calculation process that considers it.

2) Job's age factor, (older job, higher priority) is recalculated in every priority calculation process. It depends on the time the job was submitted.

Algorithm 1 Show scheduler view actions.

```

1 def woas_show_scheduler_view():
2     global waiting_queue
3     for job in list(waiting_queue):
4         if is_workflow_aware_job(job):
5             remove_job(waiting_queue, job)
6             for task_desc in job.manifest["tasks"]:
7                 new_job = create_job(task_desc)
8                 new_job.prio.geometry = job.prio.geometry
9                 new_job.prio.age = job.prio.age
10                new_job.submit_time = job.submit_time
11                new_job.copy_wf_job = job
12                insert_job(waiting_queue, new_job)

```

The priority information of the workflow aware job is propagated to its task jobs through the operation `woas_show_scheduler_view`, which transforms the waiting queue into scheduler view. The detailed actions of this operation can be followed in Algorithm 1, where each task job receives the workflow aware job geometry factor, age factor, and submit time.

This propagation has three consequences for future priority calculations of all the task jobs of the same workflow. First, Future task job age factor calculations will be based on a workflow's submit time. Second, the geometry factor of a job is set, so the priority engine does not recalculate it. Thus, future task job priority calculations will be based on the workflow aware job's geometry, not its own. Finally, all task jobs of the same workflow will have the same priority (and the same the workflow aware job would have) since it has the same geometry factor and same submit time.

This ensures that all task jobs of the workflow will have the same priority and will occupy a similar position in the waiting queue, which leads to the minimization the intermediate wait time,

As a final note, the priority propagation of non started workflows is closed by the `woas_show_priority_view` operation. As it transforms the queue in its priority view, it enforces that if a workflow has not started, it becomes again the same workflow aware job, with the same priority factors.

3.3.3 Minimize system gaming. The priority propagation mechanism described in Section 3.3.2 has another side effect. Since task job's priority factors are the same as the ones of the workflow aware job, the waiting time of the first task job is equivalent to the waiting time of a job of the geometry and submission time of the workflow aware job.

This is a desired effect to stop users from gaming the system, i.e. users submit workflows where first job is very small, expecting a short wait time to then run larger task jobs. This takes advantage

of the short wait time minimization of WoAS: As the workflow wait time depends on the workflow aware geometry, such schema would only produce longer wait times.

3.4 Batch Scheduler integration

Algorithm 2 Simplified classical scheduler algorithm with WoAS calls to enable the views model.

```

1 def scheduling_loop_with_WoAS():
2     while True:
3         if time_to_check_priority():
4             do_priority_calculations()
5         if (time_to_do_fifo() or
6             time_to_do_backfilling()):
7             woas_show_scheduler_view() // WoAS specific
8         if time_to_do_fifo():
9             do_fifo_scheduling()
10        if time_to_do_backfilling():
11            do_backfilling_scheduling()
12        woas_show_priority_view() // WoAS specific

```

WoAS was incorporated to the scheduler by modifying the core batch scheduling loop. Algorithm 2 describes a simplified representation of such process, in which a phase where jobs priority is recalculated (line 4) alternates with another in which the scheduling algorithms act (lines 8-11). In such a model, introducing WoAS does not require changing the priority or schedulers behavior. It requires adding just two actions to the loop, as listed below.

1) adding `woas_show_scheduler_view` before the scheduling phase starts (line 7). This function transforms the waiting queue for the following code (scheduling phase) to see the waiting queue through the scheduler view.

2) adding `woas_show_priority_view` after the scheduling phase starts (line 12). This function restores the waiting queue into the priority view, so the priority actions (if executed), act over that view.

In Slurm, the priority and scheduling components run concurrently, and the queue exclusive access is enforced through a lock. WoAS is integrated by adding the `woas_show_scheduler_view` call just after the scheduling code acquires the queue lock. Next, it adds `woas_show_priority_view` just after the scheduling code frees the queue lock. This ensures a behavior equivalent to Algorithm 2.

4 METHODOLOGY

We evaluate the workflow aware system using a Slurm simulator and analyze the resulting scheduling logs. In this section, we describe our simulator setup, metrics, and experiment definitions.

4.1 System

NERSC's Edison is the reference system chosen to be emulated and to model the baseline workload. Edison is a Cray XC30 supercomputer, with 6,384 nodes, 24 cores per node, and a total of 133,824 cores and 357 TB of RAM, installed in 2014. It uses an Aries interconnect and can produce a peak of 2.57 PFLOPS/s. Edison's hardware and workload are representative of systems and applications present in the high performance scientific community.

4.2 Simulation framework

We implemented WoAS in Slurm, since it is increasingly used in high performance systems. This functional implementation of a WoAS enabled Slurm will be distributed as open source. Also, previous work by the Barcelona Supercomputing Center (BSC) [12] and the Swiss Supercomputing Center (CSCS) [23] provided a Slurm simulator base code. We extended the simulator for our experiments. The simulator allows us to run experiments up to 20x faster than real time and run multiple simulations in parallel (up to 200).

The Slurm scheduler is configured similar to current HPC systems and uses FIFO, backfilling, and it gives higher priority to smaller jobs. However, to reduce complexity of the experiments and ease analysis, differentiated queues or QoS levels are not configured in our simulator. These features provide user-level conveniences and will translate to the workflow awareness and are not central to the focus of our experiments.

The core of our simulator is the Slurm scheduler. Slurm is configured to use the desired scheduling method (chained jobs, pilot job, or workflow aware). After configuration, the simulator starts Slurm and submits the workload to it, emulating the user behavior. The scheduling process is run for a configured simulation time (5 days plus an extra for cold start stabilization) and the scheduler logs are registered in a MySQL database for later analysis.

An experiment is defined by its workload characteristics, a scheduling method choice, a simulated system configuration, a target simulated time, and a random seed. To run an experiment, the workload is generated according to the workload characteristics. Workflow characteristics include the characteristics of the real HPC system workload after regular jobs are modeled, a list of specific workflows present in the workload, and their submission patterns.

Finally, each experiment is repeated using six different random seeds (producing different workloads) and their results aggregated to ensure that analyses are not based on single non representative experiments.

4.2.1 Workload generation. Each experiment has a workload composed of regular (non workflow) synthetic jobs and workflow jobs modeled after the experiment configuration. The regular jobs in our workload traces are modeled after the historical traces from three years of NERSC's Edison system, [18] and [1].

The experiment configuration defines the specific workflows present in the workload, the job format for the workflow (a pilot job, chained jobs, or a job including a workflow manifest), and the submission pattern. Our simulator supports two workflow submission patterns - workflow periodic and workflow share. In the periodic one, a workflow is submitted once every configured time period. In the share model, workflows are submitted at a uniform pace so the number of allocated core hours to workflows represents a desired share of the total core hours of the workload.

The workload generator also includes a mechanism to pre-fill the system to capture a typical state of a supercomputer system. The lengths of the jobs for pre-fill stage are configured to obtain a job wait time baseline of four hours. Also, a job pressure control mechanism adjusts the job and workflow submissions so the workload job pressure (submitted core hours over system capacity in a time period), is slightly over 1.0. This ensures that simulated

system will have enough pending work to support the wait time baseline, but with not too much to significantly increase the job wait time as the workload scheduling progresses. Also, the simulator uses a system cold-start stabilization period of one day. This workload is not representative of a regular day systems operation and is discarded for the analyses.

Finally, the workload generator uses a random number generator that can be initialized with a seed. The same seed always produces the exact same regular jobs and workflow submission times, independently of the workflow scheduling system chosen (as long as the same workload configuration is used). This is used to do a fair comparison between different scheduling techniques for the same experiment configuration.

The described workload analysis and modeling tools; the workload generator; the framework to define and run experiments; the tools to process and analyze experiment results; and the improvements on the Slurm simulator, were developed in the context of this work.

4.3 Evaluation metrics

In this section, we present the metrics used to compare experiment results and the method to calculate them.

4.3.1 Performance metrics. In our analyses, we use three workflow (wait time, run time and turnaround time) and two system (system utilization, job slowdown) performance metrics to compare the pilot job, chained job, and WoAS approaches.

Workflow wait time (w^W) is the time between the submission of the first job of the workflow and its execution start. Smaller wait times are preferred. It depends on the load in the system (waiting work vs. compute capacity, with higher loads implying overall longer waiting times), the geometry (smaller jobs tend to wait less due to backfill), and priority (higher tends to imply shorter wait time).

Workflow runtime (r^W) is the time between the execution start of the first job and the execution completion of the last job of the workflow. It includes the runtime of the jobs in the critical path of the plan and the wait time between them. Smaller runtimes indicate lesser waste between the tasks of the job. Minimum possible workflow runtime is the sum of the runtimes of the jobs in the critical path (as they run back to back).

Workflow turnaround time (t^W) is time between the submission of the first job and the execution completion of the last job of the plan. Smaller values are better. It is obtained as the sum of w^W and r^W , thus it depends on the factors the previous two depend on.

Actual utilization during a time period (t) is $\frac{\sum corehours_i^J - \sum waste_i^W}{cores^S * t}$, where $corehours_i^J$ are the core hours allocated by jobs and workflows that are executed, $waste_i^W$ is the number of core hours allocated by a workflow that are not assigned to an internal task or job where $cores^S$ are the number of cores of the compute system. This metric is a variation of the classical utilization that takes into account that workflows might allocate resources but not use them through all their runtime. It measures the actual work done over the system capacity, not just allocation. This is relevant to measure since pilot job workloads might show a

Group	A: Workflow critical path length.	B: Allocated cores, overall vs 1st job.	C: Alloc. cores and rtime, overall vs first job.
Geometry	$n \text{ jobs}/\text{rtime}:n \text{ h}/\text{max } 240 \text{ core}$	$2\text{jobs}/\text{rtime}:2n \text{ h}/\text{max } 240n \text{ cores}$	$2\text{jobs}/\text{rtime}:2n \text{ h}/\text{max } 240n \text{ cores}$
Usage/Waste	$240n \text{ core-h} / 0 \text{ core-h.}$	$240 + (n) * 240 \text{ core-h.} / (n - 1) * 240 \text{ core-h}$	$240 + n(2n - 1) * 240 \text{ core-h.} / n(n - 1) * 240 \text{ core-h.}$

Table 1: Workflows characteristics for workflow groups: critical path size, pilot job geometry (runtime and max cores), workflow tasks usage (usage), potential wasted resources (waste), and a profile of the allocated resources in time if the critical path is run with no intermediate waits.

high theoretical classical utilization and hide the fact that resources might be allocated but not used.

Job's slowdown is measured as $\frac{r^J + w^J}{r^J}$, i.e. job's turnaround divided by its runtime. This metric allows us to compare the wait time from jobs with different runtimes. We use this metric to measure the impact of different workflow scheduling techniques on the non workflow jobs. We calculate this metric for non workflow jobs grouped in three different sizes ($[0, 48)$, $[48, 960)$, $[960, \infty)$ core hours). The median values of this metric for each job group are used in comparisons.

4.3.2 Metrics calculation. All the metrics of this work are obtained over the aggregation of the results of multiple repetitions of the same experiment. To keep the meaning of each metric, the aggregation method is different.

For the workflow performance metrics, the performance values of m_i first workflows of each repetition i were aggregated and then the percentile metrics calculated. m_i of a repetition of an experiment is the minimum number of workflows completed in the three versions (WoAS, pilot job, and chained job) of that repetition i . This pre-selection is required to compare similar datasets and these metrics cannot be calculated for incomplete workflows.

Actual utilization for an experiment is calculated as the mean of the observed actual utilization in the six repetitions. This is equivalent to calculating the utilization of an experiment which was the concatenation of the six repetitions. For the aggregated calculation of the job's slowdown, all the non workflow job slowdown values in the repetitions are read, and the percentile analysis is performed on them.

4.4 Experiment sets

Two experiments sets are analyzed in this work, studying the scheduling techniques from a more analytical and real point of view, resulting in 271 experiment configurations. Each individual experiment consists of five days of simulated scheduling of the workload plus an extra initial one for the system cold start.

4.4.1 Workflow characteristics study. In this experiment set, we analyzed the effect on the workflow metrics of using different scheduling techniques to run workflows with different internal characteristics. There is a workflow group for each workflow characteristic, and inside each group, a workflow is defined by n : a knob that controls the effect of the workflow characteristic, where a larger n implies a larger effect.

These experiments allow creating a base knowledge on the expected wait time, runtime, and turnaround times for some basic workflow characteristics. These are the workflow groups as presented in Table 1:

Workflow critical path length, Group A: The goal with this group of workflows is to study the effect on the workflow metrics of the number of tasks in the workflow critical path (n defines the number of those tasks). All workflows in this group are chained lists of n tasks of the same size (240 CPU cores and 1h runtime), e.g. if $n = 3$ the resulting workflow has three tasks, in which the second depends on the first and the third of the second. Workflows with a longer critical path should suffer: a) larger difference between runtime of the pilot job and first job in chained job and WoAS approaches. b) more intermediate wait time periods between the tasks in the chained job and WoAS approaches (workflow runtime related). c) lower priority for the pilot job and workflow aware job vs the priority of the first job in the chained job approach (workflow wait time related).

Allocated CPU cores: First job vs. workflow's maximum, Group B: Group B is used to study the effect on the workflow metrics of the difference between allocated cores for the first job and the workflow maximum (n is the difference multiplier controlling the breadth of the workflow). All workflows in this group are composed of two jobs, the first allocates 240 cores for one hour and the second allocates $n * 240$ cores for one hour. When combined with the workflow scheduling approaches, a higher number of n will induce two workflow wait time related effects: a) larger difference between the allocated cores by the pilot job and the first job in the chained jobs and WoAS approaches. b) lower priority for the pilot job and workflow aware job vs the priority of the first job in the chained job approach (workflow wait time related). The difference in priority is induced by the difference in resource allocation in opposition to the workflow runtime (Group A).

Allocated CPU cores and runtime: First job vs. workflow's maximum, Group C: This group is used to study the effect of the difference in allocated cores of the first job and the workflow maximum combined with the difference in runtime between the first job and the minimum critical path runtime on the workflow metrics (n is the difference multiplier controlling the length and breadth of the workflow). All workflows in this group are composed by two jobs: The first allocates 240 cores for one hour. The second allocates $n * 240$ cores for $2n - 1$ hours. When combined with workflow scheduling techniques, a higher number of n will induce two workflow wait time related effects: a) larger difference between

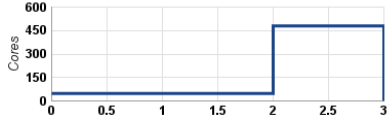
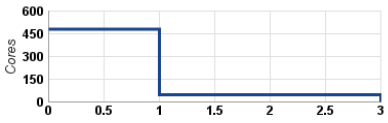
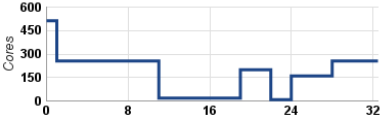
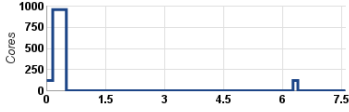
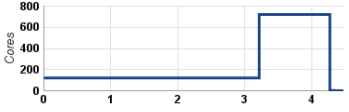

Workflow	LongWide	WideLong	Floodplain
Geometry	2jobs/rtime: 3h/480 max cores	2jobs/rtime: 3h/480 max cores	7jobs/rtime: 32.5h/512 max cores
Usage/Waste	672 core-h / 1728 core-h	672 core-h / 1728 core-h	5624 core-h / 11016 core-h
Profile			
Workflow	Montage	Cybershake	Sipht
Geometry	5jobs/rtime: 7.6h/960 max cores	5jobs/rtime: 4.5h/721 max cores	9jobs/rtime: 1.2h/384 max cores
Usage/Waste	375 core-h / 6920 core-h	1145 core-h / 2077 core-h	185 core-h / 395 core-h
Profile			

Table 2: Workflows characteristics for individual workflows including: critical path size, pilot job geometry (rtime and max cores), workflow tasks usage (usage), potential wasted resources (waste), and a profile of the allocated resources in time if the critical path is run with no intermediate waits.

the allocated cores and runtime for the pilot job and the first job in the chained jobs and WoAS approaches. b) lower priority for the pilot job and workflow aware job vs the priority of the first job in the chained job approach (workflow wait time related). The difference in priority is induced by the difference in resource allocation and workflow runtime.

In this experiment set, six workflows of each workflow group are defined ($n \in \{1, 2, 4, 8, 16, 32\}$, group C was not analyzed for $n > 8$, resulting workflows were too big and would overflow the system). For each individual workflow (16 in total), we create a workload in which a workflow is submitted with a fixed inter-workflow time. Each experiment is run using the pilot job, chained jobs, and WoAS techniques to compare the resulting metrics across techniques and values of n .

4.4.2 Performance comparison. In these experiment set we compared the performance of the different workflow scheduling techniques for two synthetic and four real workflows, which are presented in Table 2. The synthetic one (LongWide and WideLong) are the minimum building units of any workflow (a serial phase followed by a parallel one and vice-versa). The real ones allow testing our technique against more realistic workloads with has a particular characteristic: fixed jobs with a complex profile (Floodplain), many small grouped tasks (Montage), large workflow with two large parallel stages (Cybershake), and a small workflow with a complex profile shape and many small jobs (Sipht).

In the experiments, workflows are submitted using the workflow share approach with seven percentages: %1, %5 %10, %25, %50, %75, %100. The experiments with lower ones (%1 to %25) allow understanding the performance of the techniques of realistic scenarios with increasing workflow importance. The larger values (%50, %75, %100) allow understanding what happens in a system when the workload is dominated by workflows over regular jobs. The resulting 42 experiments are run using the pilot job, chained jobs, and WoAS techniques.

Similar experiments were run using the workflow period submission (periods 1/12h, 1/2h, 1/h, 2/h, 6/h). These allow comparing

the workflow metrics in cases in which the workflow presence is not important enough to influence in the whole system behavior.

5 RESULTS AND ANALYSIS

This section presents the results and analyses of our simulation experiments. Our evaluation focuses on: a) A study of the impact of workflow characteristics on the workflow metrics obtained with different workflow scheduling techniques (Section 5.1). b) A performance comparison for the different scheduling techniques (Section 5.2).

5.1 Workflow characteristics study

Figures 5, 6, and 7 present the observed median workflow wait times, runtime, and turnaround time for the experiments with workflows from Groups A, B, and C (described in Section 4.4.1). Each horizontal block corresponds to a different workflow group. Inside each block, adjacent bars represent the measured median value for the same experiment configuration but run with different scheduling approaches (pilot job, chained jobs, and WoAS). The x-axis corresponds to n , a value that defines the actual workflow used in each workflow group (Defined in Section 4.4.1). In each group a higher value of n indicates that the special characteristic of the workflow group is more present.

5.1.1 Workflow wait time. For **group A workflows** (top block of Figure 5), we observe that the relationship between the median wait times observed for the pilot job and WoAS approaches are similar, with slightly shorter wait times for WoAS at all the workflow path sizes (n). Any difference in workflow wait time are related to differences in backfilling eligibility since priority and CPU cores allocation of the pilot job and the first job are the same.

In contrast, the chained job workflow has the same FIFO and backfilling eligibility as WoAS (both first jobs have the same geometry), but higher priority (job size used for priority is bigger in WoAS). Hence, workflows run as chained jobs show much shorter wait time (almost half at $n = 32$) as the critical path and workflow aware job sizes increases and the priority gap increases.

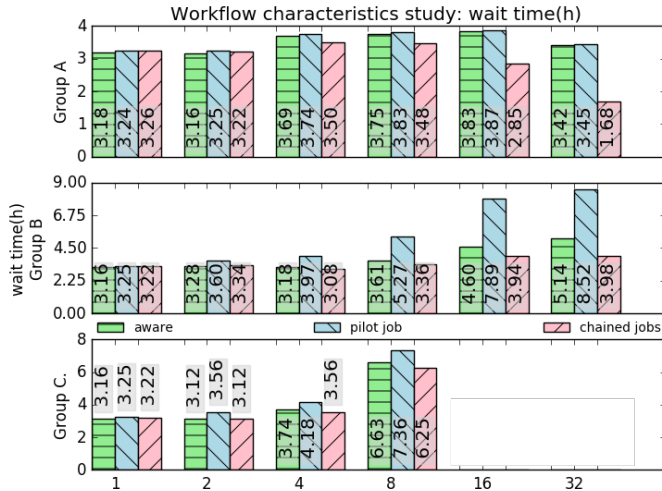


Figure 5: Wait time evolution as a dimension (one per horizontal group) of the workflow group is increased.

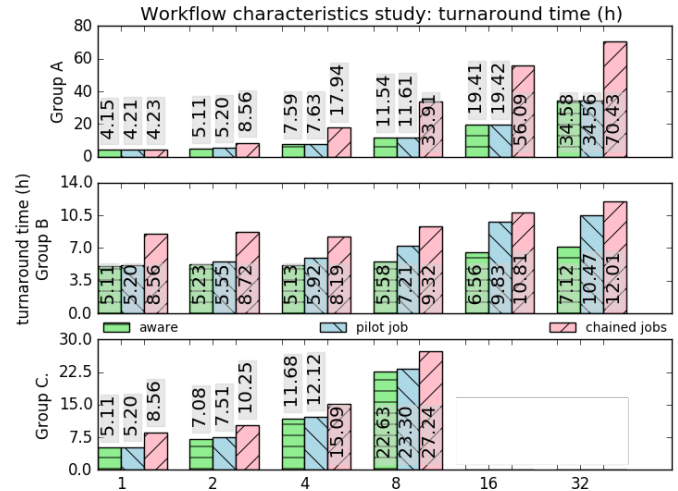


Figure 7: Turnaround time evolution as a dimension (one per horizontal group) of the workflow group is increased.

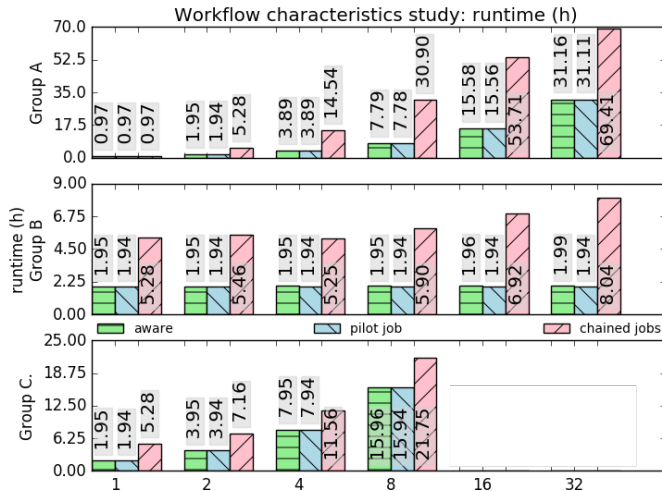


Figure 6: Runtime evolution as a dimension (one per horizontal group) of the workflow group is increased.

Similarly, workflows in groups B and C (second and third block of Figure 5) exhibit the shortest wait times when run as chained jobs, intermediate as WoAS, and much longer (specially for $n \geq 16$), as pilot jobs. The differences are due to the priority and backfilling eligibility of the workflow starting job in each group: higher priority, smaller job (more eligible) in chained job; lower priority, smaller job in WoAS; and lower priority, larger job (less eligible) in pilot job.

5.1.2 *Workflow runtime.* In Figure 6, we observe that all workflows run as pilot jobs or under WoAS present a very similar median workflow runtime, close to the expected minimum runtime for each value of n . This is expected for the pilot job, since all the tasks are run within a job with no internal wait times. We see that WoAS is able to perform as well; inter-job wait times between jobs when using WoAS is close to 0, e.g workflows in group A, $n = 32$, are

composed of 32 jobs (see Table 1) and the median of the accumulated 31 intermediate wait times accounts only for three minutes which constitutes 0.1% of the total runtime.

When run as chained jobs, group A workflows show longer accumulated inter-job wait time as the number of jobs in the workflow critical path increases. This matches the observation that most schedulers do not consider a job as truly submitted until its dependencies are resolved. Each extra job in the critical path adds an extra wait time to the runtime.

Similarly, runtime of workflows in groups B and C are the minimum possible when run as pilot jobs, and close to minimum when using WoAS. When run as chained jobs, the increasing runtimes show the effect of the wait time of the second job on the runtime: As n increases, the geometry of the second job grows (slower in B than in C) and its wait time becomes longer.

5.1.3 *Turnaround time.* In Figure 7, we observe that for all workflow groups the chained jobs approach presents the longest turnaround times; followed by the pilot job and WoAS approaches, which shows the shortest (or equal to pilot job).

Workflows run as a workflow aware job present bigger gains in turnaround time over the pilot job approach as they face significantly shorter wait times (in our experiments group B and $n \geq 4$).

5.1.4 *Summary.* We observed that running a workflow as chained jobs results in the shortest wait time but longest runtime. Running it as a single pilot job, produces the shortest runtime but longest wait time. WoAS produces intermediate wait times and close to shortest runtimes.

Also, results show that WoAS produces the best turnaround times in all the scenarios. Finally, a workflow aware job can be considered significantly better performing than the pilot job approach even if the turnaround time is similar, since the former does not waste resources for workflows with varied resource requirements (more in the next section).

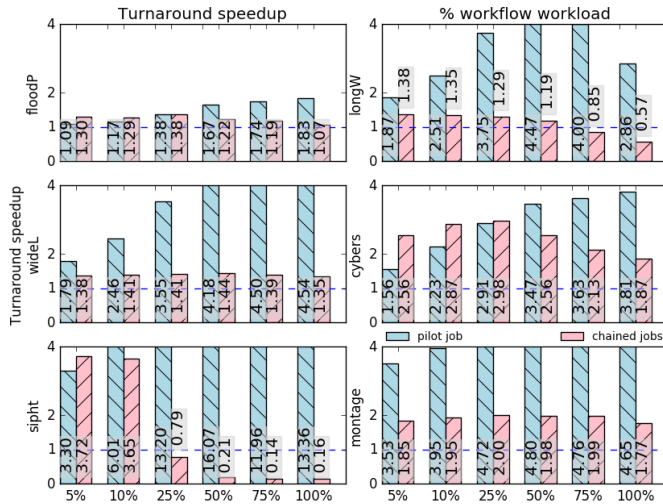


Figure 8: Workflows turnaround time speed up when scheduled as WoAS over pilot (blue) and chained job (pink). Six workflows, six different workflow shares. The dashed line is speedup=1, Value > 1 implies better performance of WoAS.

5.2 Performance comparison

In this section, we extend analyses in Section 5.1 to four real and two synthetic workflows. In this section we focus on workflow turnaround time, and the impact of the scheduling techniques over the system (actual utilization) and non-workflow job (slowdown) performance.

In this experiment set workflows are submitted using workflow share (described in Section 4.2.1) to set the percentage of workload core hours corresponding to workflows.

5.2.1 Workflow performance. Figure 8 presents the median workflow turnaround time speedup of WoAS relative to the chained jobs and pilot job approaches. The axis shows percentage of workload core hours contributed by workflows. A bar value $X = 1$ means that the median turnaround time for WoAS and the corresponding method are the same. A bar value $X > 1$, means that the median turnaround time median for the corresponding approach is X times the one observed with WoAS.

Compared to the chained job approach, WoAS showed very large speed ups for long complex workflows like Cybershake ($\approx 2x$) and Sipt ($\approx 3s$). For the rest of the workflows (shorter critical path), showed smaller but clear speedups in most cases (e.g $\approx 1.4x$ for WideLong, $\approx 1.9x$ in Montage). Floodplain has relatively small jobs reducing the effect of the intermediate wait times (1.2x-1.3x speedup).

Also, LongWide workflows showed shorter turnaround times when run as chained jobs in the 75% and 100% scenarios (< 1.0 speed ups). After analyzing system’s actual utilization and overall wait time, it was observed that, when using WoAS in the scenarios, the wait time baseline is more elevated, and utilization is lower (20% and 30% less). This is likely related to the workflow shape. The workflow consists of a long job (48 cores, 4 hours runtime) and a wide job (480 cores running, 1 hour) where the long job can become a barrier that cannot start until a number of previous jobs end. The long job also stops other jobs from being backfilled since

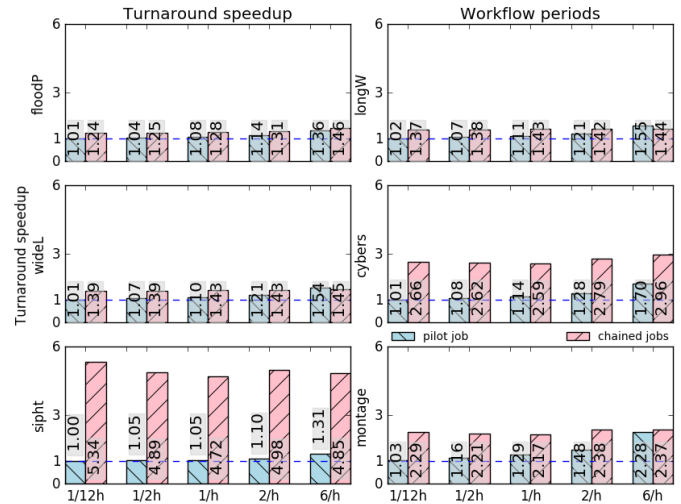


Figure 9: Workflows turnaround time speed up when scheduled with WoAS over the pilot (blue) and chained job (pink). Six workflows, six different workflow periods.

they would delay its start. This wait creates unused free resources gap that results in low utilization.

For workflow shares over 25%, Sipt experiments show turnaround speed-ups under one and get smaller as the workflow share increases. In these scenarios, the chained jobs approach achieved lower utilization values ($\approx 10\text{-}20\%$ less) and less workflows would complete than WoAS. Sipt is the workflow with more jobs, but very small resource allocation. In a workflow saturated scenario the scheduler manages a large number of active jobs, affecting its performance, and thus capacity to utilize the system. Since WoAS represents all workflows that have not started yet as a single job, the scheduler requirements to deal with such workload should be smaller.

Both situations are very unlikely in a real system, where the workflow includes different types of workflows and regular jobs, that can be used in efficient backfilling.

Compared to the pilot job approach, In Figure 8, WoAS shows turnaround times orders of magnitude shorter than the pilot job.

The long pilot job turnaround times are due to their long wait times. This is an artifact of the workload generation that is designed to contain the same amount of work i.e., workloads contain the exact same jobs and workflows, submitted at the same time. When run with the chained job technique, that work is meant to produce a job pressure of ≈ 1.0 over the system. However, when run with pilot jobs, the same work allocates more core hours because of the potentially wasted resources, increasing the job pressure. For example, in a 100% workflow share workload, the job pressure when workflows are pilot jobs is $1.0 + k$, being k the percentage of wasted core hours in the selected workflow).

In summary, for the same amount of work, the single job approach presents much longer turnaround times and loads the system significantly more than the other approaches (more in Section 5.2.3).

Finally, we assess if the previously observed short turnaround times for isolated workflows run as pilot jobs are possible for the workflows in this section. A set of experiments were performed, reducing the workflow presence in the workload by submitting

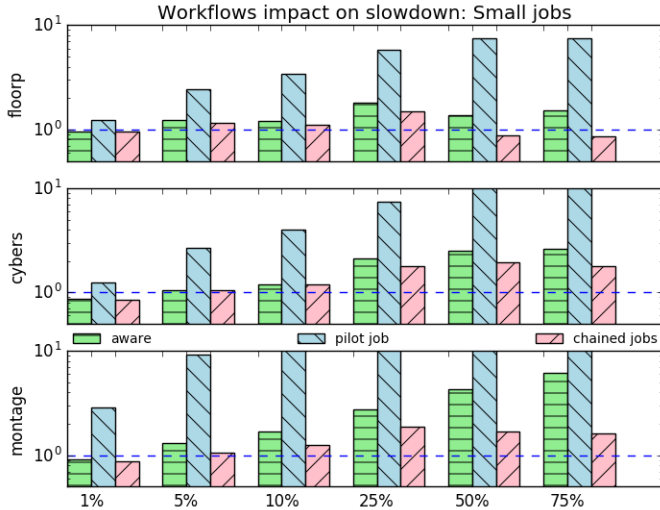


Figure 10: Relative difference on slowdown (withWorkflowSlow/noWorkflowSlow) for jobs allocating [0, 48) core hours. Tested for three real workflows and different workflow shares.

one workflow in every time period. In Figure 9, WoAS presents similar or shorter turnaround times than the pilot job approach, confirming that for isolated workflows, the pilot job approach also shows short turnaround times.

5.2.2 Job fairness. Figure 10 presents the observed median slowdown for small jobs (under 96 core hours) over different workflow shares (x-axis). A value of one means that the median slowdown for non workflow jobs is the same as in the case where no workflows are present. A number of two represents that the observed median slowdown is two times the one observed when no workflows are present. It is important to note that adding workflows changes the workload composition and small variations in the slowdown are considered normal.

Figure 10 shows that the presence of workflows affects the regular job’s slowdown. All experiments using the pilot job approach showed the biggest increases in slowdown (up to 10x), followed by WoAS (up to 8x), and the chained job ones (up to 2x). This difference is specially significant for Montage workflows run as pilot jobs, where just a 1% workflow share induces a three times bigger median slowdown, and almost 10 times slowdown with a 5% workflow share. The observed slowdown is due to the elevation of the wait time baseline due to the > 1.0 job pressure from the pilot jobs. In opposition, when the workload contained 1% of Montage workflows but were scheduled using WoAS there was no effect on non-workflow jobs.

The experiments run with chained job approach show the smallest changes in slowdown with maximum variations of 2x for over 50% workflow share scenarios. The chained job scenarios are the best possible fairness scenario for each workflow and workflow share since workflows are handled as regular jobs. It is significant that regular job’s slowdown seems to stop growing after workflow shares of 25%, even decreasing for floodplain. This effect can be explained by the lower priority of the floodplain jobs, that are larger than a good share of the workload jobs.

Gain(%)	1%	5%	10%	25%	50%	75%	100%
floodP	1.80	5.22	14.46	29.29	44.53	51.64	64.47
longW	2.30	8.33	18.93	30.84	40.25	31.99	27.18
wideL	0.33	10.64	19.74	32.35	48.22	57.19	66.16
cybers	1.66	7.72	13.92	25.58	36.72	44.45	52.83
sipht	2.55	11.41	18.16	34.85	42.77	37.27	35.83
montage	12.36	44.90	60.30	72.34	80.13	82.14	85.26

Table 3: Difference of actual utilization of WoAS over the pilot job approach for different workflow shares.

In the case of the workflow aware approach, in experiments with smaller shares ($\leq 10\%$) jobs slowdown is almost the same as the case with zero workflows. For the rest of experiments (except Montage over 25%), WoAS shows only slightly bigger slowdowns (< 2x) than the chained job approach. Big increases on slowdown (over 4x) on Montage and large workflow shares (> 25%) point that, for workloads heavily dominated by workflows with large resource allocations, WoAS might have a large impact on smaller jobs. In system heavily dominated by workflows, regular jobs might not be as important. This is an effect of the backfilling limited queue processing depth (common performance optimization practice) where non-workflow jobs have to climb up the queue by waiting longer. A technique filtering non ready queue jobs before the scheduling processes are started, could alleviate this problem.

Finally, slowdown analysis was performed for all the workflows in Table 2 and for medium ([96, 480) core hours) and large ([480, ∞) core hours) jobs. Since results were similar, the data was not included due to space limitations.

5.2.3 System utilization. We compare the actual utilization between running the same experiment using workflow aware scheduling, pilot jobs, and chained jobs.

Data shows that experiments using WoAS and chained jobs present utilization over 90% and differences $\leq 5\%$. As we already analyzed in Section 5.2.1, the only exception are the experiments with the LongWide workflows and workflow shares of 75% and 100% - chained jobs show over 90% utilization, but the WoAS ones show $\approx 70\%$ and $\approx 60\%$.

Compared with the pilot job approach, WoAS has much higher levels of actual utilization as the workflow share increases.

Thus, we see that WoAS does not waste resources while producing good turnaround times (Section 5.2.1), which indicates its suitability as a workflow scheduling technique.

5.2.4 Summary. For workloads with moderate workflow presence (< %50 core hours) WoAS presents the shortest workflow turnaround times, keeps high system utilization (over 90%), wastes no resources, and regular jobs slowdown is equivalent to the best case scenario.

For workflow dominated workloads, WoAS showed the shortest turnaround times and high utilization except for the LongWide experiments. However, slowdown of regular jobs higher than the chained job cases.

As a final note, it is possible that WoAS could perform better in the dominated workflow scenarios if some queue filtering was added (not considering jobs with dependencies for queue construction) and with workloads with more workflows diversity. However that is subject of future work.

6 CONCLUSIONS

We propose Workflow Aware Scheduling (WoAS), a new model for a batch queue scheduler that enables unmodified pre-existing scheduling algorithms to take advantage of the fine grained resource requirements to produce short turnaround times without wasting resources. We implemented WoAS and integrated it in Slurm, and our implementation will be available as open source. We evaluated WoAS by simulating NERSC's Edison supercomputer and its workload, modeled after the study of three years of its real job traces.

Our results show that with WoAS, workflows show significantly shorter turnaround times than the chained job and single job approaches, and no wasted resources. In traces that have moderate workflow presence ($< 50\%$ core hours), using WoAS, FCFS and backfilling achieves turnaround times as short or shorter than submitting workflows as single jobs and much shorter than as chained jobs (up to 3.75x speedup), while keeping the system highly utilized (over 90% and *no allocated idle resources*). It also produces utilization gains over the single job approach (e.g. 60% for Montage workflows, 10% workflow share) and, has no or negligible impact on the slowdown on non workflow jobs.

Similarly, in workloads dominated by workflows ($\geq 50\%$) experiments show that workflow performance is similar to the one stated above. However, other scheduling mechanisms, like the queue depth limits in the backfilling algorithm, may affect WoAS increasing the non workflow job slowdown, specially in the presence of large workflows (e.g. 7x in Montage, 75% share). This effect could be eased by filtering not ready jobs in the waiting queue, which is future work.

We conclude that WoAS workflow scheduling performs significantly better than current approaches to executing workflows on HPC systems while posing no significant drawbacks.

7 ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR). The National Energy Research Scientific Computing Center, a DOE Office of Science User Facility, is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Financial support has been provided in part by the Swedish Government's strategic effort eSENCE and the Swedish Research Council (VR) under contract number C0590801 (Cloud Control).

REFERENCES

- [1] Gonzalo Pedro Rodrigo Alvarez, Per-Olov Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. 2016. Towards Understanding Job Heterogeneity in HPC: A NERSC Case Study. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 521–526.
- [2] Stephen Bailey. 2016. (01 2016). <https://bitbucket.org/berkeleylab/qdo>
- [3] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. 2008. Characterization of scientific workflows. In *2008 Third Workshop on Workflows in Support of Large-Scale Science*. IEEE, 1–10.
- [4] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. 2004. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing*. Springer, 11–20.
- [5] T. Fahringer, R. Prodan, R.Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wiczorek. 2007. ASKALON: A Development and Grid Computing Environment for Scientific Workflows. In *Workflows for e-Science*, I. Taylor and others (Eds.). Springer-Verlag, 450–471.
- [6] Dror G Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. 2005. Parallel job scheduling, a status report. In *Job Scheduling Strategies for Parallel Processing*. Springer, 1–16.
- [7] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, Vol. 11.
- [8] Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Riganese, Geoffroy Hautier, and others. 2015. FireWorks: a dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience* 27, 17 (2015), 5037–5059.
- [9] William TC Kramer and Clint Ryan. 2003. Performance variability of highly parallel architectures. In *International Conference on Computational Science*. Springer, 560–569.
- [10] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 239–250.
- [11] David A Lifka. 1995. The ANL/IBM SP scheduling system. In *Job Scheduling Strategies for Parallel Processing*. Springer, 295–303.
- [12] Alejandro Lucero. 2011. Simulation of batch scheduling using real production-ready software tools. *Proceedings of the 5th IBERGRID* (2011).
- [13] Muthucumar Maheswaran, Shoukat Ali, HJ Siegal, Debra Hensgen, and Richard F Freund. 1999. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*. IEEE, 30–44.
- [14] Hashim H Mohamed and Dick HJ Epema. 2005. The design and implementation of the KOALA co-allocating grid scheduler. In *European Grid Conference*. Springer, 640–650.
- [15] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde. 2007. Falcon: a Fast and Light-weight task execution framework. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM, 43.
- [16] Lavanya Ramakrishnan and Dennis Gannon. 2008. A survey of distributed workflow characteristics and resource requirements. *Indiana University* (2008), 1–23.
- [17] Lavanya Ramakrishnan, Charles Koelbel, Yang-Suk Kee, Rich Wolski, Daniel Nurm, Dennis Gannon, Graziano Obertelli, Asim YarKhan, Anirban Mandal, T Mark Huang, and others. 2009. VGRADS: enabling e-Science workflows on grids and clouds with fault tolerance. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 1–12.
- [18] Gonzalo Rodrigo, P-O Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. 2015. HPC System Lifetime Story: Workload Characterization and Evolutionary Analyses on NERSC Systems. In *The 24th International ACM Symposium on High-Performance Distributed Computing (HPDC)*.
- [19] Gonzalo Rodrigo, Lavanya Ramakrishnan, P-O Östberg, and Erik Elmroth. 2015. A2L2: an Application Aware flexible HPC scheduling model for Low Latency allocation. In *The 8th International Workshop on Virtualization Technologies in Distributed Computing (VTDC)*.
- [20] Rizos Sakellariou and Henan Zhao. 2004. A hybrid heuristic for DAG scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 111.
- [21] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 351–364.
- [22] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. IEEE, 1–10.
- [23] Massimo Benini Stephen Trofinoff. 2015. Using and Modifying the BSC Slurm Workload Simulator. In *Slurm User Group*.
- [24] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems* 13, 3 (2002), 260–274.
- [25] Marek Wiczorek, Radu Prodan, and Thomas Fahringer. 2005. Scheduling of scientific workflows in the ASKALON grid environment. *ACM SIGMOD Record* 34, 3 (2005), 56–62.
- [26] Jia Yu and Rajkumar Buyya. 2005. A taxonomy of scientific workflow systems for grid computing. *ACM Sigmod Record* 34, 3 (2005), 44–49.
- [27] Jia Yu, Rajkumar Buyya, and Chen Khong Tham. 2005. Cost-based scheduling of scientific workflow applications on utility grids. In *First International Conference on e-Science and Grid Computing (e-Science'05)*. IEEE.
- [28] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. *HotCloud* 10 (2010).
- [29] Yong Zhao, Mihael Hategan, Ben Clifford, Ian Foster, Gregor Von Laszewski, Veronika Nefedova, Ioan Raicu, Tiberiu Stef-Praun, and Michael Wilde. 2007. Swift: Fast, reliable, loosely coupled parallel computation. In *2007 IEEE Congress on Services*. IEEE, 199–206.