Dynamically Negotiating Capacity Between On-demand and Batch Clusters

Feng Liu
University of Minnesota
liux2102@umn.edu

Kate Keahey
Argonne National Laboratory
keahey@mcs.anl.gov

Pierre Riteau
University of Chicago
priteau@uchicago.edu

Jon Weissman
University of Minnesota
jon@cs.umn.edu

***Abstract*—In the era of rapid experimental expansion data analysis needs are rapidly outpacing the capabilities of small institutional clusters and looking to integrate HPC resources into their workflow. We propose one way of reconciling on-demand needs of experimental analytics with the batch managed HPC resources within a system that dynamically moves nodes between an on-demand cluster configured with cloud technology (OpenStack) and a traditional HPC cluster managed by a batch scheduler (Torque). We evaluate this system experimentally both in the context of real-life traces representing two years of a specific institutional need, and via experiments in the context of synthetic traces that capture generalized characteristics of potential batch and on-demand workloads. Our results for the real-life scenario show that our approach could reduce the current investment in on-demand infrastructure by 82% while at the same time improving the mean batch wait time almost by an order of magnitude (8x).**

***Index Terms*—Computers and information processing, Distributed computing, Metacomputing, Grid computing.**

I. Introduction

The recent improvements in experimental devices, ranging from light sources to sensor-based deployments, lead not only to the generation of ever larger data volumes but to the need to support time-sensitive execution that can be used effectively in the management of experiments, observations, or other activities requiring quick response turnaround. This means that small, dedicated analysis clusters used by many experimental communities are now no longer sufficient and their users are increasingly looking to expanding their capacity by integrating high performance computing (HPC) resources into their workflow. This presents a challenge: how can we provide on-demand execution within HPC clusters which are today operated mostly as batch?

The inspiration for our project was provided by scientists from the Advanced Photon Source (APS) at the Argonne National Laboratory (ANL). APS is currently operating a cluster dedicated to experiment support: the execution of jobs run on the cluster has to be completed in the shortest time possible; thus the need for dedicated resources. However, as the experiments increasingly require greater processing power, an interest arose in using HPC resources so long as they can be provisioned on demand in a cost-effective manner and with environments suitable to APS computations. This conflicts with the modus operandi of HPC resources today, which are usually available via batch schedulers maximizing utilization and thus amortization of expensive resources, and do not provide environment management. In this paper, we propose a solution to this use case.

Our paper presents the design and evaluation of the Balancer: a service that dynamically moves nodes between an on-demand cluster configured with cloud technology (in our case OpenStack) and an on-availability cluster configured with a batch scheduler (in our case Torque) as the need for on-demand availability changes. The ability to integrate commodity, generally used technologies was an important requirement of our design. Another requirement was to make it as non-invasive as possible, i.e., to not kill or checkpoint running batch jobs as we have done in [1], or rely on specialized adjustments to scheduling policies of the existing tools. We propose three different algorithms for moving nodes between the on-demand and batch partitions and evaluate them, first experimentally in the context of real-life traces representing two years of a specific institutional need, and then via experiments in the context of synthetic traces that capture generalized characteristics of potential batch and on-demand traces.

Our results, based on a real-life scenario, show first that combining capacities and workloads of on-demand and batch clusters can provide sufficient capacity to satisfy all on-demand requests while reducing the dedicated portion of the cluster by 82%, improving the mean batch wait time almost by an order of magnitude (8x), and improving the overall utilization as well. We secondly show that in a general case we can support bursty on-demand workloads corresponding to up to 10% capacity of the cluster it shares with batch workload in a non-invasive way.

In summary, our paper makes the following contributions:

- We describe an architecture and implementation for dynamic non-invasive resource reassignment between two systems: a system providing resources on-demand and a system providing resources based on availability, that balances their respective objectives in terms of the number of satisfied on-demand requests and utilization.
- We propose three algorithms for balancing resources in this context: the basic Algorithm providing a baseline of our systems, the hint Algorithm that models the behavior where experimental users can register the upcoming need for on-demand cycles, and the predictive algorithm for cases where such advance notice is not possible.

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

- We evaluate these algorithms for different Balancer behaviors, to understand what workloads we can successfully balance under this system, using two years of traces from an experimental and a mid-scale cluster at Argonne. We show that we can not only support the existing use case with dedicated resources significantly reduced, but also scale the bursty on-demand workload to up to 10% of the capacity of the cluster it shares with the batch workload in a non-invasive way.

II. Approach

The inspiration for our project was provided by scientists from the Advanced Photon Source (APS) at the Argonne National Laboratory (ANL). APS provides a facility for experiments in many scientific domains. To support them, it operates a on-demand cluster dedicated to experimental analytics; the execution of jobs on this cluster is typically time-critical, where completion of the computation is needed within a time determined by the type of measurement. At the APS, nodes are reserved for specific experiments and are allowed to sit idle between data collection events, resulting in very low utilization but high responsiveness. While use of a larger-scale HPC cluster with many more nodes might provide a result much faster, which would be greatly desirable, the chance that a queue backlog could produce an unpredictable delay cannot be tolerated.

Infrastructure-as-a-Service cloud technologies, such as OpenStack [2], have been a popular solution for on-demand access as they also provide environment management via the deployment of virtual machines (VMs) or containers. We propose to use those existing cloud technologies and provide a system that will combine them with HPC schedulers in a non-invasive way, by arbitrating resource assignment between them. Specifically, the system will meet the following objectives:

- Inject on-demand and environment management for the on-demand resources into batch clusters such that we can schedule as many on-demand leases as possible with as little impact on utilization as possible (i.e., maximizing utilization while minimizing the number of rejected on-demand requests).
- Provide a solution in terms of existing commodity frameworks for both on-demand and batch, such as OpenStack or Torque, such that the user's interface to those systems does not change and the changes to the systems themselves are minimal though flexible.
- The solution should be minimally invasive in terms of interference with the normal operation of the batch scheduler. We will not e.g., kill or checkpoint/snapshot jobs in order to make room for on-demand leases [1] or rely on the availability of specialized queues with smaller sized jobs that can be used for backfilling [3].

A. Leases

To explain our approach, we will use the concept of a *lease*, defined as a temporary ownership of resources, taking place between a well-defined start time and end time. In this paper, we will differentiate leases based on their start time; the end time may be bounded (e.g., assumed or specified to last a specific amount of time) or unbounded (used until terminated by an event).

We define two types of leases, one reflecting the concern of users who are interested in controlling the start time of their computations, and the other reflecting the concern of the providers, who are interested in optimizing the utilization of their resources:

- An *on-demand lease* starts within a window of time W after the request has been made and may or may not have a defined end time. Time W is typically understood to be short, e.g., under a minute or two, and may comprise actions such as virtual machine deployment and boot. This startup time can be arbitrary and can include some system management, e.g., terminating jobs in order to make room for a lease. On-demand leases are the most common type of request in compute clouds and are implemented by all major cloud providers.
- An *on-availability lease* starts whenever the provider makes resources available for the lease. Examples of on-availability leases include resource assignments given out by a batch scheduler, high throughput leases implemented by systems such as SETI@home [4], or spot pricing leases implemented by Amazon EC2 [5]. Since the lease may not (and generally does not) start immediately, the request is typically placed on a queue; the provider selects it for resource allocation based on a variety of concerns that generally favor increasing utilization but may also take other factors into account (e.g., EC2 spot pricing).

B. Architecture

Our approach is to soft-partition nodes in a large cluster into two scheduling pools, an on-demand pool and an on-availability pool, and to implement a mechanism that will dynamically move nodes from one pool to the other to maximize our objectives.

In keeping with our assumptions, both resource managers (on-demand and on-availability) are independent of each other; nodes in the on-demand pool are managed by an on-demand resource manager (ODRM) while nodes in the batch pool are managed by an on-availability resource manager (OARM). The clients of each resource manager – such as a job queuing or scheduling systems – use their respective interfaces to request resources; they are not affected by the presence of the other resource manager, except as by having some requests rejected or delayed due to changes in resource availability. For example, an ODRM could be implemented by a job queuing system submitting resource requests to OpenStack; an OARM could be implemented as a combination of a Maui [6] job scheduler working with a Torque [7] resource manger.

Our architecture (Figure 1) consists of a service, called the Balancer, which negotiates adjustments in the respective sizes of the on-demand and on-availability resource pools with ODRM and OARM. Implementing the Balancer as a

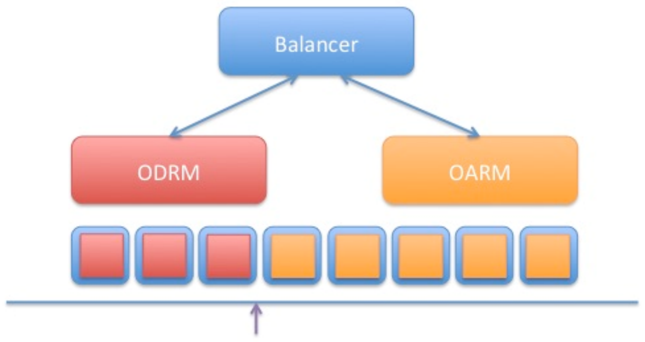

Fig. 1: High-Level Architecture

separate service, distinct from both resource managers, allows us to implement bilateral negotiation and also to implement the system with minimal changes to both resource managers. The Balancer understands the status of each node in the whole resource, as well as whether at any given moment they belong to the ODRM or OARM pool. However, the Balancer only manages which nodes belong to which pool; the scheduling decisions are left to ODRM and OARM. The boundaries between the pools are re-evaluated by the Balancer on an ongoing basis, negotiating with each scheduler for the availability of nodes in the respective clusters.

The on-demand pool contains a group of nodes called the reserve R, which may be set to zero. The reserve represents nodes that cannot be moved to the batch pool and is intended to ensure that the system has up-front capacity to schedule resources when on-demand requests come in. Otherwise, the division between the on-demand and on-availability parts of the cluster is fluid and constantly re-evaluated. Another parameter of the system is the time window W, which defines how long the system can wait before scheduling an on-demand request.

In the context of this paper, the Balancer implements a simple one-way negotiation which requests nodes from the OARM as needed; nodes from the ODRM are only contributed by the resource manager itself. Under the current assumptions, execution on the nodes in the on-availability pool has to be finished before they are contributed to the Balancer; this means that the Balancer's request for nodes from OARM to be contributed in the allotted time may be unsuccessful. Ultimately, this negotiation protocol can be extended to implement more complex constraints.

The interaction with OARM and ODRM takes place via the following interfaces.

Balancer Interface:

- $request\_nodes(int\ n)$: request n additional nodes from the Balancer (the decision of which specific nodes from the on-availability pool should be made available to the ODRM is made by the Balancer)
- $release\_nodes(node\_list)$: release specific nodes to the Balancer
- $update\_nodes(node\_states)$: attempts to update status of specific nodes. Can return an error if the status update is incorrect. This interface is used by OARM when: (1) it attempts to run a job on a node it believes is in the OARM pool and calls this interface to avoid a race condition with Balancer reclaiming this node for ODRM at the same time; and (2) it finishes executing a job, giving the opportunity for the Balancer to reclaim it if needed.

OARM Interface:

- $reclaim\_node(nodename)$: reclaim a specific node, identified by its hostname, from Balancer to OARM
- $restore\_node(nodename)$: restore a specific node, identified by its hostname, from Balancer to OARM
- $get\_status\_of\_all\_nodes()$: returns a list of data structures that for each node describes if it is busy executing a job, free, or offline, and if it is executing a job what is the remaining wall time.

C. Algorithms

The Balancer algorithms make time-varying estimations of the amount of resources to provision to on-demand requests. The primary goal of the algorithms is to reduce the number of on-demand rejections. When the primary goal is met, the secondary goal is to reduce the aggregated amount of resources that are provisioned to the on-demand workloads, such that more resources will be remained for batch workloads.

1) Basic algorithm: The objective of the basic algorithm is to implement a simple mechanism whereby the Balancer requests nodes from the OARM as on-demand requests come in and uses reserve as well as wait time to "pad" availability. Algorithm 1 shows the pseudo-code. At any point of time, a node can be in any one of 4 states: $OD\_Reserve$, OD_Alloc, OA_Idle, and OA_Busy.

R is the number of nodes that are statically reserved for the ODRM pool. When an on-demand request comes in, the Balancer allocates nodes from: (1) $OD\_Reserve$, (2) OA_Idle, and (3) OA_Busy nodes whose jobs finish before time W. A request is rejected if the Balancer cannot allocate n nodes before time W or immediately when $W = 0$.

2) Hint algorithm: The hint algorithm is a refinement of our original attempt at the basic algorithm and reflects the fact that in experimental communities it is often possible to determine resource need within a short time (15-30 minutes), though it may not always be possible to pinpoint it to a particular time days in advance. This allows us to implement a dynamic reserve (i.e., a reserve that changes according to the situation).

Both functions $request\_nodes$ and $release\_nodes$ stay the same as in the basic algorithm. The Balancer introduces another interface $add\_reserve(H, N)$, mandating the Balancer to add N extra reserve nodes before time H. Here we are essentially parameterizing the hint algorithm by two parameters: time H for "advance notice" or "hint", and N the number of nodes that are requested by the user or a third party. Note that no nodes are statically reserved in the hint algorithm—any nodes that are in $OD\_Reserve$ state for more than I seconds will be released to OARM pool.

```
Input: $R$ (default = 0), $W$ (default = 0)
Function request_nodes($n$):
    $nr \leftarrow$ nodes currently in $OD\_Reserve$ state
    $ni \leftarrow$ number of nodes in $OA\_Idle$ state
    if $nr \geq n$ then
        allocate $n$ $OD\_Reserve$ nodes
        change node state to $OD\_Alloc$
        return $node\_list$
    else
        if $nr + ni \geq n$ then
            $reclaim\_nodes(n - nr)$
            change node state to $OD\_Alloc$
            return $node\_list$
        else
            if $W = 0$ then
                return $Rejection$
            else
                $reclaim\_nodes(ni)$
                wait for $W$ seconds
                foreach received update_nodes message do
                    $reclaim\_nodes(1)$
                    if $n$ nodes can be allocated then
                        change node state to $OD\_Alloc$
                        return $node\_list$
                    end
                end
                if $W$ expires then
                    return $Rejection$
                    reclaimed nodes are kept in $OD\_Reserve$
                    state for $I$ seconds before release to
                    OARM pool
                end
            end
        end
    end

Function release_nodes($node\_list$):
    foreach node in node_list do
        change node state $OD\_Alloc \rightarrow OD\_Reserve$
        if node is not statically reserved then
            node is kept in $OD\_Reserve$ state for $I$ seconds
            before release to OARM pool
        end
    end
```
Algorithm 1: Basic algorithm

3) Predictive algorithm: The predictive algorithm is another refinement of the basic algorithm that also implements the notion of a dynamic reserve but in situations when an advance notice is not possible. The predictive algorithm is run by an out of band predictor which invokes the $add\_reserve$ interface on behalf of the users. The predictor collects historical data from the Balancer and use history to predict future on-demand requests.

Our predictive algorithm is based on three observations of the arrival time of on-demand requests. Firstly, the arrival time follows a strong diurnal pattern which can be explained by the interactive nature of the APS workload. Secondly, the arrival time shows moderate correlation between adjacent weeks, meaning that if there is a burst of requests during several hours of one week, then there is likely a similar burst during the same hours of the next week. Thirdly, sometimes there are bursts of requests during the same hours of consecutive days. Our predictive algorithm is described as follows:

The predictor divides each day into four 6-hour slots. At the end of each time slot, the predictor queries the Balancer for how many nodes were requested during the time slot. At the beginning of each time slot, the predictor invokes $add\_reserve(0, N)$ to reserve N nodes where N is estimated based on the peak number of requested nodes during the same time slot of the last month, week, and day.

D. Implementation

Our implementation of the Balancer is configured to work with the Torque resource manager and the Maui cluster scheduler, used as OARM, and OpenStack (with the KVM hypervisor [8]) as ODRM. It consists of a simple web service developed using the Flask Python micro web framework, separately from either OpenStack or Torque. It offers an HTTP endpoint capable of receiving resource requests as well as notifications of resource status changes. To move nodes between the on-demand and on-availability pools, the Balancer enables or disables them in Torque using the `pbsnodes` command with arguments `-o` to disable or `-c` to enable.

The $update\_nodes$ interface is implemented for Torque nodes as prologue and epilogue scripts which are triggered respectively when a job starts and ends execution (whether successfully or not). These notifiers make HTTP requests to the balancer in order to update its record of resource status, i.e., nodes available for stealing. No other changes were required to integrate Torque into the system.

In order to make OpenStack work with the Balancer, we had to make small modifications to the OpenStack implementation: the scheduler (Nova) requests more resources from the Balancer if it does not have enough available for scheduling virtual machines requested by on-demand users (using $request\_nodes$), and resources are released to the Balancer when instances are terminated (using $release\_nodes$). We also had to fix concurrency issues in the scheduler when using large wait times which can make many independent resource requests block and then resume execution at the same time.

III. Experimental Evaluation

We conduct our experimental evaluation in two stages. We first evaluate our approach using the basic algorithm in the context of a real-life scenario defined by two years worth of traces reflecting the needs of on-demand and batch jobs at the Argonne National Laboratory; this gives us insight into realistic demand and submission patterns. Second, we use synthetic traces to generalize the problem and evaluate and compare the three algorithms we formulated.

Our overall experimental methodology consisted of emulating the actual runs by submitting traces of on-demand and batch requests to OpenStack and Torque configurations respectively, on a cluster managed by the Balancer. The OpenStack submissions use a mechanism called FakeDriver which, instead of launching a real VM, generates the suitable internal events that track resource consumption. The Torque submissions use a ”sleep” script for the duration of the job walltime. In addition, CPU overcommitment is disabled in OpenStack.

A. Evaluating a Real-Life Scenario

To evaluate our approach we first ask the question: how would it fare under existing shared on-demand/on-availability workloads in real-life computational centers? To answer this question, we combined both workloads and resources of two systems used at the Argonne National Laboratory (ANL). The on-demand side is represented by workloads ran on a small Sun Grid Engine cluster in the Advanced Photon Source (APS) used for analytics supporting real-time experiments; hence the need for immediate execution. The batch side is represented by a general purpose mid-scale batch cluster in Laboratory Computing Resource Center (LCRC). Given this context, a more specific version of our question is: if we combined both the on-demand/on-availability workloads and the resources currently executing those workloads under our approach, what advantages or disadvantages would we observe?

To create a combined APS/LCRC workload we combined two years worth of job execution traces from APS and LCRC (between 2013-10-06 and 2015-09-05). We first mapped the job execution trace from APS onto on-demand VM deployment requests in OpenStack as follows. All APS jobs are, or can be treated as, single-core jobs [1]. The APS trace records the start/stop timestamps of all the jobs. At any APS job start/stop event, we evaluated how many single-core jobs should be running in the APS cluster and how many 16-core VMs would be needed to support them, assuming that jobs would be tightly packed. If more or fewer VMs would be needed as a result of a job start/stop event, an on-demand VM deployment request or termination event would be generated. We then combined the APS on-demand trace and the LCRC batch trace using the same start time for both, such that the VM deployment requests are submitted to OpenStack and batch job requests are submitted to Torque.

To create a combined APS/LCRC cluster we proceeded as follows. The LCRC cluster comprises 304 homogeneous 16-core nodes. The APS cluster consists of 57 heterogeneous multi-core nodes amounting to a total of 1092 cores. Since cores represent the main scheduling concern in our experiment, we modeled APS capacity as 68 16-core nodes (a total of 1088 cores, close to the actual 1092 capacity). The combined APS/LCRC cluster is thus modeled as 304 + 68 = 372 16-core nodes.

We now set out to replay the combined APS/LCRC workload on a model of the APS/LCRC combined cluster. Since we could not replay two years worth of traces in real-time, we scaled down the experiment in space and time. To scale it in space, we created an experimental environment that mapped each of the 372 combined cluster nodes onto a Docker container, each with a unique hostname and IP address, connected by an overlay network. An additional container represented the controller node. We deployed the Docker containers on the Chameleon testbed [9] version 53, using the 24-core 128 GiB RAM Xeon Haswell compute nodes, such that 24 containers were mapped to each node. To scale the experiment in time, we mapped hours to minutes (i.e., accelerated 60x). Finally, we eliminated the ramp-up effect by preloading the cluster with running jobs.

[1]Some APS jobs contain an array of subjobs. But each subjob runs on a single core.

This still left us with a potentially very long experiment, so instead of replaying two years worth of traces we focused on one week that would represent the greatest challenge to our system. In the case of the batch trace, we defined ”challenging” as low average node availability across 60 second periods, measured every second. In the case of the on-demand trace, we defined ”challenging” as high total resource usage coming from on-demand requests, calculated as a sum over the product of the time used by a job and number of cores on which the job was running. We picked the week which had the highest sum of usage and inverse of availability.

We now ran the experiments using traces from the most challenging week reflecting the modifications above, such that the modified APS trace was submitted to OpenStack and the LCRC trace was submitted to Torque. We measured the following qualities:

- Average utilization, defined as usage over time
- Mean batch wait time, defined as the time between when the job is submitted and when it starts running
- Number of on-demand rejections, or *reject rate*, calculated as the ratio between number of rejections and number of requests

Table 1 summarizes the results of this experiment in both static and dynamic configurations. The shaded column in the static section reflects the existing scenario in which the APS and LCRC clusters are separate: the LCRC cluster has 1002.8 minutes mean batch wait time and the APS cluster has no rejections. A hypothetical scenario where 100% of the combined resources are devoted to batch workload shows that the lower bound of batch wait time for this trace is 122.5 min.

In the dynamic section of the table, we see the results of seven scenarios reflecting different combinations of parameters R and W. We notice that the utilization of the combined cluster improves by 4.8 to 5.6% across all dynamic scenarios, with mean batch wait time decreasing by 85 to 88%; this is due to the fact that we can now utilize the previously idle nodes of the dedicated on-demand cluster. However, there are 30 on-demand rejections when we choose $R = 0$ and $W = 0$; we can decrease them by increasing either one of the parameters or both. From a practical perspective, the most interesting observation is that the challenging week yielded no rejections for $R = 12$ nodes which corresponds to roughly 18% of the on-demand cluster: this means that under the basic balancer algorithm, we could reduce our investment in hardware for the on-demand cluster by 82% and still have all on-demand requests satisfied. Further, the mean batch wait time under this scenario is almost the same as the lower bound for the

TABLE I: Experimental results for the most challenging week: there are 24,177 batch jobs and 141 on-demand leases being submitted in each experiment. The wait time is measured in minutes and the reserve values are given in nodes. For the dynamic case, the on-demand and batch utilization refer to the portion of utilization coming from on-demand and batch requests respectively.

	Static (Baseline)			**Dynamic**						
Parameter settings	**Dedicated batch nodes**			**W**						
	372	304	0	0	5	10	0	5	10	0
	Dedicated on-demand nodes			**R**						
	0	68	372	0			6			12
Combined utilization	84.4%	80.1%	1.25%	84.9%	85.7%	85.7%	85.3%	85.3%	85.3%	85.3%
Batch utilization	84.4%	78.8%	NA	84.5%	84.4%	84.4%	84.0%	84.1%	84.0%	84.0%
On-demand utilization	NA	1.25%	1.25%	0.38%	1.25%	1.25%	1.25%	1.25%	1.25%	1.25%
Batch wait time (min)	122.5	1002.8	NA	122.0	147.0	147.0	150.0	140.6	150.4	130.0
Rejections	141	0	0	30	3	3	1	1	0	0

combined cluster established in the static column; this brings significant benefits to the batch side as well.

Another scenario with no rejections occurs for $R = 6$ and $W = 10$; this means that an on-demand request would execute within 10 minutes which is a relatively long wait time in the context of this use case. This has been deemed not useful for our problem formulation and thus we don't explore on-demand wait time further in our experiments.

B. Evaluating Balancer Algorithms

We next asked the question: how does our system perform in a generalized scenario? What would happen if the on-demand or on-availability workloads were different—larger or composed of a different mix of applications than in our real-life scenario? In general, we sought to discover the relationship between the cluster capacity, the type of workload, and configuration parameters or Balancer algorithms we would need to employ to accommodate the on-demand workload while running the on-availability workload undisturbed. We answered these questions by generating synthetic traces representing both on-demand and batch workloads and running experiments with those traces. In order to preserve continuity with our real-life experiments, we continue to use the 372-node cluster as a base and each experiment represents one week.

1) Generating Synthetic Batch Workloads: We create five synthetic batch workloads as follows:

The Mainstream workload (U66-Main) represents the "mainstream" workload condition in the LCRC cluster. The workload is derived by randomly sampling 1% of all the jobs in the LCRC traces. We retain the node number, walltime, and runtime of each job. Each job's submission time is calculated as the time offset from the beginning of the week which the job is selected from, so that they add up to one week's worth of submissions. Since the Mainstream workload has a lower utilization compared to the real-life workload described in Section 3.1 (66.5% versus 78.8%), we also generated workloads with higher utilizations of 77% and 88%, and named them U77-Main, and U88-Main. Specifically, we generate higher utilization workloads by injecting additional jobs into the U66-Main workload.

The Wide workload (U66-Wide) is designed to model a workload composed of relatively large parallel jobs. We derive the Wide workload directly from the mainstream workload (U66-Main) by doubling the number of nodes of each job and randomly removing approximately half of the jobs to maintain close to the same aggregate utilization.

The Narrow workload (U66-Narrow) is designed to represent a workload composed of small parallel jobs. To generate it, we split each job from the mainstream workload (U66-Main) into two smaller jobs, each requiring half the number of nodes (thus the utilization stays the same).

Table 2 summarizes the job statistics of synthetic batch workloads.

TABLE II: Job statistics of 5 synthetic batch workloads.

	U66-N	U66-M	U66-W	U77-M	U88-M
Avg. Nodes	1.6	2.9	5.7	2.9	2.9
Std. Nodes	3.4	6.6	12.7	6.5	6.2
Avg. Runtime	62.6	64.1	60.8	64.8	65.4
Std. Runtime	293.6	303.1	299.2	310.7	302.1

2) Generating Synthetic On-demand Workloads: Similar to the synthetic batch workloads, we create synthetic on-demand workloads by abstracting workload patterns from real-life traces. In particular, we see to preserve their burstiness corresponding to periods when an APS experiment occurs causing the demand for time-sensitive computation. We thus reuse the VM leases' submission times and durations from the challenging week. Since the utilization (denoted by ρ) of the challenging week's on-demand workload is 1.25%, in order to achieve higher utilization, we multiply the number of nodes in the lease by 2x, 4x, 8x, 16x, and 24x. Thus, the ρ of the synthetic workloads equals to 2.5%, 5%, 10%, 20%, and 30% respectively. These synthetic workloads preserve the burstiness quality in real-life trace while exerting much higher pressure on the Balancer. For example, when $\rho = 30\%$, the peak arrival rate of on-demand requests is 264 requests per minute.

3) Result analysis of the basic algorithm: The experiments are similar to experiments in the previous section; we use various combinations of traces submitted to OpenStack and Torque respectively, having preloaded the cluster with running jobs to mitigate the ramp-up effect. Figure 2 shows the

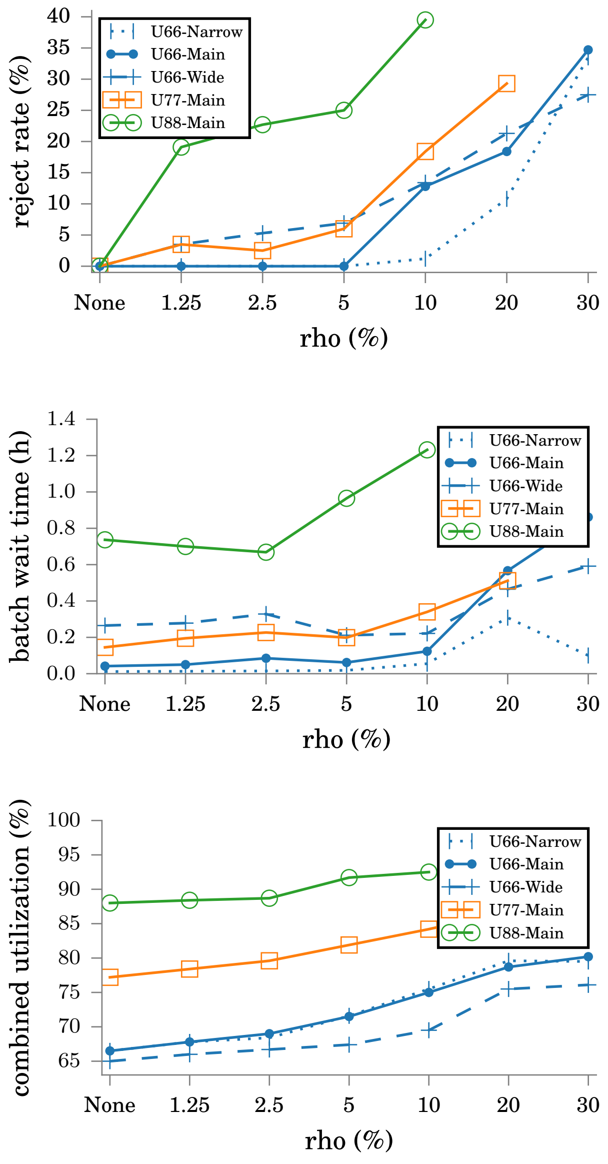

Fig. 2: Performance results of the basic algorithm, with 5 batch workloads and 6 on-demand workloads, $R = 0$, $W = 0$.

performance results of running the five batch workloads with six on-demand workloads (x-axis) with the basic algorithm (zero reserve). We skipped combinations for which the sum of batch and on-demand workloads exceeds the capacity of the cluster.

Figure 2 shows that rejection rates are influenced by both the shape of batch jobs in the trace and their density (i.e., batch utilization). While for the U66-Narrow trace we don't see rejections with ρ as high as 10%, this threshold drops to 5% as jobs become wider, and the rejection rate stays firmly above zero for every ρ value for other traces. Mean batch wait time follows a similar pattern as both smaller jobs and less utilization make it easier for batch jobs to be scheduled. Our explanation for how batch job shape affects performance is that it is easier for the batch scheduler to schedule narrower jobs than wider ones. Thus jobs finish earlier such that more space will be left open when on-demand requests arrive.

The utilization patterns follow strongly utilization of the batch traces, although all go up slightly as more on-demand jobs are added. Without any additional configuration, our approach is thus able to support on-demand workloads demanding less than 10% of cluster capacity, depending on the shape and utilization of batch jobs. To put this number in perspective, ρ from our real-life scenario was an order of magnitude lower.

To make the basic algorithm work for larger on-demand workloads, we need to use the static reserve. We thus rerun the basic algorithm with increasing R and observe the trends of rejection rate dropping. Note that since the performance is mainly determined by utilization compared to job shape, we will only use the mainstream batch workload for the rest of this paper.

Figure 3 illustrates what happens for $\rho = 10\%$. We see that a relatively small increase in the value of R (30 or 60 depending on on-demand trace density) can decrease the number of rejections by a significant factor. However, to reduce them to or near zero, we need a reserve of 120 nodes, roughly a third of the cluster. The negative effect of reserving more nodes is that the batch job performance becomes significantly worse (exponentially worse for $\rho > 10\%$). This is reflected in the combined utilization, which goes down with increased reserve for batch traces requiring higher capacity. A high static reserve is thus a very expensive solution for accommodating on-demand workloads higher than 10%; to look for a better one we turn to the hint algorithm.

4) Result analysis of the hint algorithm: To run experiments with the hint algorithm, we used a program that simulated a user notification to the Balancer 15 or 30 minutes before actual requests arrive. We used two values for this user notification: 15 minutes and 30 minutes (H15 and H30, respectively). Recall that our traces follow a real-time experimental pattern where a user would be able to make such notification.

Figure 4 shows the rejection rates for the hint algorithm. With $\rho = 10\%$ and given a 30-minute hint, we get zero rejection rates for U66-* and U77-Main and near zero ($< 1\%$) rejection rate for U88-Main. Although the H=15 results are omitted in the graph for conciseness, with a slightly shorter advance notice of 15 minutes, we get near zero ($< 1\%$) rejection rate for U66-Main and low rejection rates (less than 4%) for U77-Main and U88-Main. In comparison, the basic algorithm evaluated in the previous section needed a static reserve of 120 nodes, i.e., almost a third of the cluster to achieve the same rejection rate. A relatively accurate but short-term estimate of resource need can be then used to activate a dynamic reserve that is effectively equal to a static reserve of 120 nodes. Since a static reserve typically means purchasing and operating a cluster set aside for on-demand experimental support, this observation has significant potential for creating on-demand capacity.

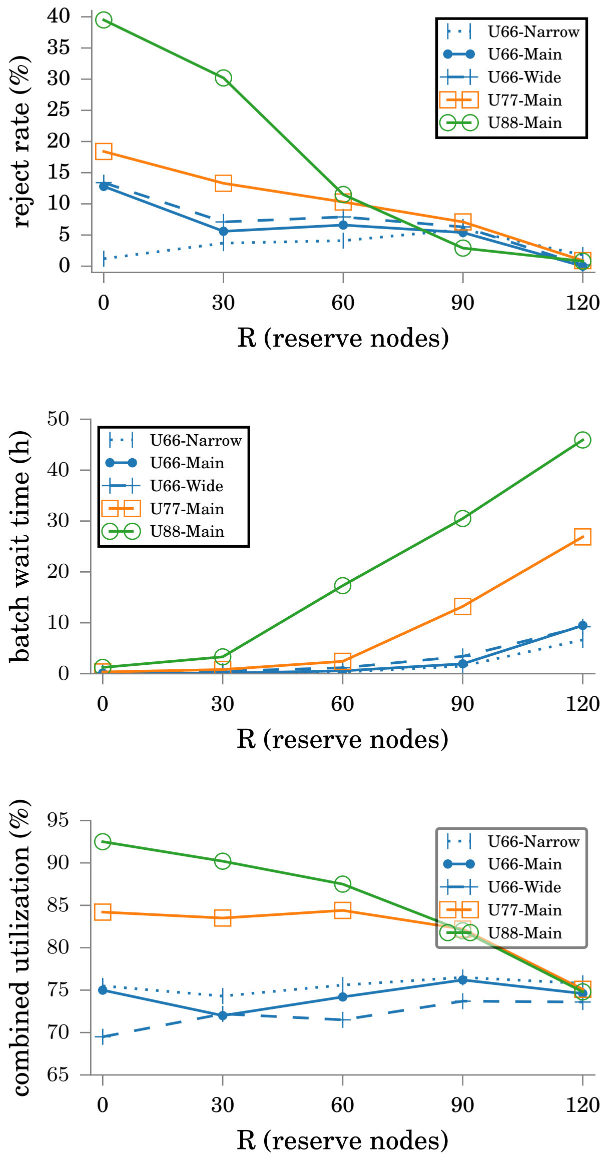

Fig. 3: Performance results of the basic algorithm with static reserve, three batch workload, $\rho = 10\%$ on-demand workloads.

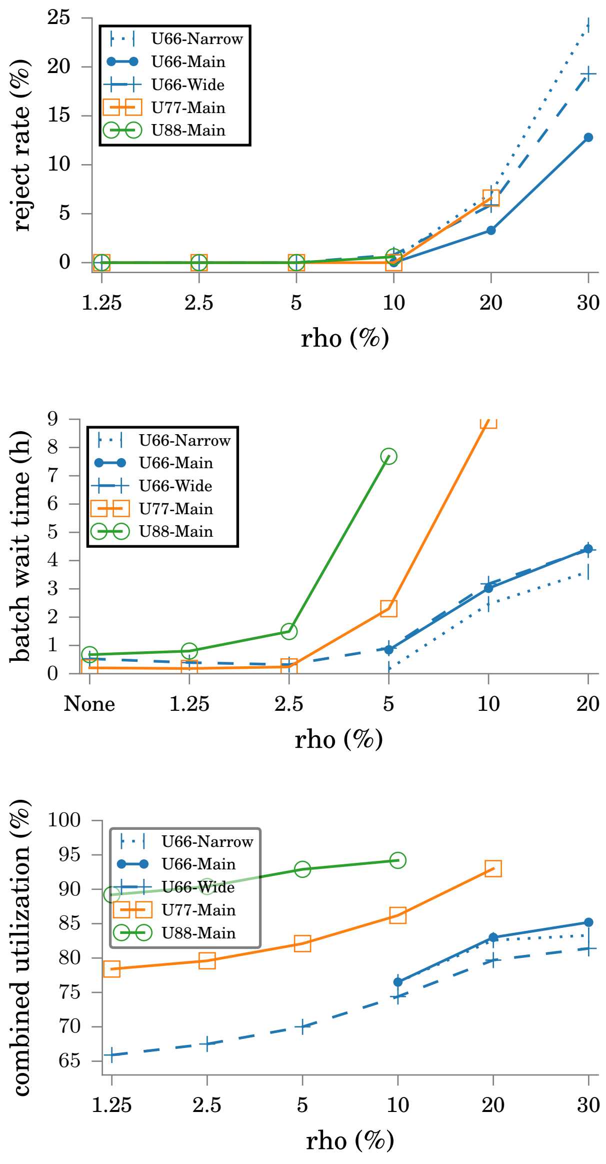

Fig. 4: Performance results of the hint algorithm, with 5 batch workloads and 6 on-demand workloads (only $H = 30$ min is displayed for conciseness).

At the same time, the impact on batch workload is much lower: for U66-Main, batch job mean wait time when $R = 120$ is over 9 hours, while the same measurement for a hint of 30 minutes is merely 50 minutes. This is because the dynamic reserve implemented by the hint algorithm acquires the nodes only when they are known to be needed and for as long as they are needed. To understand how effective it is, we looked at how much time nodes spent in reserved state without being used. Using the basic algorithm with $R = 120$; this time is 13,959 node hours whereas in the H30 case it is only 578 node hours, a reduction of 96%. This significantly increases the flexibility of the system as nodes are free to be allocated to the most pressing tasks.

Another benefit of the hint algorithm is that it improves combined utilization: most importantly, the combined utilization goes up rather than down as in the case of high reserve. In particular, batch utilization stays approximately the same, meaning that combining on-demand workload didn't hurt batch overall. The increased utilization is contributed by more on-demand workload being scheduled, e.g. the biggest boost comes from (U66-Main, ρ=30%, H30), when on-demand utilization increases by 5.1% compared to (U66-Main and ρ=30%, basic algorithm). This is also the first time the combined utilization goes above 85% (for U66 and ρ=30%)

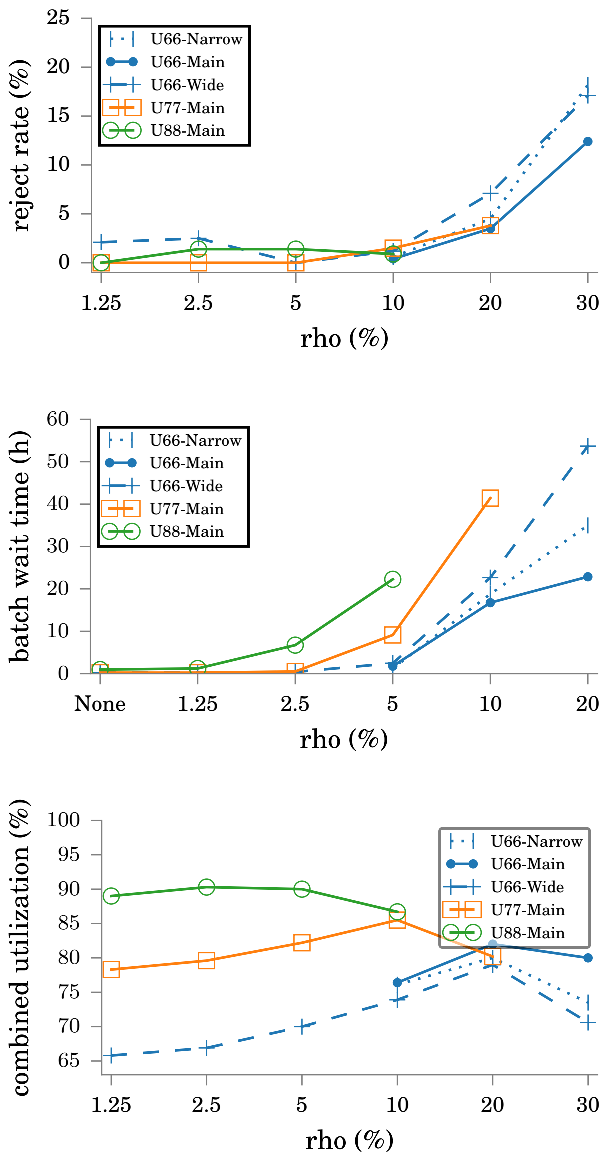

Fig. 5: Performance results of the predictive algorithm, with 5 batch workloads and 6 on-demand workloads.

demonstrating that we can indeed combine concerns of on-demand and on-availability workloads better.

5) Results analysis of the predictive algorithm: Sometimes, it is impossible to get a reliable estimate of an incoming bursty workload. In those situations, we apply a heuristic algorithm as described in section 2.3.3 to predictively adjust the reserve. To evaluate our algorithm, we first ran the predictor offline using historical data and then run live experiments using the predictor's output.

Figure 5 shows that the predictive algorithm performs equally well as the hint algorithm in terms of reject rates. When on-demand workload is low (<5%), batch wait time achieved by the predictive algorithm is comparable to that of the hint algorithm. However, when on-demand workload becomes higher, batch wait time is 1-4 times longer than the hint algorithm ($H = 30$ min). This can be explained by the fact that without additional information, the predictor can only approximately estimate on-demand request arrival time. To lower reject rate, our predictive algorithm over-reserves nodes, indicating that the predictive algorithm is significantly less efficient at estimating when the nodes will be needed.

Figure 5 also shows that the differences in utilization compared to the hint algorithm results are relatively small for low ρ and are correlated primarily to batch utilization and over-reserving nodes. Thus with larger ρ and consequently more time spent in reserve, despite the increase in on-demand utilization, it is not big enough to overshadow the drop in batch utilization. Unless there is a predictor that can accurately predict user behavior and make precise estimations of on-demand request arrival, the predictive algorithm does not perform as well as the hint algorithm in balancing batch and on-demand performance, even though the predictive algorithm performs better than the basic algorithm.

IV. Related Work

Several research groups have been exploring the suitability of the cloud environment for HPC applications in terms of virtualizing HPC execution (e.g. Palacios [10]), enabling a cloud interface for grid computing (e.g. Globus [11], Magellan [12]), and using on-demand/on-availability leases [13], [14]. Other work has focused on combining HPC and cloud systems in a hybrid environment to enable cloud bursting of HPC workload from HPC clusters to the public cloud to meet deadline constraints of HPC applications [15], [16], [17]. To reduce the cost using public clouds, a number of groups have proposed using cheaper yet unreliable spot instances to reduce the cost of executing HPC applications [18], [19], [20], [21], [22], [23]. Spot instances suffer from volatility due to price fluctuation and some work has proposed prediction methods to calculate statistical availability guarantees [23]. Our work differs from the hybrid cloud paradigm in two ways. First, it bursts on-demand applications *to* HPC clusters in a controlled fashion to meet the requirements of both on-demand and HPC batch applications. Second, unpredictable start times are avoided by reclaiming resources from the on-availability HPC cluster to convert them to on-demand resources.

In the realm of executing mixed workloads, cluster and data center operators have configured resource schedulers to improve facility utilization through co-scheduling of multiple batch and latency-critical workloads. In the HPC community, prior efforts such as Marshall et al. [1] target improving utilization of private IaaS clouds by opportunistically backfilling VMs on idle nodes which are not in use by on-demand leases, allowing HTC workloads to run on backfilled VMs. The backfilled VMs do not support start time constraints and are preemptible, unlike our work. SpeQuloS [24] explores providing QoS for executing Bag-of-Tasks applications on opportunistic grids or cloud spot instances. More recent works [25], [26] aim to improve the value of reclaimed cloud capacity by providing Service Level Objectives (SLOs) or guarantees for their use. TR-spark [27] proposed a big-data analysis framework customized to exploit transient cloud servers for Spark applications. Other systems have addressed the problem of reconciling conflicts incurred at finer-grained resource sharing (e.g., [28], [29]). Elastic schedulers such as [30] enable slots to be shared across applications to meet their SLOs and improve utilization. In this environment, slots can be taken away from loosely-coupled applications at run-time (e.g. Hadoop), but this is not applicable to the HPC environment. Through combining batch and on-demand leases, the Balancer also achieves utilization improvements but within the operating constraints of an HPC environment where batch applications are first-class citizens. The Balancer does not preempt running batch jobs to enable on-demand job execution. Additionally, due to the node-exclusive requirement of most HPC applications (unlike in a data center environment), the Balancer does not consider node-level sharing between batch and on-demand workloads, thus performance conflict is not a major concern in our work.

Finally, our work differs from prior work in that it enables resources to be dynamically shared between batch and on-demand schedulers. Thus, it is not another cluster scheduler that manages only the flow of jobs, but it also manages the flow of resources from one class of service to another. Mesos [31] is most similar to our work. Mesos adopts a two-level scheduling model which (1) offers available resources to frameworks (e.g. Torque) such that a framework can either accept or reject an offer, and (2) each framework scheduler schedules its own tasks onto the accepted resources. The Mesos master plays a similar role as the Balancer in cross-framework resource allocation. However, unlike the Balancer, Mesos does not support time-bounded resource allocation nor performance-aware resource reclamation. Both mechanisms are critical in the Balancer's target environment. Similarly, Google's Borg [32] and open-source Kubernetes enable the co-scheduling of mixed workloads, but do not adhere to the specific constraints in our HPC environment, namely, that batch jobs are the main tenant and run exclusively on allocated nodes. Thus, the Balancer must operate with fewer degrees of freedom than these general-purpose schedulers. For this reason, we opted to design a new scheduling system targeted to HPC and on-demand environments.

V. Conclusions

We proposed a model reconciling the needs of on-demand and batch workloads within one system in a non-invasive way, i.e., by operating on cycle stealing rather than disrupting job execution. The model consists of a lightweight Balancer service that dynamically arbitrates resource usage between an on-demand and on-availability scheduling framework and can be adapted to existing technologies, such as OpenStack or Kubernetes for on-demand, or Torque or Slurm for batch.

Based on a real-life scenario representing two years' worth of on-demand and batch workloads at Argonne National Laboratory, we demonstrated that by using our model on existing resources we could reduce the current investment in on-demand infrastructure by 82%, while at the same time improving the mean batch wait time almost by an order of magnitude (8x). By exploring how our model behaves under various configurations and workloads, we found that it performs best in scenarios where the on-demand workload represents less than 10% of the overall capacity (our real-life usage example needed only 1.25%). When trying to increase this limit, we found that a relatively short (15 to 30 minutes) advance notice of resource need is as effective as placing a static reservation on a third of the cluster, which has significant implications for resource usage and cost. In cases when it is not possible to obtain such advance notice, a simple prediction algorithm provides a reasonable compromise, yielding near zero rejection rates with reasonable resource usage.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, under the DOE-LAB-14-1003 and the NSF under the NSF-1443080 award. Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.

References

[1] P. Marshall, K. Keahey, and T. Freeman, "Improving Utilization of Infrastructure Clouds," in *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGRID '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 205–214.

[2] OpenStack contributors, "OpenStack Open Source Cloud Computing Software," https://www.openstack.org.

[3] F. Liu and J. B. Weissman, "Elastic job bundling: An adaptive resource request strategy for large-scale parallel applications," in *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*. IEEE, 2015, pp. 1–12.

[4] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[5] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "Deconstructing Amazon EC2 spot instance pricing," *ACM Transactions on Economics and Computation*, vol. 1, no. 3, p. 16, 2013.

[6] B. Bode, D. M. Halstead, R. Kendall, Z. Lei, and D. Jackson, "The portable batch scheduler and the maui scheduler on linux clusters." in *Annual Linux Showcase & Conference*, 2000.

[7] G. Staples, "Torque resource manager," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188464

[8] I. Habib, "Virtualization with kvm," *Linux Journal*, vol. 2008, no. 166, p. 8, 2008.

[9] K. Keahey, P. Riteau, D. Stanzione, T. Cockerill, J. Manbretti, P. Rad, and R. Paul, "Chameleon: a Scalable Production Testbed for Computer Science Research," in *Contemporary High Performance Computing vol. 3. Ed. Jeff Vetter.* Springer, 2017.

[10] J. R. Lange, K. Pedretti, P. Dinda, P. G. Bridges, C. Bae, P. Soltero, and A. Merritt, "Minimal-overhead Virtualization of a Large Scale Supercomputer," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '11. New York, NY, USA: ACM, 2011, pp. 169–180.

[11] B. Allen, R. Ananthakrishnan, K. Chard, I. Foster, R. Madduri, J. Pruyne, S. Rosen, and S. Tuecke, "Globus: A case study in software as a service for scientists," in *Proceedings of the 8th Workshop on Scientific Cloud Computing*, ser. ScienceCloud '17. New York, NY, USA: ACM, 2017, pp. 25–32.

[12] L. Ramakrishnan, P. T. Zbiegel, S. Campbell, R. Bradshaw, R. S. Canon, S. Coghlan, I. Sakrejda, N. Desai, T. Declerck, and A. Liu, “Magellan: Experiences from a Science Cloud,” in *Proceedings of the 2nd International Workshop on Scientific Cloud Computing*, ser. ScienceCloud ’11. New York, NY, USA: ACM, 2011, pp. 49–58.

[13] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn, “Case Study for Running HPC Applications in Public Clouds,” in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC ’10. New York, NY, USA: ACM, 2010, pp. 395–401.

[14] I. Sadooghi, J. H. Martin, T. Li, K. Brandstatter, K. Maheshwari, T. P. P. de Lacerda Ruivo, G. Garzoglio, S. Timm, Y. Zhao, and I. Raicu, “Understanding the Performance and Potential of Cloud Computing for Scientific Applications,” *IEEE Transactions on Cloud Computing*, vol. 5, no. 2, pp. 358–371, April 2017.

[15] F. J. Clemente-Castello, B. Nicolae, R. Mayo, and J. C. Fernandez, “Performance Model of MapReduce Iterative Applications for Hybrid Cloud Bursting,” *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2018.

[16] M. Parashar, M. AbdelBaky, I. Rodero, and A. Devarakonda, “Cloud Paradigms and Practices for Computational and Data-Enabled Science and Engineering,” *Computing in Science Engineering*, vol. 15, no. 4, pp. 10–18, July 2013.

[17] G. Fox and S. Jha, “Conceptualizing a Computing Platform for Science Beyond 2020: To Cloudify HPC, or HPCify Clouds?” in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, June 2017, pp. 808–810.

[18] S. Niu, J. Zhai, X. Ma, X. Tang, and W. Chen, “Cost-effective cloud HPC resource provisioning by building Semi-Elastic virtual clusters,” in *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2013, pp. 1–12.

[19] A. Marathe, R. Harris, D. Lowenthal, B. R. de Supinski, B. Rountree, and M. Schulz, “Exploiting Redundancy for Cost-effective, Time-constrained Execution of HPC Applications on Amazon EC2,” in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC ’14. New York, NY, USA: ACM, 2014, pp. 279–290.

[20] I. Menache, O. Shamir, and N. Jain, “On-demand, Spot, or Both: Dynamic Resource Allocation for Executing Batch Jobs in the Cloud,” in *11th International Conference on Autonomic Computing (ICAC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 177–187.

[21] Y. Gong, B. He, and A. C. Zhou, “Monetary cost optimizations for MPI-based HPC applications on Amazon clouds: checkpoints and replicated execution,” in *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2015, pp. 1–12.

[22] R. Chard, K. Chard, K. Bubendorfer, L. Lacinski, R. Madduri, and I. Foster, “Cost-Aware Cloud Provisioning,” in *2015 IEEE 11th International Conference on e-Science*, Aug 2015, pp. 136–144.

[23] R. Wolski, J. Brevik, R. Chard, and K. Chard, “Probabilistic Guarantees of Execution Duration for Amazon Spot Instances,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’17. New York, NY, USA: ACM, 2017, pp. 18:1–18:11.

[24] S. Delamare, G. Fedak, D. Kondo, and O. Lodygensky, “SpeQuloS: a QoS service for BoT applications using best effort distributed computing infrastructures,” in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM, 2012, pp. 173–186.

[25] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes, “Long-term SLOs for reclaimed cloud computing resources,” in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–13.

[26] S. Shastri, A. Rizk, and D. Irwin, “Transient Guarantees: Maximizing the Value of Idle Cloud Capacity,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 85:1–85:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3015019

[27] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda, “Tr-spark: Transient computing for big data analytics,” in *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 2016, pp. 484–496.

[28] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, “Heracles: improving resource efficiency at scale,” in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 450–462.

[29] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao, “Morpheus: Towards Automated SLOs for Enterprise Clusters,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’16. Berkeley, CA, USA: USENIX Association, 2016, pp. 117–134. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026887

[30] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica, “True Elasticity in Multi-tenant Data-intensive Compute Clusters,” in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC ’12. New York, NY, USA: ACM, 2012, pp. 24:1–24:7.

[31] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, “Mesos: A Platform for Fine-grained Resource Sharing in the Data Center,” in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308.

[32] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at Google with Borg,” in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.