Stratus: cost-aware container scheduling in the public cloud

Andrew Chung
Carnegie Mellon University

Jun Woo Park
Carnegie Mellon University

Gregory R. Ganger
Carnegie Mellon University

ABSTRACT

Stratus is a new cluster scheduler specialized for orchestrating batch job execution on *virtual clusters*, dynamically allocated collections of virtual machine instances on public IaaS platforms. Unlike schedulers for conventional clusters, Stratus focuses primarily on dollar cost considerations, since public clouds provide effectively unlimited, highly heterogeneous resources allocated on demand. But, since resources are charged-for while allocated, Stratus aggressively packs tasks onto machines, guided by job runtime estimates, trying to make allocated resources be either mostly full (highly utilized) or empty (so they can be released to save money). Simulation experiments based on cluster workload traces from Google and TwoSigma show that Stratus reduces cost by 17–44% compared to state-of-the-art approaches to virtual cluster scheduling.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → *Scheduling*;

KEYWORDS

cloud computing, cluster scheduling, transient server

ACM Reference Format:
Andrew Chung, Jun Woo Park, and Gregory R. Ganger. 2018. Stratus: cost-aware container scheduling in the public cloud. In *Proceedings of SoCC '18: ACM Symposium on Cloud Computing , Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 14 pages.
https://doi.org/10.1145/3267809.3267819

1 INTRODUCTION

Public cloud computing has matured to the point that many organizations rely on it to offload workload bursts from traditional on-premise clusters (so-called "cloud bursting") or even to replace on-premise clusters entirely. Although traditional cluster schedulers could be used to manage a mostly static allocation of public cloud virtual machine (VM) instances,[1] such an arrangement would fail to exploit the public cloud's elastic on-demand properties and thus be unnecessarily expensive.

A common approach [15, 36, 38, 54] is to allocate an instance for each submitted task and then release that instance when the task completes. Although straightforward, this new-instance-per-task approach misses significant opportunities to reduce cost by packing tasks onto fewer and perhaps larger instances. Doing so can increase utilization of rented resources and enable exploitation of varying price differences among instance types.

What is needed is a *virtual cluster (VC) scheduler* that packs work onto instances, as is done by traditional schedulers, without assuming that a fixed pool of resources is being managed. The concerns for such a scheduler are different than for traditional clusters, with resource rental costs being added and queueing delay being removed by the ability to acquire additional resources on demand rather than forcing some jobs to wait for others to finish. Minimizing cost requires good decisions regarding which tasks to pack together on instances as well as when to add more instances, which instance types to add, and when to release previously allocated instances.

Stratus is a scheduler specialized for virtual clusters on public IaaS platforms. Stratus adaptively grows and shrinks its allocated set of instances, carefully selected to minimize cost and accommodate high-utilization packing of tasks. To minimize cost over time, Stratus endeavors to get as close as possible to the ideal of having every instance be either 100% utilized by submitted work or 0% utilized so it can be immediately released (to discontinue paying for it). Via aggressive use of a new method we call *runtime binning*, Stratus groups and packs tasks based on when they are predicted to complete. Done well, such-packed tasks will fully utilize an instance, complete around the same time, and allow release of the then-idle instance with minimal under-utilization. To avoid extended retention of low-utilization instances due to mispredicted runtimes, Stratus migrates still-running tasks to clear out such instances.

Stratus's scale-out decisions are also designed to exploit both instance type diversity and instance pricing variation (static and dynamic). When additional instances are needed in the virtual cluster in order to immediately run submitted tasks, Stratus requests instance types that cost-effectively fit sets of predicted-completion-time-similar tasks. We have found that achieving good cost savings requires considering packings of pending tasks in tandem with the cost-per-resource-used of instances on which the tasks could fit; considering either alone before the other leads to many fewer <packing, instance-type> combinations considered and thereby higher costs. Stratus co-determines how many tasks to pack onto instances and which instance types to use.

Simulation experiments of virtual clusters in AWS spot markets, driven by cluster workload traces from Google and TwoSigma, confirm Stratus's efficacy. Stratus reduces total cost by 25% (Google) and 31% (TwoSigma) compared to an aggressive state-of-the-art non-packing task-per-VM approach [47]. Compared to two state-of-the-art VC schedulers that combine dynamic virtual cluster scaling with job packing, Stratus reduces cost by 17–44%. Even with static instance pricing, such as is used for AWS's on-demand instances as

[1] We use "instance" as a generic term to refer to a virtual machine resource rented in a public IaaS cloud.

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA

ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267819

well as Google Compute Engine and Microsoft Azure, Stratus reduces cost by 10–29%. Attribution of Stratus's benefits indicates that significant value comes from each of its primary elements—runtime-conscious packing, instance diversity-awareness, and under-utilization-driven migration. Further, we find that the combination is more than the sum of the parts and that failure to co-decide packing and instance type selection significantly reduces cost savings.

This paper makes four primary contributions. **(1)** It identifies the unique mix of characteristics that indicate a role for a new job scheduler specialized for virtual clusters (VCs). **(2)** It describes how runtime-conscious packing can be used to minimize under-utilization of rented instances and techniques for making it work well in practice, including with imperfect runtime predictions. **(3)** It exposes the inter-dependence of packing decisions and instance type selection, showing the dollar cost benefits of co-determining them. **(4)** It describes a batch-job scheduler (Stratus) using novel packing and instance acquisition policies, and demonstrates the effectiveness of its policies with trace-driven simulations of two large-scale, real-world cluster workloads.

2 BACKGROUND AND RELATED WORK

Job scheduling for clusters of computers has a rich history, with innovation still occurring as new systems address larger scale and emerging work patterns [16, 19, 22, 24, 31, 32, 41, 46, 55, 56]. Generally speaking, job schedulers are the resource assignment decision-making component of a cluster management system that includes support for detecting and monitoring cluster resources, initiating job execution as assigned, enforcing resource usage limits, and so on. Users submit jobs consisting of one or more tasks (single-computer programs that collectively make up a job) to the cluster management system, often together with resource requests for each task (e.g., how much CPU and memory is needed). The job scheduler will decide when and on which cluster computer to run each task of the job. Each task is generally executed in some form of *container* for resource isolation and security purposes.

Stratus is a cluster scheduler aimed to schedule batch processing workloads (e.g., machine learning model training, parallel test-suites, and distributed ETL workloads such as MapReduce [18] and Spark [59]) on virtual clusters (a "VC scheduler"). This paper describes how Stratus reduces cost by exploiting public clouds' effectively-unbounded virtual cluster elasticity, instance type diversity, and rental price variation. This section overviews relevant aspects of public IaaS cloud offerings and discusses related work.

2.1 Cloud service provider offerings

IaaS instance types and contracts. *Cloud service providers (CSPs)* offer an effectively infinite (from most customers' viewpoints) set of VM instances available for rental at fine time granularity. Each CSP offers diverse VM instance "types", primarily differentiated by their constituent hardware resources (e.g., core counts and memory sizes), and leasing contract models.

The two primary types of contract model offered by major CSPs [3, 6, 8] are *on-demand* and *transient*. Instances leased under an *on-demand* contract are non-preemptible. Instances leased under a *transient contract* are usually much cheaper, but can be unilaterally revoked by the CSP at any time. The price of on-demand instances are usually fixed for long periods of time, whereas the price of transient instances may frequently vary over time.

In AWS EC2 [3], instances leased under transient contracts are termed *spot instances*. Prices of spot instances are dictated by a *spot market* [4], which fluctuates over time but typically remains 70–80% below the prices of corresponding on-demand instances [28]. To rent a spot instance, a user specifies a *bid price*, which is the maximum price s/he is willing to pay for that instance. The spot instance can be *revoked* at any moment, but this rarely occurs when using common bidding strategies (e.g., bidding the on-demand price) [47].

Autoscaling of virtual clusters. There are two parts to autoscaling a virtual cluster (VC): determining the capacity to scale to and picking the right set of instances to scale to said capacity.

CSPs offer VC management frameworks (e.g., Amazon EC2 Spot Fleet [9]) for choosing and acquiring instances to scale based on a user-specified strategy, up to a target capacity. Available strategies in Spot Fleet include *lowestPrice* (always add new instances with the currently lowest spot price) and *diversified* (add new instances such that the diversity in the spot VM pool increases).

Determining the VC's target capacity can be done by the VC scheduler reactively, scaling up whenever a new job's tasks cannot be run on existing resources. Whereas forecasting the right target capacity is needed for web services to prevent violation of SLOs in which the tolerable latency is on the order of seconds or less [21, 27, 39, 49], the cluster workloads targeted by job schedulers are more forgiving. Even compared to the ideal of always being able to immediately start every new job, we observe reactive VC scaling provides reasonable job latencies (see Sec. 5).

Assigning containerized tasks to instances. Container services enable containerized user application tasks to be run on a public cloud. There are two dominant flavors of container management services available: *server-based* and *container-based*.

In the server-based model [1], users provide a pool of instances (e.g., via Spot Fleet) while the container service schedules tasks on to available VMs according to a configured placement policy. For example, available task placement policies in Amazon ECS [2] include *binPack* (place task on to instance with least amount of available resource), *random*, and *spread* (round-robin). Server-based container services, in the VC scheduling context, are responsible for packing containerized tasks on to VM instances.

In the container-based model [5], the container service automatically manages container placement, execution, and all underlying infrastructure. A user is billed in terms of resources consumed by the container. As explained further in Section 4, this approach is currently significantly more expensive than using a server-based model with a virtual cluster of spot instances for substantial cluster workloads.

2.2 Related work

Private cluster schedulers. Private clusters generally have a fixed set of machine composition, with whatever hardware heterogeneity was present at deployment time. Existing state-of-the-art schedulers [17, 19, 22, 24, 29, 31, 32, 41] frequently optimize scheduling decisions based on the existing set of instances. But, public clouds offer instances of many types and sizes, allowing a virtual cluster

to vary over time not only in size but in composition, so the best-match instance type for a given job can usually be acquired when desired. Naturally, different instance types have different rental prices, which must be considered. Further complicating decision-making is the fact that rental costs for particular instance types can vary over time, most notably in the AWS spot markets (as discussed below). Such differences require VC schedulers to focus on different issues than traditional cluster schedulers.

Task-per-instance virtual cluster schedulers. Most previous work on scheduling jobs on public cloud resources maps each task of each job to an instance, acquired only for the duration of that task. This approach works both for *cloud bursting* configurations [15, 25, 54], in which excess load from a private cluster is offloaded onto public cloud resources, and full virtual cluster configurations.

Various policy enhancements have been explored for task-per-instance schedulers. Mao et al. [36, 38] proposed framework-aware VC scheduling techniques that balance job deadlines and budget constraints. Niu et al. [40] discuss scheduling heuristics to address AWS's previous hour-based billing model by reusing instances for new tasks with time remaining in a paid-for hour. HotSpot [47] addresses (exploits) the dynamic nature of spot markets and the diversity of instance types, always allocating the cheapest instance on which a new task will fit and migrating tasks from more expensive instances to cheaper instances as spot market prices fluctuate.

Packing VC schedulers. Compared to the common approach of assigning a single-task-per-instance in existing VC scheduling literature, schedulers that *pack* tasks (from the same or different jobs) onto instances may reduce overall cost, as they reduce the risk of lower utilization due to imperfect fit.

One reasonable approach [10] is to pack containerized tasks on an elastic VC using CSP-offered services like those discussed above. Specifically, one can use server-based container services (e.g., ECS) to place containerized tasks on to (spot) instances, while maintaining the pool of running instances with an instance management frameworks (e.g., SpotFleet). This combination essentially results in a packing VC scheduler, and it is one of the approaches to which we compare Stratus in Section 5.

SuperCloud is a system that enables application migration across different clouds [48], and it includes a subsystem (SuperCloud-Spot) used for acquiring and packing spot instances [30]. SuperCloud-Spot appears to be designed primarily for a fixed set of long-running jobs (e.g., services), since methods for on-line packing to address dynamic task arrival/completion and varied task CPU/memory demands were not discussed. But, it represents an important step toward effective VC scheduling, and we include it in our evaluations. We also evaluate natural extensions to it as part of understanding the incremental benefits of Stratus's individual features.

Energy-conscious scheduling. Energy-conscious schedulers attempt to reduce the energy consumption of a cluster by actively causing some machines to be idle and powering them down. To do so, they attempt to pack tasks onto machines as tightly as possible to minimize the number that must be kept on [13, 14, 35]. This goal draws a parallel to the goal of VC schedulers, whose primary objective is to minimize the cluster's bill typically by using less instance-time and packing instances more efficiently. Acquiring/releasing a VM instance in the cloud is akin to switching on/off a physical machine.

Although energy-conscious schedulers and VC schedulers share a goal of maximizing utilization of active machines, energy-conscious schedulers generally do not address the opportunities created by instance heterogeneity or price variation aspects of VC scheduling. Strictly focusing on task packing, however, the closest scheme to Stratus is a scheduler proposed by Knauth et al [33], which packs VMs onto physical machines based on pre-determined runtimes (rental durations). Unlike that scheduler, Stratus does not have known runtimes, but it does exploit runtime predictions to pack tasks expected to finish around the same time.

3 STRATUS

Stratus is a VC scheduler designed to achieve cost-effective job execution on public IaaS clouds. Stratus combines a new elasticity-aware packing algorithm (Sec. 3.2) with a cost-aware cluster scaler (Sec. 3.3) that exploits instance type diversity and instance pricing variation. Stratus reduces cost in two ways: (1) by aligning task runtimes so (ideally) all tasks on an instance finish at the same time, allowing it to transition quickly from near-full utilization to being released and (2) by selecting which new instance types to acquire during scale-out in tandem with task packing decisions, allowing it to balance the cost benefits of instance utilization and time-varying instance prices. This section describes the design and implementation of Stratus.

3.1 Architecture

This section presents the architecture and key components of Stratus (Fig. 1) and walks the reader through the lifetime of a job processed by Stratus. Stratus acts as the scheduling component of a runtime environment, such as a YARN or Kubernetes cluster. The *Resource Manager* (RM) (e.g., YARN RM/Kubernetes master) is still responsible for enforcing scheduling decisions in the environment.

Jobs submitted to the VC are processed as follows:

(1) Job requests are submitted by users and received by the Resource Manager (RM). A job request contains the number of tasks to be launched and the amount of resource required to execute each task.

(2) If a job is admitted, the RM spins off task requests from the job and dispatches them to the Stratus *RM Proxy*. The RM Proxy is responsible for receiving state events (e.g., new task request, task failure, task completion, etc.) from the RM and routing them to the scheduler.

(3) The *scheduler* consists of the *packer* (Sec. 3.2) and the *scaler* (Sec. 3.3). The *packer* decides which tasks get scheduled on which available instances. The *scaler* determines which and when VM instances should be acquired for the cluster as well as when task migrations need to be performed to handle task runtime misalignments (Sec. 3.4). Given a task request from the RM Proxy, the packer puts the task request into the scheduling queue. Pending tasks are scheduled in batches during a periodic *scheduling event*; the frequency of the scheduling event is configurable.

(4) The packer and scaler make scheduling and scaling decisions based on task runtime estimates provided by a *Runtime Estimator*.

(5) If there are tasks that cannot be scheduled on to any available instances in the cluster, the packer relays the tasks along with their runtime estimates to the scaler, which decides on the instances

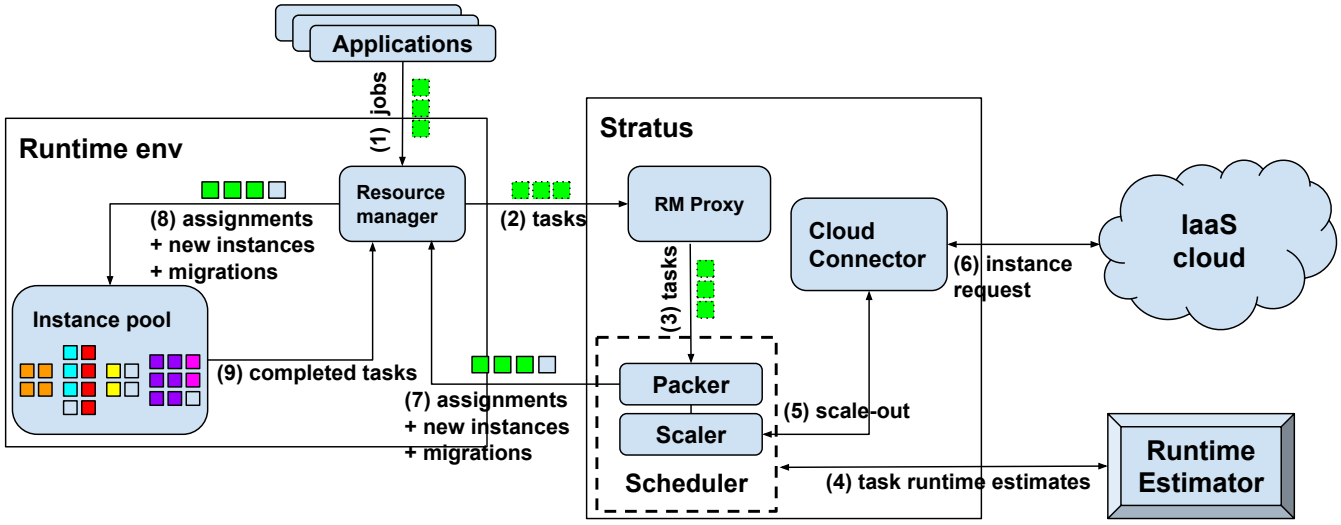


Figure 1: Stratus architecture

to acquire for these tasks. The scaler sends the corresponding instance requests to the *Cloud Connector*, which is the pluggable cloud-provider-specific module that acquires and terminates instances from the cloud for Stratus.

(6) The Cloud Connector translates the request and asynchronously calls IaaS cloud platform APIs to acquire new instances. When new instances are ready, the Cloud Connector notifies the packer via an asynchronous callback.

(7) The scheduler informs the RM of task placement decisions, availability of new instances, and tasks to migrate at the end of a scheduling event.

(8) The RM enforces task placements and adds new instances to its pool of managed instances.

(9) After tasks complete on instances, completion events are propagated to the RM. A job is completed when the RM receives all task completion events of the job's tasks, and its task runtimes are reported to Runtime Estimator to update job run history (Sec. 3.4).

3.2 Packer

This section describes the on-line packing component of Stratus, which places newly arriving tasks on to already-running instances. The Scaler (Sec. 3.3), which decides which new instances to acquire based on the packing properties of tasks that cannot be packed on to running instances, uses a compatible scheme.

3.2.1 Setup. The primary objective of Stratus is to minimize the cloud bill of the VC, which is driven mostly by the amount of resource-time (e.g., VCore-hours) purchased to complete the workload. Thus, the packer aims to pack tasks tightly, aligning remaining runtimes of tasks running on an instance as closely as possible to each other; otherwise, some tasks will complete faster than others and some of the instance's capacity will be wasted.

Packer input. The inputs to the packer are:

(1) *Queue of pending task requests*, where each task request contains the task's *resource vector* (VCores and memory), estimated runtime, priority, and scheduling constraints (e.g., anti-affinity, hardware requirements, etc.).

(2) *Set of available instances.* For each instance, Stratus tracks the amount of resource available on the instance and the remaining runtimes of each task assigned to the instance (*i.e.*, time required for the task to complete).

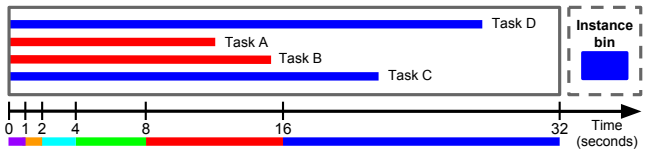
Runtime binning. The packer maintains logical bins characterized by disjoint runtime intervals. Each bin contains tasks with remaining runtimes that fall within the interval of the bin. Similarly, an instance is assigned to a bin according to the *remaining runtime of the instance*, which is the longest remaining runtime of the tasks assigned to the instance. In both cases, the boundaries of the intervals are defined exponentially, where the interval for the i^{th} bin is $[2^{i-1}, 2^i)$. For ease of discussion, we compare runtime bins according to the upper-bound of their defined runtime intervals—*i.e.*, the smallest bins are bins with runtime intervals [0, 1), [1, 2), [2, 4), …, and so on.

3.2.2 Algorithm description. At the beginning of a scheduling event, the packer organizes tasks and instances into their appropriate bins. Tasks are then considered for placement in descending order by runtime—longest task first. For each task, the Packer attempts to assign it to an available instance in two phases: the *up-packing* phase and the *down-packing* phase.

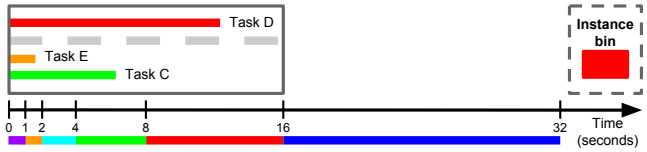
Up-packing phase. In placing a task, the packer first looks at instances from the same bin as the task. If multiple instances are eligible for scheduling the task, the packer chooses the instance with the remaining runtime closest to the runtime of the task.

If the task cannot be scheduled on any instance in its native runtime bin, the packer considers instances in progressively *greater* bins. If there are multiple candidate instances from a greater bin, the task is assigned to the instance with the most available resources (as opposed to assigning to the instance with the closest remaining runtime). The reasoning is to leave as much room as possible in the instance, which will increase the chance of being able to schedule

(a) Tasks A, B, and C scheduled. All tasks in [16, 32) bin. Instance in [16, 32) bin. One empty slot.



(b) Time progresses and tasks A and B move down to [8, 16) bin. Task C remains in [16, 32) bin. Task D scheduled on instance in [16, 32) bin. Instance remains in [16, 32) bin. Instance is full.



(c) Time progresses further and tasks A and B finish. Task C moves down to [4, 8) bin. Task D moves down to [8, 16) bin. Task E is *up-packed* to the instance and placed in [1, 2) bin. Instance moves down to [8, 16) bin. One empty slot.

Figure 2: Toy example showing how runtime binning works with the scheduling of tasks on to an instance over time (subfig. a–c). This simple example assumes all tasks are uniformly sized, and that the instance can hold four tasks in total. The solid gray box outlines the instance. Runtime bins are color-coded (*e.g.*, blue and red represent bins [16, 32) and [8, 16), respectively). Bars inside the instance represent tasks assigned to it. Task bars are color-coded to the bins they are assigned to. The dotted box shows the runtime bin that the instance assigned to.

tasks from the same bin on to the instance when tasks arrive in the future. If the task cannot be scheduled on any instance, the packer proceeds to examine instances in the *next-greatest* bin until all instances in greater bins have been examined.

While up-packing can cause instance runtime misalignments as Stratus attempts to pack tasks with shorter runtimes on instances with greater remaining runtimes, it also increases utilization of instances in greater bins and prevents the acquisition of new instances for small and short tasks when there are enough resources to run them on already-acquired instances. Up-packing minimally disrupts the scheduling opportunities of tasks of greater bins arriving in the future, as the *up-packed* task uses only half of the remaining runtime on the instance. Fig. 2 shows a toy example of runtime binning in the up-packing phase on a single instance over time.

Down-packing phase. After all greater bins have been examined for VMs to schedule the task on, the Packer examines progressively *lesser* bins for a suitable VM in the *down-packing* phase. If there are multiple candidate VMs from a lesser bin Stratus, like when up-packing, finds the VM with the most available resources that the task fits on. Down-packing the task *promotes* the VM to the task's native runtime bin.

While promoting an instance may cause task runtime misalignments on an instance, it is counter-intuitively beneficial in practice. Since tasks with similar runtimes and resource requests are often submitted concurrently/in close-succession for batch data processing jobs, promoting a large, poorly-packed instance may allow for more opportunities to fully utilize the instance with unscheduled tasks of such a job—especially because the need to down-pack implies that VMs that satisfy the current task's resource requirement be found neither in the task's native runtime bin nor in greater runtime bins. Promoting an instance also increases the chance of better utilizing the instance in later scheduling cycles, since tasks are always up-packed prior to being down-packed. Furthermore, if task runtimes are already inaccurate, it is likely that some of the tasks assigned to an instance in fact belong in some greater bin, especially if an instance is large. If a promoted instance remains under-utilized, instance clearing (Sec. 3.4) can then be used to de-allocate the instance and redistribute tasks to their rightful bins.

3.3 Scaler

When Stratus does not have enough instances to accommodate all tasks in a scheduling event, it scales out immediately and acquires new instances for the unscheduled tasks. Stratus's process of deciding which instances to acquire is iterative. It decides on a new instance to acquire at the end of each iteration, assigns unscheduled tasks to the instance, and continues until each unscheduled task is assigned to some new instance.

During scale-out, Stratus considers task packing options together with instance type options, seeking to achieve the most cost-efficient combination. In each iteration, it considers unscheduled tasks in each bin in descending order of runtime bins. The scaler constructs several candidate groups of tasks to be placed on the new instance. Each candidate group is assigned a *cost-efficiency score* for each possible instance type. The candidate group with the greatest cost-efficiency score is assigned to its best-scoring instance type, which is acquired and added to the virtual cluster.

Considering both in tandem is crucial to achieving high cost-efficiency. Doing either (task packing or instance type selection) in isolation, and then doing the other, results in too many missed opportunities—selecting instance types first leads to lower utilization of selected instances due to poor packing fits, whereas packing tasks first excludes opportunities to exploit dynamic price variations by limiting the instance sizes that make sense. Stratus's iterative approach balances the complexity of the potentially massive search space of combinations with the importance of exploring varied points in that space.

Candidate task groups. Candidate task groups are constructed so that the i^{th} group contains the first i tasks in the list sorted in descending runtime order. The first group contains the longest task, the second group contains the two longest tasks, and so on. The scaler continues to build candidate task groups until the aggregate resource request of the largest task group exceeds that of the largest allowed instance type.

Cost-efficiency score. The cost-efficiency score, computed as

$$score = \frac{\text{normalized used constraining resource}}{\text{instance price}},$$

for each candidate <task group, instance> pair, evaluates the resource efficiency of the placement of a candidate task group on a candidate instance relative to the cost of the instance.

To find the *normalized used constraining resource*, we first find the constraining resource type by computing the utilization for each resource type (VCores, memory) as if the task group is assigned to the instance. The resource type yielding the greatest utilization is the constraining resource type. Knowing the constraining resource type, the amount of constraining resource requested by the task group is the used constraining resource. Finally, we normalize the used constraining resource by the amount of resource of the constraining resource type available on the smallest instance type that we can acquire to obtain the normalized used constraining resource. The normalized used constraining resource is used to facilitate comparisons across <task group, instance> pairs with different constraining resource types.

For example, if a candidate task group requests 4 VCores and 1 GiB of memory and a candidate instance has 8 VCores and 16 GiB of memory, the constraining resource type would be VCores (4 VCores / 8 VCores > 1 GiB / 16 GiB), the used constraining resource would be 4 VCores. Assuming that the smallest instance type that we can acquire provides 2 VCores, the normalized used constraining resource would be 2 (= 4 VCores / 2 VCores).

Intuitively, if only a single resource dimension is considered, acquiring the instance with the greatest cost-efficiency score is equivalent to acquiring the instance with the lowest *cost-per-resource-used*.

At the end of each scale-out iteration, the candidate (task group, instance) pair with the best cost-efficiency score is chosen, and the corresponding task group is scheduled on to the instance. If there remains any unscheduled tasks, the scaler begins another iteration to place the rest of the tasks and continues until all tasks are scheduled.

Scaling in. There are two opportunities when Stratus terminates instances: (1) when an instance does not have any tasks assigned to it, and (2) when it continuously experiences low utilization, in which case its tasks are migrated off of it (Sec. 3.4).

Runtime interval bin selection. With a sufficient amount of tasks in a runtime bin, more instance scale-out options become available—specifically for instances that are larger, potentially more cost-efficient, and less prone to resource fragmentation. As often observed [20, 44], job and task runtime distributions are frequently long-tailed. We therefore define runtime bin intervals exponentially to enable a principled way to group tasks with runtimes at the tail while not sacrificing packing efficiency for tasks that fall in lesser interval bins. Defining runtime bins exponentially also allows us to bound the number of bins without having to specify statically-sized runtime intervals or determine a particular best bin size.

Bidding strategy and instance revocations. Stratus *does not* try to take advantage of the refund policy of spot instance revocations, where spot instances revoked within the first hour are fully refunded; rather, it focuses only on attaining cost-efficiency by exploiting cost-per-resource dynamicity[2]. Stratus uses a safe instance bidding scheme, where it always bids for an instance at its corresponding *on-demand* price. Shastri et al. [47] found that bidding the on-demand price for Spot instances result in very long times-to-revocation (25 days on average). Our experiments confirm their observation, as only a single spot price-spike was experienced in all our experiments.

[2]In the EC2 spot market, the cost-per-resource (e.g., VCore) of instances changes frequently. For *m4* instances in *us-west-2* only, the sorted order of cost-per-VCore changes 850 times/day (Aug. to Sept. 2017).

3.4 Runtime estimates

Runtime Estimator Runtime Estimator is the component that provides runtime estimates from a queryable task runtime estimate system for tasks submitted to Stratus. The topic of estimating job and task runtimes has been researched extensively [34, 50–52], and Stratus does not attempt to innovate on this front; instead, we obtained a copy of JVuPredict [52] and modified it to predict average task runtime rather than job runtime.

JVuPredict's algorithm works as follows: For each incoming job, JVuPredict identifies candidate groups of similar jobs in job execution history based on job attributes (e.g., submitted by same user, same job name submitted during the same hour of day,…, etc). For each group, several candidate estimates are produced by applying estimators (average, median,…, etc) to the average task runtimes of all jobs in the group. JVuPredict associates the estimate produced by the attribute-estimator pair that historically performs best (measured by normalized median absolute error) to the incoming job.

Handling runtime misestimates. The accuracy of task runtime estimates plays a large role in Stratus's packing algorithm. While Stratus's use of exponentially-sized runtime bins already tolerates some degree of task runtime misestimates, it is beneficial to incorporate more specialized methods to deal with larger misestimates. Stratus uses two heuristics to mitigate the impact of task runtime misestimates on cost:

Heuristic 1: Task runtime readjustment. In adjusting for task runtime under-estimates, Harchol-Balter et al. [26] observed that the probability that a process with age T seconds lasts for at least another T seconds is approximately 1/2. Stratus thus readjusts task runtime underestimates by assuming that the task has already run for half of its runtime.

Heuristic 2: Instance clearing. Stratus migrates tasks away from instances that continuously (e.g., for more than three scheduling events of one minute each in our experiments) experience low resource utilization due to task runtime mis-alignments of various scales such that they can be terminated safely without losing task progress. We define such an instance as one whose resources are less than 50% utilized in each dimension, since this is often when all tasks on an instance can be migrated to a smaller instance based on how many CSPs size their VMs [3, 6, 8].

VM candidates are evaluated for clearing in decreasing order of cost-per-resource-used. For each VM candidate, either all or none of its tasks are migrated—if an instance only ends up partially-migrated, its utilization decreases while the VC operator still has to pay the same amount of money to keep the instance running; therefore, whenever an instance is selected to be migrated, it is placed on a blacklist such that no new tasks can be scheduled on to it. For each task on a VM candidate, Stratus attempts to re-pack the tasks on to currently running instances using the packing algorithm described in Sec. 3.2. If no suitable instance is found, Stratus may also choose to acquire a new, potentially smaller/cheaper instance on which to place all of a candidate's tasks. Stratus computes the

Instance/ Server/ VM	A bundle of resources rented from the IaaS platform, generally in the form of a virtual machine (e.g., Amazon EC2).
Container	An isolated environment deployable in an instance, e.g., a nested-VM or Linux container.
Resource	Instance hardware resources, for example, VCores and memory.
Resource util	Aggregate percent instance resources *allocated* to tasks.
Task	The smallest logical unit of a computation, typically executed in a single container.
Job	A collection of tasks that perform a computation submitted by the user of a cluster.
Runtime bin	Logical bin defined by a time interval, consisting of a set of instances whose task runtimes are estimated to continue to run for less than the upper bound of the time interval. Used by Stratus to assign tasks of similar runtimes on to instances.

Table 1: Summary of terms used.

tradeoff of clearing an instance before executing the task migrations. Stratus only clears an instance if the predicted runtime for the instance's longest task is greater than the estimated migration time (plus spin-up time, in the case of new instance acquisitions).

4 EXPERIMENTAL SETUP

We use simulation-based experiments to evaluate Stratus and other VC scheduling approaches in terms of dollar cost, resource utilization, and job latency. This section describes our experimental setup.

4.1 Environment

Simulator. We built a high-fidelity event-based workload simulator that takes as input a job trace (Sec. 4.2) and a Spot market trace for each allowed instance type (discussed below). It simulates instance allocation and job placement decisions made by evaluated schedulers (Sec. 4.3), advancing simulation time as jobs arrive and complete. The simulator includes instance spin-up delays consistent with observations on AWS [37, 47], drawing uniformly from instance spin-up times ranging from 30 to 160 seconds. Container migration times are computed based on the container's memory footprint and a transfer rate of 160MBps for container memory [47]. To simulate the effect of spot market price movements, including the very rare spot instance revocations,[3] we use price traces provided by Amazon [11] spanning a three month period starting from June 5th, 2017.

Instance types and regions available. We limit our experiments to use instances of the same family in EC2 (*m4* instances) in order to (1) avoid unknown performance comparisons among compute resources[4] and (2) justify runtime estimates produced by JVuPredict, as JVuPredict does not consider tasks' runtime environments (e.g., underlying VM configuration) when generating runtime estimates[5]. We list the amount of resources available in each of the instance types in Fig. 3c, and we assume that valid instance requests are always fulfilled. We limit instance allocations to the *us-west-2* region, because migrations and data transfers across regions incur significant cost.

VM acquisition/termination. For all evaluated schedulers, (1) instances are bid for at or above (HotSpot) the on-demand price, and (2) an instance is voluntarily released to the CSP when no more tasks are running on it.

4.2 Workload traces

Our experiments use two traces from production clusters. Fig. 3a shows their task runtime distributions. For each trace, our evaluations use twenty 1-day ranges of the trace, starting at random points within the traced period. We filter out jobs that start before the trace start time and jobs that end after the trace end time. In addition to avoiding inclusion of partial jobs, this filtering removes long-running services from the Google trace, allowing the evaluation to focus on interactive and batch jobs.

Google trace. The Google trace [44, 45], released in 2011, records jobs run on one of Google's production clusters with 12.5k machines spanning a period of 29 days. The amount of requested resources for each task has been obfuscated by Google, with each dimension re-scaled to have a value between 0 and 1 based on the largest capacity of the resource available on any machine in the trace. In our simulations, for each task resource dimension, we scale the requests to the largest corresponding resource dimension of instance types used (64 VCores and 256 GiB).

We observe the following job/task properties in the filtered Google trace: (1) Tasks are typically CPU-heavy (*i.e.*, tasks are limited by the CPU dimension when scheduled), (2) the number of tasks per job is very small—in fact, more than 75% of the jobs contain less than 10 tasks, and (3) tasks are short, with most shorter than two minutes.

TwoSigma trace. The TwoSigma trace [12] contains 3.2 million jobs and was collected on two private computing clusters of Two Sigma, a quantitative hedge fund, over a nine-month period from Jan. to Sept. 2016. The clusters consist of a total of 1313 machines with 24 CPU cores and 256GiB RAM each. The majority of jobs in the TwoSigma trace are batch-processing jobs that analyze financial data with home-grown data analysis applications or Spark [59] programs. The workload does not contain any long-running services.

We observe the following job/task properties in the TwoSigma trace: (1) tasks typically have substantial memory footprints, (2) number of tasks per job is greater than in the Google trace, and (3) tasks are longer on average compared to tasks from the Google trace.

Runtime predictor performance. We report the task runtime estimate error profiles of the modified JVuPredict (Sec. 3.4) for both traces in Fig. 3b. The estimates are less accurate in the TwoSigma trace compared to the Google trace because the estimate quality largely depends on (1) the ability of JVuPredict to identify similar jobs in the history and (2) the variability of runtimes within the

[3] Spot instance revocation can be determined from the spot market price trace, because they occur when the market price exceeds the bid price. We use the common approach of bidding the on-demand price and, like others, observe that revocation is very infrequent [28, 47].

[4] Amazon used to report ECU as a unifying measurement to describe the CPU performance across varying instance types, but ECU measurements have since slowly disappeared from EC2's documentation, presumably due to the difficulty in summarizing the compute power of different instance types in a single number.

[5] The Runtime Estimator is a pluggable component which can be extended to use runtime estimates produced by a more sophisticated runtime estimator that is VM configuration aware [57].

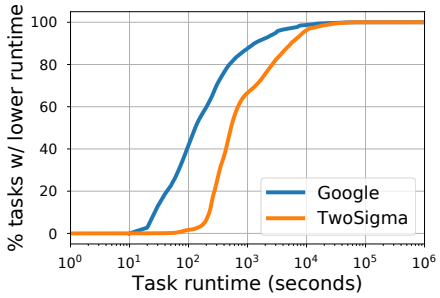


Figure 3(a): Task runtime distributions of the Google and TwoSigma traces. The time axis is plotted in log-scale.

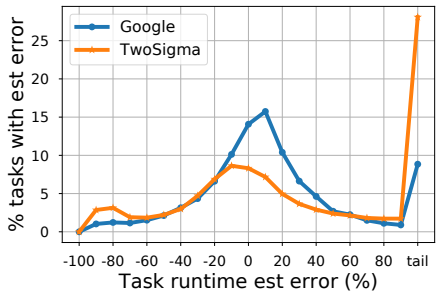


Figure 3(b): PMF of the task runtime estimation error of the modified JVuPredict.

Instance type	VCores	Memory
m4.large	2	8GiB
m4.xlarge	4	16GiB
m4.2xlarge	8	32GiB
m4.4xlarge	16	64GiB
m4.10xlarge	40	160GiB
m4.16xlarge	64	256GiB

Figure 3(c): The resource capacity of each instance type.

group of similar jobs. TwoSigma trace is reported [43] to be inferior on both measures.

Assumptions We make the following assumptions about jobs and tasks in our simulation workloads: (1) tasks can be migrated without losing progress potentially using checkpoint-restore solutions such as CRIU [7], (2) tasks do not have any hard placement constraints other than (for some) anti-affinity, (3) there are no inter-task dependencies in the workloads, and (4) decisions regarding task co-location have minimal impact on task runtimes.

Of the above, (1) is recommended practice in implementing distributed applications, and (2) and (3) arise in part from the obfuscation of production data in the traces. We believe that (4) is a reasonable premise for two reasons. First, interference effects between two tasks co-located on a VM instance would likely still be present if they each ran in their own smaller instance, because smaller instances similarly share physical hardware. Second, some co-location effects are already reflected in the runtimes recorded in the traces, since both traced clusters co-locate tasks.

4.3 Approaches evaluated

Our experiments compare Stratus against several alternative solutions. Each solution is implemented as closely as possible to its respective source documentation. This section introduces these approaches and modifications made to adapt them to the problem of minimizing the cost of running a workload where tasks have multi-dimensional resource requests and varying runtimes.

HotSpot: HotSpot (Sec. 2.2) is a single-task-per-instance VC scheduler that always chooses the cheapest instance type on which a new task will fit and will migrate the task if a different instance type becomes cheaper before it completes. We build ***HSpot***, a VC scheduler that implements HotSpot's migration and scaling policies, and enhance it with perfect runtime knowledge (unlike Stratus's imperfect predictions) so it can evaluate the tradeoff between cost added due to migration overhead vs. cost reduction for running on the new cheaper instance.

Spot Fleet + ECS: A reasonable way to place containerized tasks on to Spot instances is to use one of Amazon's ECS container placement strategies in combination with EC2 Spot Fleet [10], which acquires and releases Spot instances based on the allocation policy specified by the user. We build a scheduler ***Fleet*** that uses the most cost-efficient VM acquisition policy in Spot Fleet (*lowestPrice* [9]), in tandem with the most cost-efficient packing strategy in ECS (*binpack* [2]). Sec. 2.1 provides an overview of Spot Fleet and ECS, along with their respective policies.

SuperCloud Spot instances: SuperCloud-Spot (Sec. 2.2) is a packing VC scheduler specifically designed for scheduling nested VMs on Spot instances. We build ***SCloud***, implementing features as closely as possible to what was documented in the paper describing SuperCloud-Spot [30].

SCloud uses SuperCloud-Spot's greedy packing algorithm on the most-constrained resource type rather than its dynamic programming (DP) algorithm, which cannot be generalized to tasks of different sizes and with multiple resource request dimensions (a known NP-hard problem [42]).

SuperCloud-Spot's original migration scheme is designed for AWS's previous hour-based billing model; SuperCloud-Spot therefore makes sub-optimal decisions when computing the trade-off to re-pack tasks to new instances as it assumes no extra cost for leaving instances running as long as the instance-hour has not yet expired. So, we enhance SCloud with both HSpot's migration scheme that is suited for instances that are charged per-second and perfect task runtime knowledge.

AWS Fargate: AWS Fargate is a service that allows users to run containerized workloads without having to manage VM servers. We evaluated Fargate as an alternative to manually deploying VM clusters and running tasks on top of it. As Fargate charges on-demand prices per-resource plus a premium for managing containers for users, we posit the Fargate-based solution to be much more expensive than any other VC scheduling alternatives. Indeed, simulated experiments show that at its current price-point (May 2018), Fargate costs on average 4.4× more than HSpot. Our discussions thus focus on the Spot VC schedulers introduced above.

5 EXPERIMENTAL RESULTS

This section evaluates *Stratus*, yielding four key takeaways. First, Stratus is adept at reducing the VC cloud bill, such as by 25–31% compared to the non-packing VC scheduler (HSpot). Second, Stratus's runtime binning and tandem-consideration of task packing and instance selection allows it to reduce VC cost by 17–44% over the other packing-based VC schedulers (Fleet and SCloud). Third, each of Stratus's key techniques is important to achieving its cost reductions. Fourth, Stratus's instance clearing technique is beneficial and necessary in VC scheduling when runtime estimates are inaccurate.

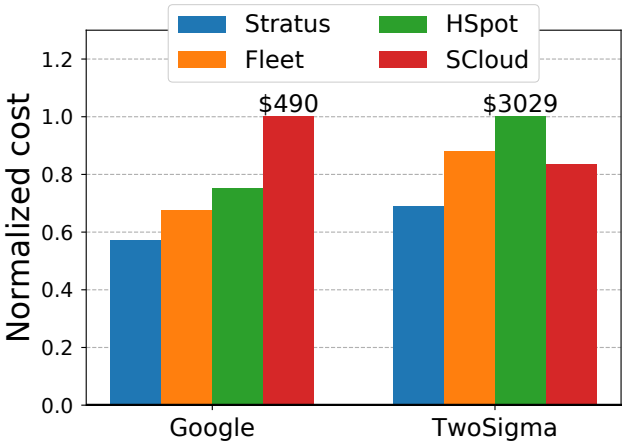


Figure 4: Average daily cost for each VC scheduler on the Google and TwoSigma workloads, normalized to the most costly option for the given trace. Stratus reduces the cost of other schedulers by at least 17% in both traces.

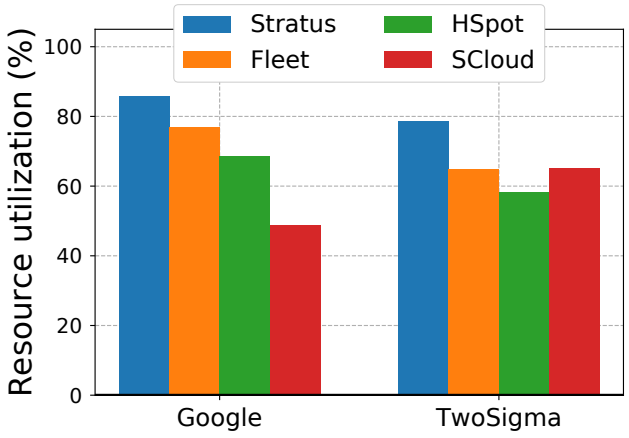


Figure 5: Constraining resource utilization (VCores for Google and memory for TwoSigma) with the different VC schedulers.

5.1 Stratus vs state-of-the-art

This section compares Stratus against existing VC scheduling solutions such as HSpot, SCloud, and Fleet.

Cost reduction. Fig. 4 shows the average daily costs of scheduling the Google and TwoSigma workloads for each VC scheduler, normalized to the most expensive case for each trace. Stratus outperforms the other VC schedulers by combination of its alignment of task runtimes and coordinated task packing and instance type selection.

Stratus outperforms HSpot by reducing the cloud bill by 25% (Google) and 31% (TwoSigma) through continuously packing newly arriving tasks on to cost-effective instances. Stratus also reduces cost by 44% (Google) and 17% (TwoSigma) compared to SCloud. While ideas from SuperCloud-Spot may have been well-suited for long-running services, it does not carry over well to workloads where task runtimes can greatly vary (Google). SCloud's scaling algorithm often bids for large VMs to reduce fragmentation and improve cost-per-resource at the time of packing. However, if task runtimes on the VM are misaligned, the large VMs acquired by SCloud will often be under-utilized as tasks on the VM complete. Because SCloud does not specify how newly arriving tasks are packed on to existing VMs, the resource holes will be unfilled until all tasks on the VM completes. Stratus outperforms SCloud by a smaller margin on the TwoSigma workload because (1) task runtimes on the TwoSigma workload tend to be longer and more runtime-aligned (Fig. 3a) and because (2) task runtime estimates are significantly less accurate on the TwoSigma trace (Fig. 3b).

Although Fleet utilizes on-line packing, it still incurs higher cloud bills compared to Stratus. Stratus reduces the cloud bill of Fleet by 17% (Google) and 22% (TwoSigma). Aside from Stratus's runtime binning, another primary reason as to why Fleet's use of on-line packing is not as effective is due to its use of Spot Fleet's *lowestPrice* scaling algorithm. Fleet always acquires the cheapest (and frequently the tightest-fitting) instances for newly arriving tasks, leaving little room to pack more tasks on an instance and leading to greater resource fragmentation. In addition, the cheapest instance for a task may not be the most cost-efficient instance for the set of pending tasks that are available. By considering the packing of groups of tasks and their runtime alignments *while* selecting instance types, Stratus is able to achieve lower fragmentation and acquire instances with better cost-per-resource-used.

To confirm this observation, we experimented with a version of Stratus that always selects the best-fitting instance for a new task when scaling out (akin to Fleet) while using runtime binning. Consistent with our observation that schedulers that always acquire best-fitting or cheapest-fitting instances for individual tasks have little opportunity to pack, we observe the cost for Stratus increases by 17% (Google) and 25% (TwoSigma), only slightly beating Fleet in both traces.

Resource utilization. Much of Stratus's cost reduction comes from increased utilization of rented resources. Fig. 5 shows the utilization of the constraining resource, VCore in the case of the Google workload and memory for TwoSigma, for the four VC schedulers.

Stratus attains higher resource utilization than the other VC schedulers, achieving 86% and 79% utilization, respectively, for the two workloads. Stratus's high resource utilization results from its combination of aligning task runtimes in tasks packed onto a given instance, acquiring instances of suitable sizes, and judicious use of instance clearing to avoid retaining under-utilized instances on which most tasks already completed. Importantly, Stratus's selection of instance types during scale-out in light of different possible packing configurations, rather than only considering packing after selection, significantly increases utilization. At the same time as both, Stratus considers instance pricing differences per-resource-used, resulting in the overall cost reductions described above.

Job latency. We define *normalized job latency* as the observed job latency normalized to an idealized job runtime that incurs no scheduling or instance spin-up delays. Table 2 shows the 50^{th} and 95^{th} percentile normalized job latencies for each compared scheduler on each trace.

Overall, we observe that schedulers that always acquire new instances for tasks (SCloud and HSpot) incur greater normalized job latency than those that pack (Fleet and Stratus) on workloads

Scheduler	Google 50%-ile	Google 95%-ile	TwoSigma 50%-ile	TwoSigma 95%-ile
Stratus	1.3	3.2	1.1	1.6
Fleet	1.2	3.1	1.0	1.4
HSpot	2.2	7.0	1.1	1.5
SCloud	2.5	11.4	1.2	2.0

Table 2: The normalized job latencies for each evaluated VC scheduler. Schedulers that pack continuously (Stratus and Fleet) incur lower job latencies than those that do not (HSpot, SCloud) when jobs are short and small (Google).

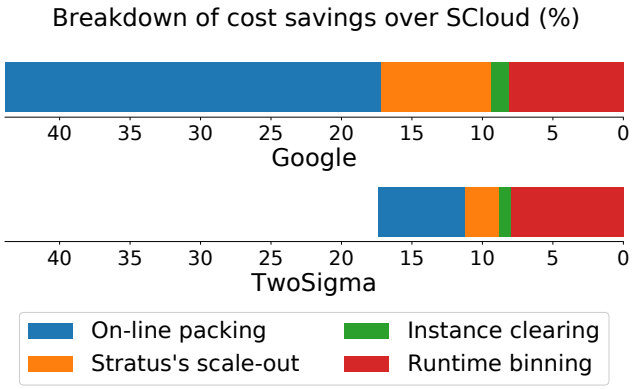


Figure 6: Break-down of Stratus's cost savings over SCloud (44% for Google and 17% for TwoSigma). The cost of running workloads reduces as Stratus features are added to SCloud, starting with features from left to right (on-line packing to runtime binning). The closer to zero, the smaller the cost difference between SCloud and Stratus.

with jobs that are mostly short and small (Google), as instance start-up delay can cause proportionally significant job slowdowns. For workloads where jobs are longer (TwoSigma), instance start-up delays are obviously less significant. And, with more memory-heavy tasks (TwoSigma), use of migration for instance clearing induces marginally higher job latencies for Stratus, because such tasks take longer to migrate.

5.2 Benefit attribution: SCloud to Stratus

Stratus uses a combination of heuristics to reduce cost. This section evaluates the incremental contributions of each by adding each to SCloud, one by one, until it matches Stratus. Fig. 6 shows the breakdown of how much of Stratus's cost savings is realized with each heuristic added to SCloud.

SCloud. We start the incremental build-up with SCloud, as described in Sec. 4.3, which only implements features as explicitly noted in the original SuperCloud-Spot paper. As discussed above (Sec. 5.1), Stratus reduces cost compared to SCloud by 44% on the Google trace and by 17% on the TwoSigma trace.

Adding online vector bin packing. To close the gap between SCloud and Stratus, we add support for packing new tasks on to running instances whenever possible, via the dot-product geometric heuristic for online vector bin-packing [23, 42] to SCloud. This technique is effective in reducing the cost of SCloud. But, there remains a cost-gap of 17% (Google) and 11% (TwoSigma) between SCloud and Stratus.

Also adding Stratus's scale-out policy. While SCloud's greedy instance acquisition algorithm is effective with tasks whose requests are uniform and tasks with resource requests only in a single dimension, it performs less well when tasks request a varying amount of resources in multiple dimensions. Using Stratus's instance acquisition scheme that considers the cost-per-resource-used of each group of tasks *assigned on to each instance* lowers the cost of SCloud with on-line packing (by 8% on the Google workload and by 3% on the TwoSigma workload). Implementing SCloud + on-line packing + Stratus's scaling heuristic closes the cost gap between SCloud and Stratus down to 9% (Google) and 8% (TwoSigma).

Also adding instance clearing. We found that incremental addition of instance clearing via migration did not help much. Interestingly, we found that instance clearing is not effective when used without taking task runtime into account. When instance clearing was used without taking task runtime into account on the enhanced SCloud, we found that cost increased on the TwoSigma trace for two reasons: (1) Although TwoSigma tasks are generally more uniform in runtime compared to Google tasks (Sec. 3a), task runtimes can become increasing mis-aligned with the introduction of instance clearing, as the runtimes of partially-run tasks may vary more significantly. Task runtime mis-alignment causes the number of instance under-utilizations to fluctuate, increasing the number of task migrations required. (2) TwoSigma tasks require more time to migrate, as they have larger memory footprints. Without knowing the cost-benefit tradeoff of clearing an instance, which depends on how much longer tasks will run, the number of task migrations can increase significantly.

Therefore, we enhance our enhanced SCloud with perfect runtime knowledge, such that it only migrates instances when the benefit in migration outweighs the task migration cost. Even with this unrealistic knowledge, instance clearing was not very effective in reducing cost in the enhanced SCloud. With previous features plus instance clearing, SCloud reduces the cost-gap by 1% (Google and TwoSigma) only. Stratus still reduces the cost by 8% (Google) and 7% (TwoSigma).

Side note: Stratus without instance clearing. In addition to evaluating SCloud enhanced with the previous features and instance clearing, we also evaluated Stratus with instance clearing disabled. The latter increases the cloud bill for Stratus by 28% for the Google trace and by 15% for TwoSigma. This shows that instance clearing is effective in assisting Stratus in putting tasks into their rightful bins, whereas it is less effective on other packing schemes where tasks are placed on to instances without regard to task runtimes. Unlike with the enhanced SCloud, less time is spent on task migration by Stratus (up to 23% less). Futher, tasks that have been migrated at least once in Stratus are only on average migrated 1.2 times before they reach an instance on which they terminate.

Also adding runtime binning. Adding Stratus's runtime binning to the rest of the enhancements to SCloud, we end up with Stratus.

5.3 Attribution: dynamic instance pricing

Stratus's scale-out policy exploits dynamic instance pricing better than the other VC schedulers, because it considers different amounts of packing as part of selecting the most cost-efficient instance types based on current prices. Even with statically-priced instances like

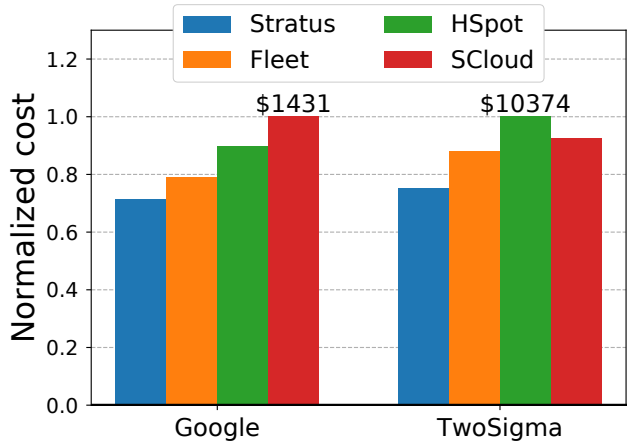


Figure 7: Average daily cost for each VC scheduler on the Google and TwoSigma workloads, using *only on-demand VMs*, normalized to the most costly option for each trace.

those offered in Google Cloud Engine and Microsoft Azure, however, Stratus's use of runtime binning and instance clearing to align co-located tasks' completion times remains beneficial.

Fig. 7 shows the average daily costs of the VC schedulers when using only on-demand instances instead of the price-varying spot instances used in our other experiments. As expected, costs are much higher for all VC schedulers, since on-demand instances are usually more expensive than spot instances. Although the differences are somewhat smaller, the relative rankings of the VC schedulers are the same, and we find that Stratus still reduces cost compared to the others—by 10–29% for Google and by 14–25% for TwoSigma.

5.4 Sensitivity to runtime est. accuracy

This section characterizes the effect of task runtime estimate accuracy on Stratus.

5.4.1 Stratus with perfect runtime knowledge. We evaluated Stratus with perfect runtime knowledge on the Google and TwoSigma workloads to see how much more Stratus could lower cost in this ideal scenario. As expected, enhancing Stratus with perfect runtime knowledge improves its cost-efficiency, further reducing cost by 5% (Google) and 9% (TwoSigma). Stratus with known runtimes reduces the cost of Stratus with JVuPredict by improving the constrained resource utilization of Stratus from 86% to 90% (Google) and from 79% to 84% (TwoSigma), and by reducing the total task migration time by 64% (Google) and by 82% (TwoSigma).

Less accurate task runtime estimates necessarily induce more task runtime misalignment for runtime-aware schedulers, leading to less effective usage of resources on instances as tasks do not complete in a coordinated manner. When task runtime estimates are inaccurate, tasks may unexpectedly complete early or late on an instance, leaving a portion of its resources idle. Although Stratus has implemented instance clearing to reduce the number of active VMs in case of under-utilization, instance clearing comes with non-trivial cost. Namely, tasks that are being migrated do not make any progress but reserve resources on both the source and the destination VMs, and during instance clearing, newly arriving tasks cannot be assigned on to the cleared VM such that the cleared VM can be terminated when its task migrations complete.

	Google		TwoSigma	
h	50%-ile	95%-ile	50%-ile	95%-ile
0.01	1.3	3.2	1.1	1.7
1	1.3	3.3	1.0	1.6
100	1.3	3.2	1.1	1.8

Table 3: Normalized job latencies for values of h (Sec. 5.4.2).

5.4.2 Runtime estimate accuracy sensitivity. Our previous experiments evaluate Stratus using a real state-of-the-art task runtime estimator and, in Sec. 5.4.1, a hypothetical runtime estimator providing perfect estimates. This section characterizes the effect of task runtime estimate accuracy on Stratus at a finer granularity by controlling the range of (synthetic) runtime estimate errors.

Setup. In each experiment, we generate runtime estimates for each task by scaling the actual runtime of the task by a factor of h_τ, where h_τ is uniformly sampled from a range of $[h, 1)$ if $h < 1$ and from $[1, h]$ if $h \geq 1$. Setting $h = 1$ is the same as using perfect task runtime knowledge. We perform 29 experiments on each trace, with each experiment consisting of five runs on different slices of the trace, for $h \in [0.01, 100]$.

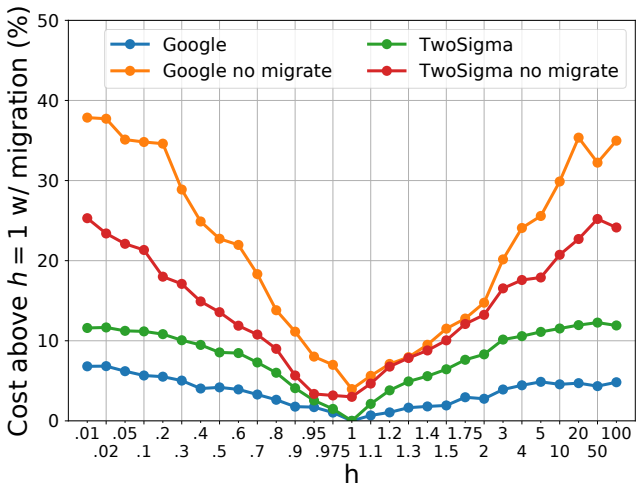
Cost trends and estimate accuracy. Fig. 8a shows Stratus's sensitivity with respect to cost to the accuracy of its task runtime estimates. As expected, cost increases as the quality of the runtime estimates degrades, whether under-estimates (to the left in the graph) or over-estimates (to the right). As runtime estimates become less accurate, Stratus makes less informed decisions in choosing which tasks to co-locate on instances.

Comparing variants of Stratus with and without instance clearing ("no migration"), we observe that instance clearing reduces the impact of runtime misestimates and misalignments. Stratus is efficient even without instance clearing when runtime estimates are accurate ($h = 1$), only incurring 4% (Google) and 3% (TwoSigma) more cost than with instance clearing. Instance clearing helps significantly when runtime estimates are increasingly inaccurate. Stratus achieves 31% (Google) and 14% (TwoSigma) lower cost at $h = 0.01$ with instance clearing than without. Cost savings at $h = 100$ with and without instance clearing is comparable.

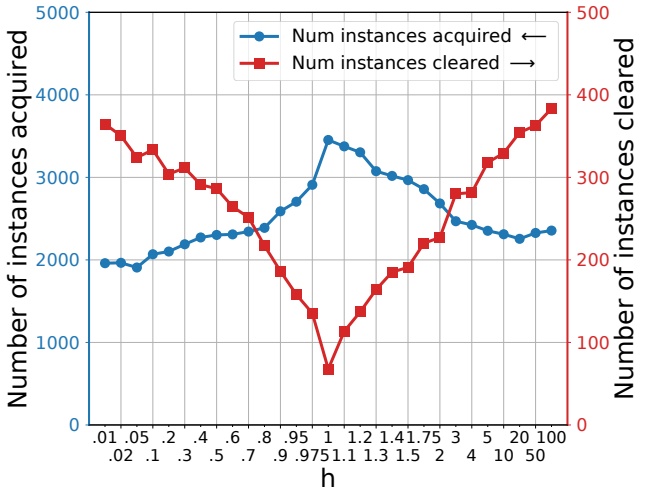
The disparate behavior between the results with and without instance clearing for the Google and TwoSigma traces stems from the different characteristics of their jobs, as discussed in Sec. 4.2: TwoSigma jobs consist of tasks longer in duration (the time that new instances remain well-packed is longer) and often have more tasks per job (there are more tasks with similar runtimes).

Instance acquisition/clearing and estimate accuracy. The blue line in Fig. 8b shows Stratus's sensitivity to task runtime estimates with respect to number of instances acquired, while the red line shows Stratus's sensitivity to task runtime estimates with respect to number of instances cleared. Table 3 shows the normalized job latencies for polar values of h on the Google and TwoSigma traces.

As task runtime estimates become increasingly accurate, opportunities to release empty instances increase as task runtimes become better-aligned, decreasing the need to migrate tasks using



(a) Stratus's cost savings on the Google and TwoSigma traces. Instance clearing is disabled on *no migrate* variants of Stratus. The cost increases as the range of potential estimate errors are expanded. Instance clearing is more beneficial when estimates are less accurate.



(b) Stratus's instance acquisition and clearing profiles on the Google trace. The number of instance acquisitions decreases as the range of potential estimate errors are expanded, while the number of instance clearings increases. Plot of results on the TwoSigma trace are comparable.

Figure 8: Experiments varying the degree of runtime estimate error in completing jobs from traces. Each experiment consists of tasks with runtime estimates set to runtime $* h_\tau$, where h_τ is uniformly sampled from $[h, 1)$ if $h < 1$ and from $[1, h]$ if $h \geq 1$.

the instance clearing heuristic. This, however, also means that as new tasks arrive there are less instances available on which to place the new tasks, leading to a greater number of instances acquired and a larger portion of job latency spent on waiting for instances to spin up.

Similarly, as task runtime estimates become less accurate, fewer instances are acquired since instances generally "stick around" for a longer period of time due to mis-aligned runtimes. Mis-aligning runtimes raises the chance to trigger instance clearing, increasing the number of instances cleared and causing the job latency to be increasingly dominated by task migration time.

Our experimental results (Table 3) show that for the Google and TwoSigma traces, the impact of increased instance spin-up time vs increased task migration time on job latency approximately balance out.

6 CONCLUSION

The Stratus cluster scheduler exploits cloud properties and runtime estimates to reduce the dollar cost of cluster jobs executed on public clouds. By packing jobs that should complete around the same time, simultaneously considering possible packings and available instance types/prices, and judicious use of task migration to clear under-utilized instances, Stratus actively avoids having leased machines that are not highly utilized. We expect Stratus's approach to be a core element of future virtual cluster management for public clouds.

ACKNOWLEDGMENTS

We thank our SoCC 2018 reviewers for valuable suggestions. We thank Alex Glikson and Aaron Harlap for their feedback. We thank the members and companies of the PDL Consortium: Alibaba, Dell EMC, Facebook, Google, HP Enterprise, Hitachi, IBM, Intel, Micron, Microsoft, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate, Toshiba, Two Sigma, Veritas and Western Digital for their interest, insights, feedback, and support. This research is supported in part by a Samsung Scholarship.

REFERENCES

[1] 2018. Amazon ECS Container Instances. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ECS_instances.html. (2018).

[2] 2018. Amazon ECS Task Placement Strategies. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-placement-strategies.html. (2018).

[3] 2018. AWS EC2. http://aws.amazon.com/ec2/. (2018).

[4] 2018. AWS EC2 Spot Instances. https://aws.amazon.com/ec2/spot/. (2018).

[5] 2018. AWS Fargate. https://aws.amazon.com/AWS/Fargate. (2018).

[6] 2018. Azure Virtual Machines. https://azure.microsoft.com/en-us/services/virtual-machines/. (2018).

[7] 2018. CRIU. http://criu.org/. (2018).

[8] 2018. Google Compute Engine. https://cloud.google.com/compute/. (2018).

[9] 2018. How Spot Fleet Works. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-fleet.html. (2018).

[10] 2018. Powering your Amazon ECS Clusters with Spot Fleet. https://aws.amazon.com/blogs/compute/powering-your-amazon-ecs-clusters-with-spot-fleet/. (2018).

[11] 2018. Spot Instance Pricing History. https://aws.amazon.com/ec2/spot/. (2018).

[12] George Amvrosiadis, Jun Woo Park, Gregory R Ganger, Garth A Gibson, Elisabeth Baseman, and Nathan DeBardeleben. 2018. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/atc18/presentation/amvrosiadis

[13] Anton Beloglazov and Rajkumar Buyya. 2010. Adaptive Threshold-based Approach for Energy-efficient Consolidation of Virtual Machines in Cloud Data Centers. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science (MGC '10)*. ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/1890799.1890803

[14] Anton Beloglazov and Rajkumar Buyya. 2012. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurrency and Computation: Practice and Experience* 24, 13 (2012), 1397–1420.

[15] Luiz Fernando Bittencourt and Edmundo Roberto Mauro Madeira. 2011. HCOC: a cost optimization algorithm for workflow scheduling in hybrid clouds. *Journal of Internet Services and Applications* 2, 3 (2011), 207–227.

[16] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing.. In *OSDI*, Vol. 14. 285–300.

[17] Carlo Curino, Djellel E Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. 2014. Reservation-based scheduling: If you're late don't blame us!. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–14.

[18] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1251254.1251264

[19] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. 2015. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference*. USENIX Association, 499–510.

[20] Dror G Feitelson. 2015. *Workload modeling for computer systems performance evaluation*. Cambridge University Press.

[21] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A Kozuch. 2012. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems (TOCS)* 30, 4 (2012), 14.

[22] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 99–115. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gog

[23] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2015. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 455–466.

[24] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. Graphene: Packing and Dependency-aware Scheduling for Data-parallel Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 81–97. http://dl.acm.org/citation.cfm?id=3026877.3026885

[25] Tian Guo, Upendra Sharma, Prashant Shenoy, Timothy Wood, and Sambit Sahu. 2014. Cost-aware cloud bursting for enterprise applications. *ACM Transactions on Internet Technology (TOIT)* 13, 3 (2014), 10.

[26] Mor Harchol-Balter and Allen B Downey. 1996. Exploiting process lifetime distributions for dynamic load balancing. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 24. ACM, 13–24.

[27] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R Ganger, and Phillip B Gibbons. 2018. Tributary: spot-dancing for elastic services with latency SLOs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/atc18/presentation/harlap

[28] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R Ganger, and Phillip B Gibbons. 2017. Proteus: agile ML elasticity through tiered reliability in dynamic resource markets.. In *EuroSys*. 589–604.

[29] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 261–276.

[30] Qin Jia, Zhiming Shen, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. 2016. Smart spot instances for the supercloud. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*. ACM, 5.

[31] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, ÃĺÃśigo Goiri, Subramaniam Venkatraman Krishnan, Jana Kulkarni, and Sriram Rao. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters, In OSDI'16. https://www.microsoft.com/en-us/research/publication/morpheus-towards-automated-slos-enterprise-clusters/

[32] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters.. In *USENIX Annual Technical Conference*. 485–497.

[33] Thomas Knauth and Christof Fetzer. 2012. Energy-aware scheduling for infrastructure clouds. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*. IEEE, 58–65.

[34] Shonali Krishnaswamy, Seng Wai Loke, and Arkady Zaslavsky. 2004. Estimating computation times of data-intensive applications. *IEEE Distributed Systems Online* 5, 4 (2004).

[35] B. Li, J. Li, J. Huai, T. Wo, Q. Li, and L. Zhong. 2009. EnaCloud: An Energy-Saving Application Live Placement Approach for Cloud Computing Environments. In *2009 IEEE International Conference on Cloud Computing*. 17–24. https://doi.org/10.1109/CLOUD.2009.72

[36] Ming Mao and Marty Humphrey. 2011. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. IEEE, 1–12.

[37] Ming Mao and Marty Humphrey. 2012. A performance study on the vm startup time in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 423–430.

[38] Ming Mao, Jie Li, and Marty Humphrey. 2010. Cloud auto-scaling with deadline and budget constraints. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*. IEEE, 41–48.

[39] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. 2013. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service.. In *ICAC*. 69–82.

[40] Shuangcheng Niu, Jidong Zhai, Xiaosong Ma, Xiongchao Tang, Wenguang Chen, and Weimin Zheng. 2016. Building Semi-Elastic Virtual Clusters for Cost-Effective HPC Cloud Resource Provisioning. *IEEE Transactions on Parallel and Distributed Systems* 27, 7 (2016), 1915–1928.

[41] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 69–84.

[42] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. 2011. Heuristics for Vector Bin Packing. (January 2011). https://www.microsoft.com/en-us/research/publication/heuristics-for-vector-bin-packing/

[43] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 2018. 3Sigma: Distribution-based Cluster Scheduling for Runtime Uncertainty. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 2, 17 pages. https://doi.org/10.1145/3190508.3190515

[44] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 7.

[45] Charles Reiss, John Wilkes, and Joseph L Hellerstein. 2011. Google cluster-usage traces: format+ schema. *Google Inc., White Paper* (2011), 1–14.

[46] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 351–364.

[47] Supreeth Shastri and David Irwin. 2017. HotSpot: automated server hopping in cloud spot markets. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 493–505.

[48] Zhiming Shen, Qin Jia, Gur-Eyal Sela, Weijia Song, Hakim Weatherspoon, and Robbert Van Renesse. 2017. Supercloud: A Library Cloud for Exploiting Cloud Diversity. *ACM Transactions on Computer Systems (TOCS)* 35, 2 (2017), 6.

[49] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 5.

[50] Ozan Sonmez, Nezih Yigitbasi, Alexandru Iosup, and Dick Epema. 2009. Trace-based evaluation of job runtime and queue wait time predictions in grids. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*. ACM, 111–120.

[51] Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. 2007. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems* 18, 6 (2007), 789–803.

[52] Alexey Tumanov, Angela Jiang, Jun Woo Park, Michael A Kozuch, and Gregory R Ganger. 2016. *JamaisVu: Robust Scheduling with Auto-Estimated Job Runtimes*. Technical Report. CMU-PDL-16-104. Carnegie Mellon University.

[53] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 35.

[54] Ruben Van den Bossche, Kurt Vanmechelen, and Jan Broeckhove. 2010. Cost-optimal scheduling in hybrid iaas clouds for deadline constrained workloads. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE, 228–235.

[55] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 5.

[56] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 18.

[57] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H. Katz. 2017. Selecting the Best VM Across Multiple Public Clouds: A Data-driven Performance Modeling Approach. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM, New York, NY, USA, 452–465. https://doi.org/10.1145/3127479.3131614

[58] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*. ACM, 265–278.

[59] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.