The Part-Time Cloud: Enabling Balanced Elasticity Between Diverse Computing Environments

Dmitry Duplyakin
Department of Computer Science
University of Colorado
dmitry.duplyakin@colorado.edu

David Johnson
School of Computing
University of Utah
johnsond@cs.utah.edu

Robert Ricci
School of Computing
University of Utah
ricci@cs.utah.edu

Abstract

Clouds, HPC clusters, HTC systems, and testbeds all serve different parts of the computing ecosystem: each are designed for different types of workloads and suited to different types of research and commercial users. We propose that an effective way to share resources among these diverse applications is to not shoehorn them all into the same resource management framework, but to partition a common hardware substrate among different frameworks: for example, to have part of a cluster managed by a cloud framework such as OpenStack, part of it managed by an HPC scheduler such as SLURM, etc. In order to efficiently manage such a shared resource, it must be possible to adjust the set of resources controlled by each in an elastic manner.

While resource allocation and scheduling *within* each of these types of environments are well studied, what we consider in this paper is elasticity *between* them. Our goal is to enable each management framework to separately manage the resources currently within its own domain, scheduling jobs, VMs, etc. according to its own needs and policies. At the same time, the frameworks can coordinate with one another so that when resources must be moved between them, it can be done in the most fair and efficient manner possible. We evaluate our ideas using a prototype that shares resources between a testbed and an HPC cluster, and with simulations using real workload traces. We find that with only minimal information flow it is possible to elastically adjust resource assignments while each framework optimizes for its own internal criteria.

Keywords

Cloud Computing, Testbeds, Elasticity, Resource Preemption

ACM Reference format:
Dmitry Duplyakin, David Johnson, and Robert Ricci. 2017. The Part-Time Cloud: Enabling Balanced Elasticity Between Diverse Computing Environments. In *Proceedings of ScienceCloud'17, Washington , DC, USA, June 26-30, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3086567.3086568

1 Introduction

Resource management frameworks for multi-tenant compute environments are generally best-suited to the types of jobs for which they were originally designed. For example, HPC schedulers such as SLURM [1] are specialized to schedule batch jobs of various sizes and durations within a tightly-coupled cluster (e.g. with a high-speed Infiniband or Ethernet network). HTC systems such as HTCondor [2] focus on less tightly-coupled jobs. Cloud management software such as OpenStack [3] focuses on virtualization and is primarily designed for web services, general server hosting, and multi-tier applications. Testbeds such as those based on the Emulab [4] framework focus on bare-metal allocation and physical isolation between tenants.

While it is possible, to some extent, to run jobs intended for one of these environments on top of another, the results are often suboptimal. For example, the virtualization overheads associated with cloud computing often add variable overhead to the networking stack and scheduling noise to the CPU, interfering with tightly-coupled computations (see e.g., [5] and [6]). Other combinations are impossible; many testbeds require users to have a level of control that is simply not possible in a virtualized cloud environment [7]. Thus, an attractive way to share a cluster between multiple types of workloads is to *share* it rather than to *stack* them; that is, to have a single hardware pool, where individual compute nodes are, at different times, controlled by a cloud management framework, an HPC scheduler, a testbed manager, etc.

In our design, the physical cluster is divided into a number of *partitions*: each physical resource is assigned to one partition at a time. Each partition is associated with a *management framework* (such as OpenStack, SLURM, HTCondor, etc.) that is responsible for scheduling jobs on the resources that are assigned to its partition. Each management framework manages its own policies (e.g., job priorities, resource limits, preemption, autoscaling, etc.). We also refer generically to the *environment* provided by a particular framework; e.g., SLURM is said to provide an "HPC environment". Figure 1 illustrates the logical relationships between environments and management frameworks on a cluster with two partitions. Since the

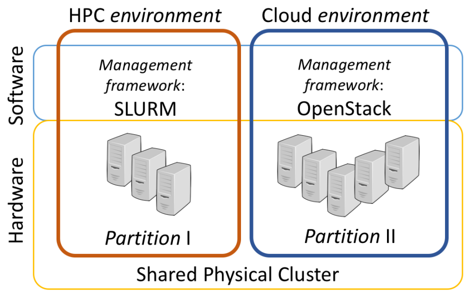


Figure 1: Multiple environments sharing the same cluster.

fundamental goal of this system is to dynamically assign resources to partitions based on time-varying demands, there must be policies regarding how to handle elasticity *between* the partitions: for example, to decide when resources should be moved from the cloud partition managed by OpenStack to the HPC partition managed by SLURM. Each management framework has its own complex methods for determining the "value" of a particular resource at a particular time. For example, SLURM may know that a particular node has been running code that is part of a long, non-preemptable job for a long time, and much work would be wasted if the node was removed from its control (thereby killing the job). On the other hand, a cloud may know that a particular server is not currently hosting any VMs, or the ones that is hosting could be easily migrated, and thus there would be very little "cost" to removing it from the cloud's partition. The values that frameworks place on the nodes under their control will change over time, as different jobs start, complete, change priority, etc. Thus, it is critical that the high-level scheduler have *enough* visibility into the relative values of each resource to make decisions that help each framework optimize its own resource scheduling, *without* having to understand the full internal details of how each computes that value. Our hypothesis is that we can establish simple, general-purpose communication between each computing framework and the overall cluster management process, and use it to avoid wasteful behavior on the cluster as a whole.

We have built a prototype of this system (described in Section 3), running on one of our clusters and available to selected users, to share resources between users of a network testbed and an HPC cluster. In Section 2, we discuss the related work in elastically scaling workloads *within* one compute environment, which contrasts with our work on elasticity of resources *between* partitions supporting diverse computing environments. In Section 4, we use a simulator with real workload traces to look at different high-level scheduling policies, and show that it is possible for an HPC framework to optimize for metrics such as minimal wasted cycles and to implement its own priority policies. We finish up with conclusions and directions for future work in Section 5.

2 Background and Related Work

There is much work in the literature, e.g., [8] and [9], that looks at dynamic scaling of individual applications. This is commonly referred to as *elasticity*. As described in [10], *horizontal* elasticity allows applications to "grow" or "shrink" the pools of their computing resources by adding or removing physical or virtual machines. This is in contrast to *vertical* elasticity, where physical resources are added to or removed from a single virtual machine. In this section, we survey some of this related work. It is complementary to the contribution of this paper, which makes clouds, compute clusters, etc. *themselves* elastic.

2.1 Targeted Computing Environments

In this work, we target modern research cyberinfrastructure environments. These take many forms, including supercomputers [11], clusters [12], grids [13], and clouds [14]. Each of these classes of systems can be characterized by unique load characteristics, resource allocation models, software stacks, administrative practices, etc. Another class of cyberinfrastructure, *testbeds*, which serve the research community by providing access to special-purpose resources and controlled environments, blurs the traditional class distinctions. They operate much like clouds, allowing users to provision groups of resources with software stacks. Unlike clouds, testbeds may provide a variety of non-uniform resource types, advanced research-driven environment pre-configuration, and controlled environments that foster reproducible research. Testbed users deploy a wide variety of software on the same testbed, resulting in diverse co-existing workloads. The NSF-funded testbeds established in the last several years include CloudLab [7], GENI [15, 16], Apt [17], Jetstream [18] and Chameleon [19].

Among the container-focused platforms, Apache Mesos [20] employs the two-level scheduling model that provides extensive support in hosting multiple computing frameworks on shared clusters. At the lower level, it offers sets of containers to individual frameworks, whereas at the higher level those frameworks internally decide how to utilize them and schedule computations. However, the lack of bare-metal provisioning makes Mesos deployments less compatible with HPC workloads considered in this paper.

2.2 Backfill

In [21], the authors describe a mechanism called *backfill* for increasing the utilization of clouds via opportunistic provisioning of cloud resources for HTC workloads. Our work incorporates the following distinctions. First, in contrast with their model with a single backfill per environment, we allow multiple elastic partitions to co-exist on the same system; a system-wide manager service communicates with them and ensures fairness. Second, while they only consider abrupt terminations of the backfilled resources, we propose a delayed, graceful preemption where the resources being preempted get a chance to complete their jobs within a specified time period. Third, comparing to the backfilled virtual machines in multiple clouds, our experiments consist of co-located physical machines that are more suitable for running HPC workloads.

2.3 Cloud Bursting

Cloud-bursting techniques for augmenting clusters with additional computing resources running in clouds are described in many studies, including [22], [23], and [24]. Much attention in these studies is paid to the analysis of how quickly the resources can be provisioned and configured at moments of peak utilization. In contrast, we consider upscaling scenarios to be thoroughly investigated and focus on downscaling, and supporting management frameworks and workloads for which bursting is non-optimal. Specifically, we investigate involuntary downscaling scenarios where the partition manager "steals" resources from partitions in response to greater demand from other partitions. This downscaling is unusual on commercial clouds because of their enormous sizes and the guarantees given to their paying customers. However, on academic clusters, testbeds, and clouds with limited capacity, steady-state behavior tends to be that all resources are near full utilization. In this case, the problem becomes moving resources between compute environments in such a way as to cause minimal disruption to the jobs managed by their associated management frameworks.

2.4 HPC Resource Management

In the large body of research on resource management strategies for HPC systems, the most relevant concepts are investigated in the following studies. In [25], the authors measure quantities that reflect the *Value of Service* (VoS) for each computing resource—an individual virtual machine in their environment. They consider such factors as completing a job within the requested wall time and staying under the specified energy consumption limit (based on additional information that user provides to the scheduler at job submission); if these conditions are violated, then programmable penalties are applied and VoS values are reduced. An importance factor that specifies the relative significance among tasks is also considered. Similarly, [26] describes penalty functions that are applied in situations where computing jobs are discarded or aborted. In our work, we do not optimize scheduling algorithms in the environments with such penalties, but rather use a common scheduling algorithm to investigate tradeoffs between scheduling-related preemption policies that influence the penalties. We will also return to the idea of penalizing jobs that run longer than expected in our discussion of future work.

3 Enabling Elastic Partitions

We now describe our system for sharing a single hardware cluster among multiple partitions, each providing a different compute environment. Our system has two major components that operate as follows. First, a **hosting cluster** is responsible for assigning individual compute resources to specific partitions. The software managing the hosting cluster must be capable of bare-metal provisioning, so that it can boot nodes into the hypervisor used by cloud hosts, the software build used by an HPC cluster, etc. Second, **elastic partitions (EP)** grow and shrink on demand, as they obtain and release resources from the hosting cluster. While we do not place any restrictions on the management framework that runs an EP, in this work we focus on frameworks where a job scheduler, compilers, and libraries are installed to allow users to run computational jobs of their choice. Users interacting with the EP see only the user interface of the framework that runs within it; for example, the SLURM job queues or the OpenStack Horizon web interface.

EP elasticity is implemented through the following three operations. First, when an EP experiences high utilization, it should acquire additional resources from the hosting cluster to increase its workload throughput; we refer to this as *partition-driven upscaling*. Second, EPs may also react to drops in utilization with *partition-driven downscaling*: being a conscious tenant of the hosting cluster implies releasing unused resources in a proactive and timely manner. The hosting cluster will reclaim these resources and make them available to other partitions. Third, *host-driven downscaling* allows the hosting cluster or its administrators to re-balance resources between partitions, by forcibly reclaiming resources from one EP and allowing another EP to request them. Specifically, we refer to the desired scenario as *graceful preemption*: the hosting cluster contacts an EP and gives it a fixed period of time, the *Grace Period* (GP), to free up P out of N EP's resources. After GP is exceed, P resources will be withdrawn from the EP without further notice.

These upscaling and downscaling operations must be defined such that they intelligently select which *particular* machines are shifted between partitions. For instance, in an upscaling scenario,

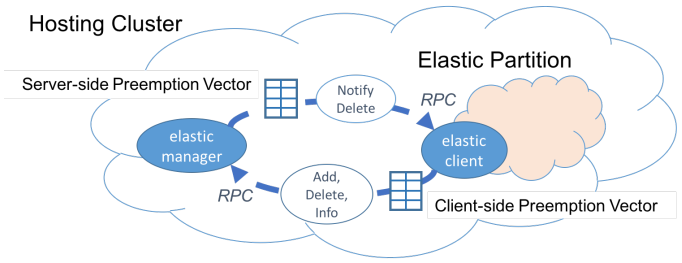


Figure 2: Experiment's elasticity facilitated by the exchange of preemption vectors.

an EP may request machines of a specific type (e.g., type of process or quantity of RAM), or with a necessary feature (GPU, high-performance interconnect, etc.). In contrast, in host-driven downscaling, the magnitude of the negative impact caused by preemptions may depend on which particular machines are preempted in the EP. Therefore, it is advantageous to allow EP to periodically report back to the hosting cluster how valuable its resources are to its applications. Given this information, the hosting cluster can preempt the least-valuable resources. In the following sections, we describe how EPs report resource values to facilitate elasticity.

3.1 Architecture

Figure 2 depicts the relationship between the hosting cluster and EPs, including the types of interactions between them. Each partition is equipped with a controller: the *elastic manager* runs in conjunction with the hosting cluster, and one *elastic client* runs in conjunction with each EP. They interact via Remote Procedure Calls (RPC) and frequently exchange resource values in the form of *Preemption Vectors* (PVs). These vectors include (*resource ID*, *value*) pairs for all resources under each partition's control. (We limit the range of these values to floats in the interval $[0, 1]$, where 1.0 is most-valued, and 0.0 is least, without sacrificing the expressive power of the interface.) PVs reported by the hosting cluster reflect the probability with which EPs can acquire specific resources; in the simplest case, 0.0 can represent available resources (those not currently assigned to any EP), and 1.0 can be used for resources in use by another EP. Responsibility for providing client-side PVs falls to the framework managing each EP, and the mechanism for calculating them depends on the goals of compute environment provided by that framework, and on framework-specific knowledge of the jobs that are currently running. For example, a testbed, which simply views resources as "allocated" or "unallocated" might use the same binary distinction described for the hosting cluster. As we will see in the sections below, frameworks that host long-running compute jobs may use values that reflect how many cycles would be "wasted" if a particular job is killed. Other frameworks may use their own notions of job priority, preemptability, etc. to decide which jobs—and therefore, nodes—are the most valuable.

Elastic clients use RPC or a GUI interface to create a partition, and later invoke the `AddNodes` and `DeleteNodes` RPCs to add or remove one or more nodes to or from their partition. They can call the `DiscoverResources` RPC at any time to list the hosting cluster's resources and metadata, such as availability. Clients send updated preemption vectors to the elastic manager at the hosting cluster via `SetResourceValues`.

Elastic clients also host an RPC endpoint through which the hosting cluster's elastic manager notifies them of its own resource values, and resource preemption. The elastic manager invokes the client's `NotifyDeletePending` method to inform the client that one or more of its nodes are being reclaimed by the hosting cluster. This method is invoked every minute for a pending node reclamation until either the client informs the manager that it has finished with the node, or until the grace period has expired for node.

3.2 Implementation

Our implementation of this architecture uses the Emulab cluster-management software [4, 27] in the role of the hosting cluster. The API calls for exchanging PVs and moving nodes between partitions are implemented as an extension to the ProtoGENI [28] API, one of the RPC interfaces supported by the Emulab software, and partitions are modeled using the "slice" abstraction that this API inherits from the GENI [15] design. We implemented an elastic manager that supports multiple elastic clients, and are running it in a restricted-use mode on the Apt cluster of the CloudLab testbed [29], where it is used to share that cluster between testbed and HPC users.

We also implemented two elastic clients: a general-purpose, reference implementation, and one that interacts with a SLURM job scheduler in order to create PV values for the nodes it manages. The SLURM client uses Chef for configuration management of its nodes, and the Chef cookbooks developed in [30]. While most management frameworks are not designed to have nodes dynamically added and removed from them, we note that most *do* have mechanisms for disabling nodes for maintenance, power savings, etc. It is the latter mechanism that our implementation hooks into in SLURM: we give SLURM a view of the entire hosting cluster, but tell it that all nodes not belonging to its partition are currently "off". In order to downscale, we tell SLURM that the effected nodes are to be turned off for maintenance, causing it to drain jobs from them. To upscale, we tell SLURM that the affected nodes have been turned back on, which makes them available for scheduling and running new jobs from the work queue.

3.3 Policies

In our elastic client for SLURM, we developed five policies that report client-side PVs as follows. The **RANDOM** policy randomly assigns preemption values. We implement this policy to evaluate the base case in which the hosting cluster has no knowledge of how much the EPs value their nodes, and therefore selects arbitrarily for downscaling. Under the **FIFO** (First-In, First-Out) policy, the longer a node has been running its current job, the *lower* the value that gets assigned, and therefore the more likely it is to be reclaimed during host-driven downscaling. Under the **LIFO** (Last-In, First-Out) policy, the longer a node has been running its current job, the *higher* the value it gets, making it less likely to be reclaimed. For workloads with multi-node jobs, the **PAP** (Parallel-Aware) policy takes into account the number of nodes used by each job; preemption values are calculated as products of job runtimes and node counts scaled to [0,1]. This makes nodes running smaller jobs more likely to be reclaimed. PAP is identical to LIFO for single-node workloads. Finally, the **PAP+** policy is an extension of PAP that weights jobs by their priority; this policy prefers to put high PV values on nodes that are currently running high-priority jobs.

We use **RANDOM** as the baseline value, since without our PV exchange protocol, the hosting cluster would have no way to know what jobs are running on the each partition's nodes and would have no basis for making efficient or fair decisions about which nodes to reclaim during host-driven downscaling. By comparing the other policies to **RANDOM**, we can quantify the effectiveness of exchanging information between the hosting cluster and EPs.

4 Evaluation

To study the effectiveness and efficiency with which the hosting cluster and EPs cooperate to share compute nodes, we developed a discrete event simulator for simulating job execution. The simulator approximates SLURM's *sched/backfill* policy by using the First-Come-First-Served job scheduling with backfilling. With subtle differences in job ordering (which we consider insignificant in this evaluation), the simulations closely resemble real-time execution experiments performed on the prototype. To draw reliable conclusions about aggregate characteristics, we simulate the behavior of EPs which stay occupied with processing jobs from realistic scientific workloads for long periods of time. Lengthy time periods are necessary to illustrate the behavior of developed policies in a variety of situations corresponding to many mixes of simultaneously running jobs with diverse runtimes and node counts. The simulator helps us understand the behavior of the policies in multiple system configurations, as well as evaluate fairly large EPs, without requiring the intractable usage of the hosting cluster's resources which would be otherwise needed for real-time experiments with the selected workloads.

Each run of the simulator takes as input a set of compute jobs and treats it as a single batch submitted at the beginning of the run. It simulates job execution until all jobs complete and records the state of execution every time one of the jobs completes. While advancing through the simulated time between these moments, the simulator also records the state every T seconds; in this evaluation, we use $T = 30$. Each of these recorded states is passed to the functions that return PVs for different policies. Using these PVs along with the information about job runtimes and used node counts, we compare the proposed policies and their variations based on the negative effects caused by preemptions, as described in detail in Section 4.1. In this paper, we describe the host-driven downscaling scenarios with $N = 20$ (EP node count) and $P = 10$ (EP nodes preempted) and briefly mention several simulation results for larger EPs.

In our analysis, we do not explicitly model sequences of preemptions followed by reductions in the EP's capacity and job throughput. Without access to real preemption event logs that are required for such modeling, we would need to make assumptions about how often and when with respect to the beginning of batch processing the host-driven preemptions are likely to occur. We choose an alternative approach and focus on the analysis of a single preemption where we accept the fact that it can happen at any time. In other words, we treat all aforementioned execution states as the moments at which the preemption has the same probability of occurrence. By capturing and processing as many such execution states as possible, we aim to characterize distributions of the policy efficiency metrics and, for instance, understand their ranges, the means, and the median values. Thus, we compare the proposed policies based on

Table 1: Parameters of the selected workloads.

Workload	# of Jobs	Total node-hours	Runtime, seconds						Node Count					
			mean	min	25%	50%	75%	max	mean	min	25%	50%	75%	max
HTC	11200	1379.05	443.27	54.00	84.00	106.00	134.25	7140.00	1.0	1.0	1.0	1.0	1.0	1.0
PEREGRINE	7275	26838.58	7119.07	33.00	439.50	1921.00	6840.5	85945.00	1.42	1.00	1.00	1.00	1.00	16.00

their behavior during tens and hundreds of thousands of simulated independent preemptions.

We use two traces of scientific computing workloads to drive the simulator. We obtain and process these traces as follows. Our **HTC** workload consists of short- and long-running single-node job traces obtained from HTCondor Log Analyzer [31]; we previously studied this trace in [32]. To prolong the simulated execution and observe more variability in the policy behavior, we create the HTC workload with over 11K jobs by concatenating 10 shuffled copies of the original trace from our study in [32]. The **PEREGRINE** dataset is a large volume of accurate job information from the HPC energy efficiency research at the National Renewable Energy Laboratory (NREL), available at [33]. We parse the dataset with 10K jobs randomly selected from two years of execution on the NREL's flagship supercomputer called Peregrine and select a subset of jobs based on the following criteria: node count $\leq$ 20 (otherwise, jobs will not run on the simulated EP), runtime $\leq$ 24 hours (longer jobs may be viewed as bad candidates for running in shrinking environments), successfully finished and have no missing fields. These criteria yield 7,275 jobs from 24 applications and 7 job queues.

Table 1 provides the detailed statistics for the jobs in these workloads. The real-time execution of PEREGRINE would require over 55 days on a 20-node HPC environment, whereas our simulator completes the processing in several hours. Our Jupyter notebooks with complete simulation, post-processing code, and all results are made available at [34].

4.1 Counting Wasted Cycles

We use *wasted cycles* as our primary efficiency metric. When a job is terminated before completion, the cycles spent so far on it are wasted, and the computation must be performed again when the job is restarted. Note that for parallel HPC jobs, if any node on which the job is running is preempted, the entire job must be restarted, so all cycles on all nodes running the job are considered wasted.

The simulator treats the $P = 10$ out of $N = 20$ nodes with the lowest values in PVs as the sets of nodes recommended for preemption by the elastic client. It counts wasted cycles (WC) for each of the jobs j from the set of jobs J_p that currently run on the preempted nodes as:

$\mathrm{WC}_j = 0$, if $j.remaining\_time < \mathrm{GP}$,

$\mathrm{WC}_j = (j.elapsed\_time + \mathrm{GP}) * j.node\_count$, otherwise.

The former rule corresponds to the cases where the grace period (GP) gives jobs sufficient time to complete. In the latter cases, all cycles spent on these jobs so far and also the future cycles within GP are wasted because the jobs are unable to complete on time. In theory, we could also count the cycles spent in the former case between the job completion and the end of GP, and treat such idle cycles as another evaluation metric. However, our prototype's elastic client terminates the nodes that are marked as draining (i.e. running their last allowed job before preemption) immediately after they become idle. Reducing the waste across the EP in this notation can be formulated as: $\mathrm{WC} = \sum_{j \in J_p} \mathrm{WC}_j \rightarrow \min$.

4.2 Varying Preemption Grace Period

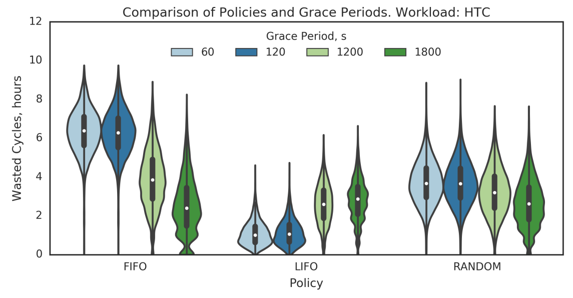
In engineering the waste-minimizing host-driven downscaling, we must carefully select the preemption grace period GP, the parameter that directly affects WC. It is intuitive to believe that the larger GPs yield lower WC values as they allow more jobs to complete. However, as we describe in the following section, this statement is not statistically accurate. We choose specific GPs for our simulations as follows: we start with GP=120s, which corresponds to the notification period for preemption of spot instances on the AWS cloud (referenced in [35]). We also consider GP=60s as an example of a hosting cluster with fast preemptions. In addition to the larger values GP=1200s (20m) and GP=1800s (30m), which can still be considered "responsive", we include in our analysis a range of less responsive values between 1 and 6 hours. This responsiveness relates to the fact that its interactive users need to wait for the requested resources to become available when all resources are utilized by EPs: the larger the GP, the longer the maximum waiting time. Finding efficient combinations of GPs that are not prohibitively large and the policies that minimize WC is one of the primary goals of our evaluation and the subject of the following section.

4.3 Quantifying Tradeoffs

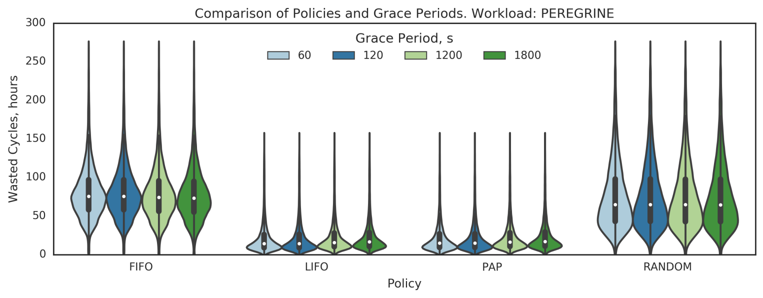
For each of the selected policy-GP combinations, our simulations yield over 14K samples for HTC and 168K samples for PEREGRINE. The distributions of the WC values that correspond to these samples are depicted in Figure 3. These violin plots characterize the probability of the observed WC values: the width of the violins represents the relative frequency of the corresponding y-axis values in the collected samples, the thick vertical lines depict interquartile ranges (IQRs), and white dots show median values (for heavily skewed data, such as the shown distributions, the median is viewed as a more representative measure of central tendency than the mean). Based on these distributions, we draw the following conclusions.

For HTC, *LIFO provides significant reductions in WC* compared to FIFO and RANDOM. LIFO's GP=60s and GP=120s statistically perform better than the larger GPs. This can be explained by the fact that the long-running jobs will often incur penalties from premature terminations (second rule for WC) shortly after they start running, and the advantage of increasing the probability of job completion does not outweigh these penalties for LIFO.

For PEREGRINE, *LIFO and PAP demonstrate similar results*, although they are significantly more efficient than FIFO and RANDOM. In fact, in more than 75% of the samples, the nodes preempted by LIFO and PAP are the same (therefore, WC is identical in these cases). This is due to the fact that the node rankings based on $j.elapsed\_time$ and the product $j.elapsed\_time * j.node\_count$ are similar for PEREGRINE where over 82% of the jobs use a single



(a) **HTC: single-node workload.**



(b) **PEREGRINE: multi-node workload.**

Figure 3: Distributions of Wasted Cycles for the developed policies and selected Grace Periods.

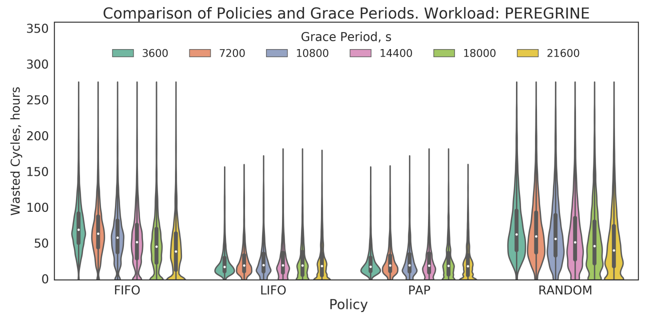


Figure 4: Distributions of Wasted Cycles for the developed policies coupled with Grace Periods between 1 and 6 hours.

node. We obtained similar results in simulations of much larger environments with $N = 200$ and $P = 100$. Since PEREGRINE is dominated by the long-running jobs, GP is a factor with negligible impact on WC, at least for GP $< 30m$.

In Figure 4, we show the impact of much larger GPs, between 1 and 6 hours, on WC for PEREGRINE simulations. The median values for WC with FIFO and RANDOM monotonically decrease with the growth of GP because an increasing number of jobs can complete execution; IQRs slide towards zero. At the same time, the WC median values for LIFO and PAP remain constant, and IQRs do not change much. It appears that the growth of the frequencies of the lower, near zero WC values is counterbalanced by the increased frequencies of the larger values. In the extreme case where GP is so large that it exceeds the majority of job runtimes, WC will approach zero; however, such GPs are unresponsive and impractical from the testbed perspective. We conclude that the large but practical GP values up to 6 hours do not provide noticeable advantages in terms of the WC's median and IQR for workloads similar to PEREGRINE.

4.4 Improving Parallel-Aware Policy

We can enable practical optimizations in PAP if we consider relative significance among jobs. For instance, two jobs that use the same number of nodes and start roughly around the same time may have drastically different values that characterize their importance: small for cycle-scavenging, low-priority computations and much larger for applications with strict deadlines. Similarly, the relative importance can be expressed using job queues. In PEREGRINE, the dataset suggests that the NREL's system has 7 job queues. We can view *bigmem* or *large* queue as more important than *debug* and help jobs in these queues to have fewer terminations.

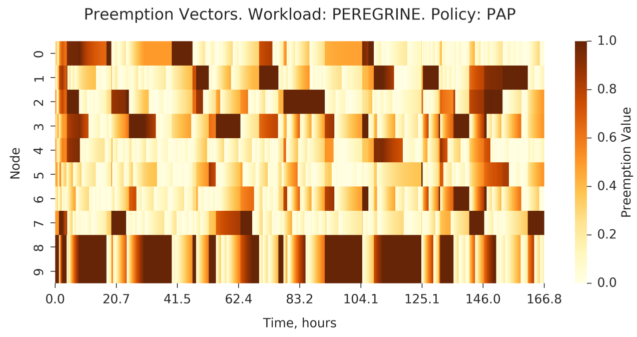
We developed the PAP+ policy that considers two priority classes, but the proposed PV exchange with the [0,1] values can support an arbitrary number of classes. To evaluate the potential tradeoffs, we choose 76 jobs in the *bigmem* queue in PEREGRINE (∼1% of the total number) to be the highly valuable subset with $j.priority = 10.0$; for the rest of the jobs, we assume the default class: $j.priority = 1.0$. The elastic client calculates PVs as:

$$\text{PV} = \text{scale}(j.elapsed\_time * j.node\_count * j.priority),$$

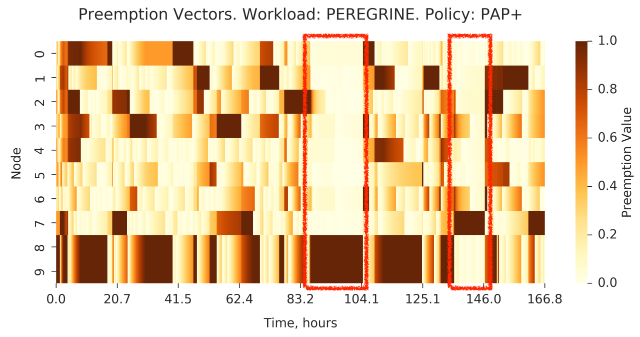
where scale() divides vector components by their maximum value (in contrast, PAP treats the last factor as 1.0 for all jobs).

Figure 5 shows the difference between PAP and PAP+ using heatmaps for the PVs they return during a simulation of a fraction of the PEREGRINE workload. We can clearly see several intervals of time (marked with red rectangles) when the high-priority jobs run. These jobs gain importance after they start running much faster than other jobs and also reduce the relative importance of the rest of the simultaneously running jobs (i.e. the rest of the red rectangles appear mostly white, indicating little relative value).

With the same procedure for estimating WC, we compare PAP and PAP+ based on the cumulative amount of WC. In Figure 6, we can see how the cumulative WC values change throughout the simulated executions and compare these values at the end of PEREGRINE simulations. PAP+ reduces cumulative WC for high-priority jobs by 82.9% comparing to PAP. This happens at the expense of the default-priority jobs: PAP+ increases cumulative WC for default-priority jobs by 14.3% comparing to PAP. The vertical axes on the shown plots use different scales, indicating a large contrast between the corresponding absolute differences. However, the 7.5% growth of the combined cumulative WC is well justified by the significant reduction of the negative impact on the high-priority jobs. In the similar manner, PAP+ can focus on preserving jobs that represent a particular application. In our experiments where we assign $j.priority = 10.0$ to jobs with $j.application$ = gaussian (341 jobs, ∼ 5% of total), we observe the 97.2% reduction in the cumulative WC for high-priority jobs provided by the 7.1% increase for default-priority jobs (only 3.8% growth in total).



(a) PAP policy.



(b) PAP+ policy. Red rectangles highlight the moments when the running high-priority jobs reduce the relative importance of the running default jobs.

Figure 5: Heatmaps that visualize Preemption Vectors for a 10-node elastic partition running the PEREGRINE workload. The darkest areas represent the nodes that are most valuable. Only a small fraction, under 15%, of the entire simulation is shown.

4.5 Discussion

Below, we discuss the limitations of our analysis and the nuances which we must consider when moving our prototype into production. Using a more advanced scheduler simulator such as the SLURM simulator described in [36] can enable experimentation with schedules that better represent production HPC systems with fairness and quality-of-service optimizations. However, the referenced system takes as input continuous SLURM event logs, and running it on the sampled, anonymized, and formatted records published at [33] is infeasible. To experiment with the SLURM simulator, we will need to find alternative sources of HPC traces. Alternatively, we can switch from simulations back to real-time execution experiments on the developed prototype. It will never be practical to run 55-day experiments on our shared testbed resources, but we can gain valuable experience with the optimized scheduling algorithms in SLURM using modestly long experiments.

Do large Grace Periods provide any advantage? In our evaluation, they "freeze" preempted nodes for long periods of time without reducing the amount of wasted computations for the most efficient policies. If this conclusion holds for other workloads – candidates for running on elastic partitions, we may consider reducing GP to its minimum value, on the order of several seconds. Thus, SLURM will receive the time that is only enough to kill the currently running jobs and mark the preempted nodes as unavailable. Such preemptions will still proceed gracefully, and, from the testbed perspective, will eliminate the need to maintain a pool of idle resources unavailable to elastic partitions. On the contrary, with larger GPs, if their use is justified, the testbed needs to have such pool to quickly respond to requests from its interactive users. This idle pool is likely to reduce the overall utilization of the testbed.

Another practical concern relates to stale PVs, i.e. PVs that represent states of execution at moments in the past that are far from the current time. Making preemption decisions based on such PVs is likely to cause non-optimal terminations. Stale PVs may adequately represent long-running jobs, but for the jobs that have recently started running they will include faulty, unreliable values. The longer the delay between the moments when PVs are reported and the moments when preemption actions are triggered, the faster LIFO, PAP, and PAP+ degrade to the RANDOM policy.

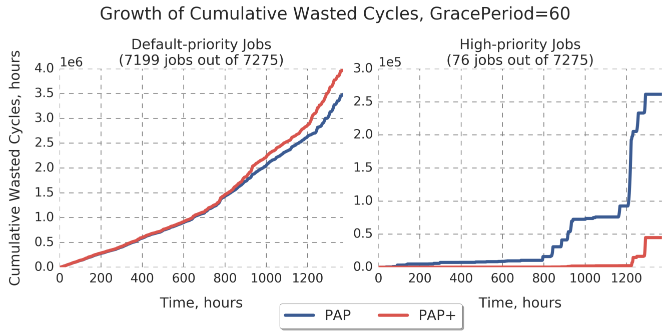


Figure 6: Using cumulative Wasted Cycles to compare PAP and PAP+ that prioritizes jobs from the specified queue. At the expense of a small growth for default-priority jobs, PAP+ significantly reduces Wasted Cycles for high-priority jobs.

In this degradation, the gains provided by these policies are diminished. Therefore, the hosting cluster needs to ensure that it indeed uses the recently received node value information; otherwise, preemptions should be delayed or aborted.

5 Conclusions and Future Work

In this work, we have investigated interactions between portions of a shared cyberinfrastructure managed by diverse computing frameworks, and presented a novel infrastructure for resource sharing and optimization of resource usage. In our analysis, we use an elastic prototype—a SLURM-managed elastic cluster—deployed on the CloudLab testbed and also a discrete event simulator we developed for simulating workload traces from real HPC systems. By simulating a trace from the Peregrine supercomputer at NREL, we obtained insights into how these elastic compute clusters operate under realistic HPC workloads with long-tailed distributions for runtimes and numbers of compute nodes. In balancing their batch and interactive workloads, we rely on a minimal information flow based on the exchange of preemption vectors that abstractly represent environment-specific resource values. Using wasted cycles as a metric, we demonstrated that, with this minimal information sharing, we are able to give each environment the ability to optimize for its goals. We also found that, somewhat counter-intuitively, shorter downscaling grace periods lead to fewer wasted cycles, as most jobs do not finish within the grace period anyway.

Experimenting with traces from other HPC centers and characterizing the behavior of the developed elastic controllers under corresponding workloads constitute one of the directions in our future research. We will also consider developing policies that take into account user-requested job wall clock times. However, we are skeptical of the accuracy of such estimates; e.g., for PEREGRINE, they demonstrate extremely low accuracy: for over 50% of the jobs, the requested wall clock exceeds the actual wall clock by the factor of 19.2. While the accurate prediction of job runtimes is considered a difficult problem and, to the best of our knowledge, there is no common practical solution, we imagine that assigning heavier weights to the most recent jobs for each user, application, queue, etc., and predicting whether the remaining runtimes are likely to exceed the Grace Period or not is feasible. Another direction to further optimize PAP+ is to introduce a job prioritization that changes after jobs exceed their requested wall clock times. Similarly to the scheduler configurations used at many HPC centers that terminate such jobs, PAP+ can deprioritize such jobs, and therefore increase the probability of preemption for the nodes on which they run. Finally, we will continue to experiment with the prototype and the developed simulator in order to obtain additional insights and gain confidence in the fact that the elasticity mechanisms are reliable and sufficiently intelligent before releasing them to the community of users on CloudLab and related testbeds.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1338155. We would like to thank Caleb Phillips from the National Renewable Energy Laboratory for making the Peregrine workload traces publicly available. We would also like to thank the CloudLab and Emulab staff at the University of Utah for their feedback on the design and implementation of the elasticity controllers. Finally, we thank Joe Breen and Brian Haymore from the Center for High Performance Computing at the University of Utah for integrating an elastic partition on CloudLab with the production HPC system.

References

[1] SchedMD. Slurm workload manager, 2017. URL https://slurm.schedmd.com/.

[2] HTCondor. Computing with HTCondor, 2017. URL http://research.cs.wisc.edu/htcondor/.

[3] OpenStack. Open source software for creating private and public clouds., 2017. URL https://www.openstack.org/.

[4] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, December 2002.

[5] Jinho Hwang, Sai Zeng, Frederick y Wu, and Timothy Wood. A component-based performance comparison of four hypervisors. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 269–276, 2013.

[6] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, page 1. ACM, 2016.

[7] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[8] Trieu C Chieu, Ajay Mohindra, Alexei A Karve, and Alla Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *IEEE International Conference on E-Buisness Engineering*, pages 281–286, 2009.

[9] Yi Wei and M Brian Blake. Proactive virtualized resource management for service workflows in the cloud. *Computing*, 98(5):523–538, 2016.

[10] Guilherme Galante, Luis Carlos Erpen De Bona, Antonio Roberto Mury, Bruno Schulze, and Rodrigo Rosa Righi. An analysis of public clouds elasticity in the execution of scientific applications: A survey. *J. Grid Comput.*, 14(2):193–216, June 2016. ISSN 1570-7873.

[11] Dror G Feitelson. The supercomputer industry in light of the top500 data. *Computing in science & engineering*, 7(1):42–47, 2005.

[12] Rajkumar Buyya. High performance cluster computing: Architecture and systems, volume i. *Prentice Hall, Upper SaddleRiver, NJ, USA*, 1:999, 1999.

[13] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[14] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[15] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. GENI: A federated testbed for innovative network experiments. *Computer Networks*, 61(0):5–23, 2014.

[16] Rick McGeer, Mark Berman, Chip Elliott, and Robert Ricci, editors. *The GENI Book*. Springer International Publishing, 2016.

[17] Robert Ricci, Gary Wong, Leigh Stoller, Kirk Webb, Jonathon Duerig, Keith Downie, and Mike Hibler. Apt: A platform for repeatable research in computer science. *ACM SIGOPS Operating Systems Review*, 49(1), January 2015.

[18] Craig A Stewart, Timothy M Cockerill, Ian Foster, David Hancock, Nirav Merchant, Edwin Skidmore, Daniel Stanzione, James Taylor, Steven Tuecke, George Turner, et al. Jetstream: a self-provisioned, scalable science and engineering cloud environment. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, page 29. ACM, 2015.

[19] Chameleon Team. Chameleon cloud: A configurable experimental environment for large-scale cloud research, 2017. URL https://www.chameleoncloud.org/.

[20] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[21] Paul Marshall, Kate Keahey, and Tim Freeman. Improving utilization of infrastructure clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 205–214, 2011.

[22] Maciej Malawski, Gideon Juve, Ewa Deelman, and Jarek Nabrzyski. Algorithms for cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. *Future Generation Computer Systems*, 48:1 – 18, 2015. ISSN 0167-739X. Special Section: Business and Industry Specific Cloud.

[23] M. Mattess, C. Vecchiola, and R. Buyya. Managing peak loads by leasing cloud infrastructure services from a spot market. In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pages 180–188, Sept 2010.

[24] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic site: Using clouds to elastically extend site resources. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 43–52, Washington, DC, USA, 2010. IEEE Computer Society.

[25] Dylan Machovec, Cihan Tunc, Nirmal Kumbhare, Bhavesh Khemka, Ali Akoglu, Salim Hariri, and Howard Jay Siegel. Value-based resource management in high-performance computing systems. In *Proceedings of the ACM 7th Workshop on Scientific Cloud Computing*, pages 19–26, 2016.

[26] Shuo Liu, Gang Quan, and Shangping Ren. On-line scheduling of real-time services for cloud computing. In *Services (SERVICES-1), 2010 6th World Congress on*, pages 459–464. IEEE, 2010.

[27] Emulab. Network emulation testbed home, 2017. URL https://www.emulab.net/.

[28] Jay Lepreau et al. Protogeni. In *1st GENI Engineering Conference*, 2007.

[29] CloudLab. Flexible, scientific infrastructure for research on the future of cloud computing., 2017. URL https://cloudlab.us/.

[30] Dmitry Duplyakin and Robert Ricci. Introducing configuration management capabilities into cloudlab experiments. In *Computer Communications Workshops (INFOCOM WKSHPS), 2016 IEEE Conference on*, pages 39–44, 2016.

[31] Condor Log Analyzer, 2016. URL http://condorlog.cse.nd.edu/.

[32] Dmitry Duplyakin, Paul Marshall, Kate Keahey, Henry Tufo, and Ali Alzabarah. Rebalancing in a multi-cloud environment. In *Proceedings of the 4th ACM workshop on Scientific cloud computing*, pages 21–28, 2013.

[33] HPC Energy Research, 2017. URL https://cscdata.nrel.gov/#/datasets/d332818f-ef57-4189-ba1d-beea291886eb.

[34] Dmitry Duplyakin. Jupyter notebooks for experimental evaluation, 2017. URL http://dmitry.duplyakin.org/p/elastic/.

[35] Hao Wu, Shangping Ren, Steven Timm, Gabriele Garzoglio, and Seo-Young Noh. Experimental study of bidding strategies for scientific workflows using aws spot instances. In *Proceedings of 8th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS)*. ACM, 2015.

[36] Alejandro Lucero. Simulation of batch scheduling using real production-ready software tools. *Proceedings of the 5th IBERGRID*, 2011.