Quasar: Resource-Efficient and QoS-Aware Cluster Management

Christina Delimitrou and Christos Kozyrakis
Stanford University
{cdel, kozyraki}@stanford.edu

Abstract

Cloud computing promises flexibility and high performance for users and high cost-efficiency for operators. Nevertheless, most cloud facilities operate at very low utilization, hurting both cost effectiveness and future scalability.

We present Quasar, a cluster management system that increases resource utilization while providing consistently high application performance. Quasar employs three techniques. First, it does not rely on resource reservations, which lead to underutilization as users do not necessarily understand workload dynamics and physical resource requirements of complex codebases. Instead, users express performance constraints for each workload, letting Quasar determine the right amount of resources to meet these constraints at any point. Second, Quasar uses classification techniques to quickly and accurately determine the impact of the amount of resources (scale-out and scale-up), type of resources, and interference on performance for each workload and dataset. Third, it uses the classification results to jointly perform resource allocation and assignment, quickly exploring the large space of options for an efficient way to pack workloads on available resources. Quasar monitors workload performance and adjusts resource allocation and assignment when needed. We evaluate Quasar over a wide range of workload scenarios, including combinations of distributed analytics frameworks and low-latency, stateful services, both on a local cluster and a cluster of dedicated EC2 servers. At steady state, Quasar improves resource utilization by 47% in the 200-server EC2 cluster, while meeting performance constraints for workloads of all types.

Categories and Subject Descriptors C.5.1 [Computer System Implementation]: Super (very large) computers; D.4.1 [Process Management]: Scheduling

Keywords Cloud computing, datacenters, resource efficiency, quality of service, cluster management, resource allocation and assignment

1. Introduction

An increasing amount of computing is now hosted on public clouds, such as Amazon's EC2 [2], Windows Azure [65] and Google Compute Engine [25], or on private clouds managed by frameworks such as VMware vCloud [61], OpenStack [48], and Mesos [32]. Cloud platforms provide two major advantages for end-users and cloud operators: *flexibility* and *cost efficiency* [9, 10, 31]. Users can quickly launch jobs that range from short, single-process applications to large, multi-tier services, only paying for the resources used at each point. Cloud operators can achieve economies of scale by building large-scale datacenters (DCs) and by sharing their resources between multiple users and workloads.

Nevertheless, most cloud facilities operate at very low utilization which greatly adheres cost effectiveness [9, 51]. This is the case even for cloud facilities that use cluster management frameworks that enable cluster sharing across workloads. In Figure 1, we present a utilization analysis for a production cluster at Twitter with thousands of servers, managed by Mesos [32] over one month. The cluster mostly hosts user-facing services. The aggregate CPU utilization is consistently below 20%, even though reservations reach up to 80% of total capacity (Fig. 1.a). Even when looking at individual servers, their majority does not exceed 50% utilization on any week (Fig. 1.c). Typical memory use is higher (40-50%) but still differs from the reserved capacity. Figure 1.d shows that very few workloads reserve the right amount of resources (compute resources shown here, similar for memory); most workloads (70%) overestimate reservations by up to 10x, while many (20%) underestimate reservations by up to 5x. Similarly, Reiss et al. showed that a 12,000-server Google cluster managed with the more mature Borg system consistently achieves aggregate CPU utilization of 25-35% and aggregate memory utilization of 40% [51]. In contrast, reserved resources exceed 75% and 60% of available capacity for CPU and memory respectively.

Twitter and Google are in the high end of the utilization spectrum. Utilization estimates are even lower for cloud facilities that do not co-locate workloads the way Google and

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2305-5/14/03. . . $15.00.
http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2541940.2541941

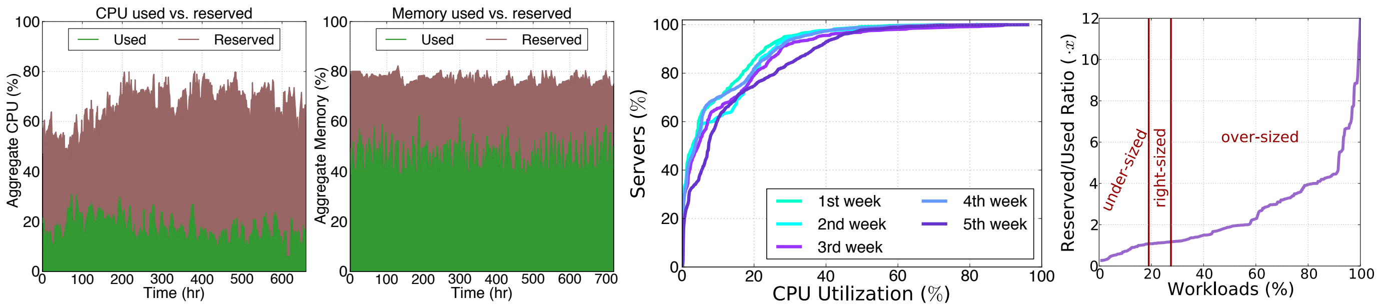

Figure 1: Resource utilization over 30 days for a large production cluster at Twitter managed with Mesos. (a) and (b): utilization vs reservation for the aggregate CPU and memory capacity of the cluster; (c) CDF of CPU utilization for individual servers for each week in the 30 day period; (d) ratio of reserved vs used CPU resources for each of the thousands of workloads that ran on the cluster during this period.

Twitter do with Borg and Mesos respectively. Various analyses estimate industry-wide utilization between 6% [15] and 12% [24, 59]. A recent study estimated server utilization on Amazon EC2 in the 3% to 17% range [38]. Overall, low utilization is a major challenge for cloud facilities. Underutilized servers contribute to capital expenses and, since they are not energy proportional [36, 42], to operational expenses as well. Even if a company can afford the cost, low utilization is still a scaling limitation. With many cloud DCs consuming 10s of megawatts, it is difficult to add more servers without running into the limits of what the nearby electricity facility can deliver.

In this work, we increase resource utilization in datacenters through better cluster management. The manager is responsible for providing resources to various workloads in a manner that achieves their performance goals, while maximizing the utilization of available resources. The manager must make two major decisions; first allocate the right amount of resources for each workload (*resource allocation*) and then select the specific servers that will satisfy a given allocation (*resource assignment*). While there has been significant progress in cluster management frameworks [21, 32, 54, 63], there are still major challenges that limit their effectiveness in concurrently meeting application performance and resource utilization goals. First, it is particularly difficult to determine the resources needed for each workload. The load of user-facing services varies widely within a day, while the load of analytics tasks depends on their complexity and their dataset size. Most existing cluster managers side-step allocation altogether, requiring users or workloads to express their requirements in the form of a reservation. Nevertheless, the workload developer does not necessarily understand the physical resource requirements of complex codebases or the variations in load and dataset size. As shown in Figure 1.d, only a small fraction of the workloads submitted to the Twitter cluster provided a right-sized reservation. Undersized reservations lead to poor application performance, while oversized reservations lead to low resource utilization.

Equally important, resource allocation and resource assignment are fundamentally linked. The first reason is heterogeneity of resources, which is quite high as servers get installed and replaced over the typical 15-year lifetime of a DC [9, 20]. A workload may be able to achieve its current performance goals with ten high-end or twenty low-end servers. Similarly, a workload may be able to use low-end CPUs if the memory allocation is high or vice versa. The second reason is interference between co-located workloads that can lead to severe performance losses [41, 69]. This is particularly problematic for user-facing services that must meet strict, tail-latency requirements (e.g., low 99^{th} percentile latency) under a wide range of traffic scenarios ranging from low load to unexpected spikes [16]. Naïvely co-locating these services with low-priority, batch tasks that consume any idling resources can lead to unacceptable latencies, even at low load [41]. This is the reason why cloud operators deploy low-latency services on dedicated servers that operate at low utilization most of the time. In facilities that share resources between workloads, users often exaggerate resource reservations to side-step performance unpredictability due to interference. Finally, most cloud facilities are large and involve thousands of servers and workloads, putting tight constraints on the complexity and time that can be spent making decisions [54]. As new, unknown workloads are submitted, old workloads get updated, new datasets arise, and new server configurations are installed, it is impractical for the cluster manager to analyze all possible combinations of resource allocations and assignments.

We present *Quasar*, a cluster manager that maximizes resource utilization while meeting performance and QoS constraints for each workload. Quasar includes three key features. First, it shifts from a *reservation-centric* to a *performance-centric* approach for cluster management. Instead of users expressing low-level resource requests to the manager, Quasar allows users to communicate the performance constraints of the application in terms of throughput and/or latency, depending on the application type. This high-level specification allows Quasar to determine the *least* *amount of the available resources* needed to meet performance constraints at any point, given the current state of the cluster in terms of available servers and active workloads. The allocation varies over time to adjust to changes in the workload or system state. The performance-centric approach simplifies both the user and cloud manager's roles as it removes the need for exaggerated reservations, allows transparent handling of unknown, evolving, or irregular workloads, and provides additional flexibility towards cost-efficient allocation.

Second, Quasar uses *fast classification techniques to determine the impact of different resource allocations and assignments on workload performance*. By combining a small amount of profiling information from the workload itself with the large amount of data from previously-scheduled workloads, Quasar can quickly and accurately generate the information needed for efficient resource assignment and allocation without the need for a priori analysis of the application and its dataset. Specifically, Quasar performs four parallel classifications on each application to evaluate the four main aspects of resource allocation and assignment: the impact of scale-up (amount of resources per server), the impact of scale-out (number of servers per workload), the impact of server configuration, and the impact of interference (which workloads can be co-located).

Third, Quasar *performs resource allocation and assignment jointly*. The classification results are used to determine the right amount and specific set of resources assigned to the workload. Hence, Quasar avoids overprovisioning workloads that are currently at low load and can compensate for increased interference or the unavailability of high-end servers by assigning fewer or lower-quality resources to them. Moreover, Quasar monitors performance throughout the workload's execution. If performance deviates from the expressed constraints, Quasar reclassifies the workload and adjusts the allocation and/or assignment decisions to meet the performance constraints or minimize the resources used.

We have implemented and evaluated a prototype for Quasar managing a local 40-server cluster and a 200-node cluster of dedicated EC2 servers. We use a wide range of workloads including analytics frameworks (Hadoop, Storm, Spark), latency-critical and stateful services (memcached, Cassandra), and batch workloads. We compare Quasar to reservation-based resource allocation coupled with resource assignment based on load or similar classification techniques. Quasar improves server utilization at steady state by 47% on average at high load in the 200-server cluster, while also improving performance of individual workloads compared to the alternative schemes. We show that Quasar can right-size the allocation of analytics and latency-critical workloads better than built-in schedulers of frameworks like Hadoop, or auto-scaling systems. It can also select assignments that take heterogeneity and interference into account so that throughput and latency constraints are closely met.

2. Motivation

2.1 Cluster Management Overview

A cluster management framework provides various services including security, fault tolerance, and monitoring. This paper focuses on the two tasks most relevant to resource efficiency: resource allocation and resource assignment of incoming workloads. Previous work has mostly treated the two separately.

Resource allocation: Allocation refers to determining the amount of resources used by a workload: number of servers, number of cores and amount of memory and bandwidth resources per server. Managers like Mesos [32], Torque [58], and Omega [54] expect workloads to make resource reservations. Mesos processes these requests and, based on availability and fairness issues [27], makes resource offers to individual frameworks (e.g., Hadoop) that the framework can accept or reject. Dejavu identifies a few workload classes and reuses previous resource allocations for each class to minimize reallocation overheads [60]. CloudScale [56], PRESS [29], AGILE [46] and the work by Gmach et al. [28] perform online prediction of resource needs, often without a priori workload knowledge. Finally, auto-scaling systems such as Rightscale [52] automatically scale the number of physical or virtual instances used by webserving workloads to react to observed changes in server load.

Resource assignment: Assignment refers to selecting the specific resources that satisfy an allocation. The two biggest challenges of assignment are server heterogeneity and interference between colocated workloads [41, 44, 69], when servers are shared to improve utilization. The most closely related work to this paper is Paragon [20]. Given a resource allocation for an unknown, incoming workload, Paragon uses classification techniques to quickly estimate the impact of heterogeneity and interference on performance. Paragon uses this information to assign each workload to server type(s) that provide the best performance and colocate workloads that do not interfere with each other. Nathuji et al. [45] developed a feedback-based scheme that tunes resource assignment to mitigate interference effects. Yang et al. developed an online scheme that detects memory pressure and finds colocations that avoid interference on latency-sensitive workloads [69]. Similarly, DeepDive detects and manages interference between co-scheduled applications in a VM system [47]. Finally, CPI2 [73] throttles low-priority workloads that induce interference to important services. In terms of managing heterogeneity, Nathuji et al. [44] and Mars et al. [40] quantified its impact on conventional benchmarks and Google services and designed schemes to predict the most appropriate server type for each workload.

2.2 The Case for Coordinated Cluster Management

Despite the progress in cluster management technology, resource utilization is quite low in most private and public clouds (see Figure 1 and [15, 24, 38, 51, 59]). There are two

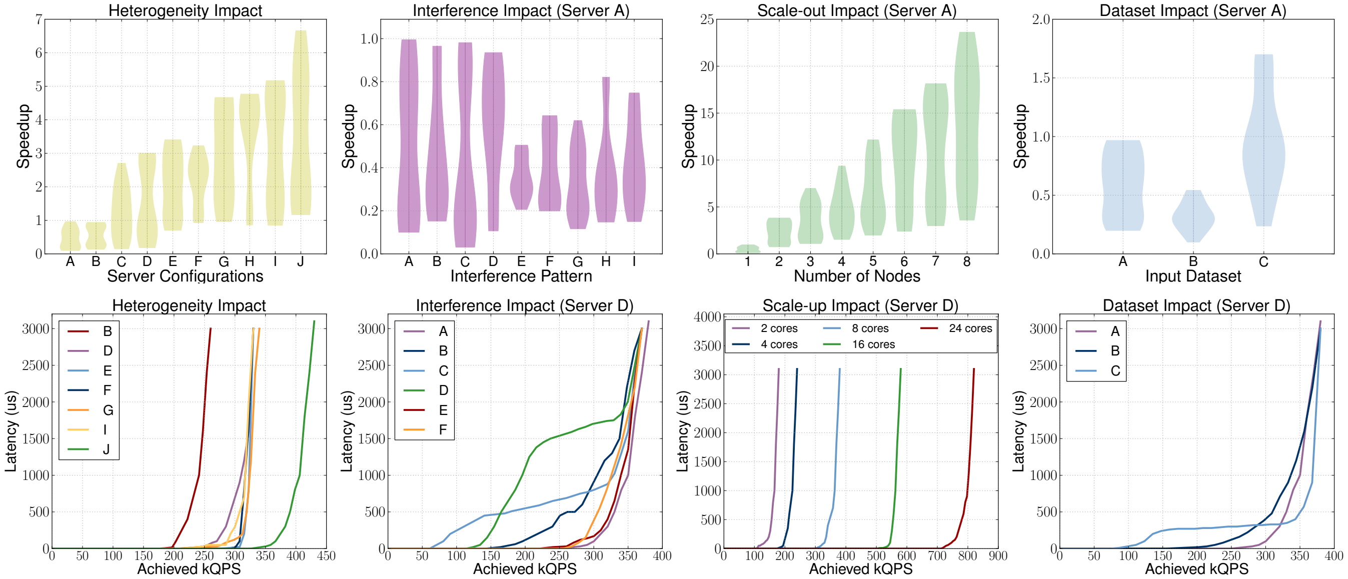

Figure 2: The impact of heterogeneity, interference, scale-out, scale-up, and dataset on the performance of Hadoop (top row) and memcached (bottom row). Server configurations, interference patterns, and datasets are summarized in Table 1. For Hadoop, the variability in the violin plots is due to scaling-up the resource allocations within a server (cores and/or memory).

platforms	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
cores	2	4	8	8	8	8	12	12	16	24
memory(GB)	4	8	12	16	20	24	16	24	48	48

interference	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
pattern	-	memory	L1I cache	LL cache	disk I/O	network	L2 cache	CPU	prefetch

input dataset	**A**	**B**	**C**
hadoop	netflix: 2.1GB	mahout: 10GB	wikipedia: 55GB
memcached	100B reads	2KB reads	100B reads-100B writes

Table 1: Server platforms (A-J), interference patterns (A-I) and input datasets (A-C) used for the analysis in Figure 2.

major shortcomings current cluster managers have. First, it is particularly difficult for a user or workload to understand its resource needs and express them as a reservation. Second, resource allocation and assignment are fundamentally linked. An efficient allocation depends on the amount and type of resources available and the behavior of other workloads running on the cluster.

Figure 2 illustrates these issues by analyzing the impact of various allocations, assignments, and workload aspects on two representative applications, one batch and one latency-critical: a large Hadoop job running a recommendation algorithm on the Netflix dataset [11] and a memcached service under a read-intensive load. For Hadoop, we report speedup over a single node of server configuration A using all available cores and memory. Server configurations, interference settings and datasets are summarized in Table 1. The variability in each violin plot is due to the different amounts of resources (cores and memory) allocated within each server. For memcached, we report the latency-throughput graphs. Real-world memcached deployments limit throughput to achieve 99th-percentile latencies between 0.2ms and 1ms.

The first row of Figure 2 illustrates the behavior of Hadoop. The heterogeneity graph shows that the choice of server configuration introduces performance variability of up to 7x, while the amount of resources allocated within each server introduces variability of up to 10x. The interference graph shows that for server A, depending on the amount of resources used, Hadoop may be insensitive to certain types of interference or slowdown by up to 10x. Similarly, the scale-out graph shows that depending on the amount of resources per server, scaling may be sublinear or superlinear. Finally, the dataset graph shows that the dataset complexity and size can have 3x impact on Hadoop's performance. Note that in addition to high variability, the violin plots show that the probability distributions change significantly across different allocations. The results are similar for memcached, as shown in the second row of Figure 2. The position of the knee of the throughput-latency curve depends heavily on the type of server used (3x variability), the interference patterns

(7x variability), the amount of resources used per server (8x variability), and workload characteristics such as data size and read/write mixes (3x variability).

It is clear from Figure 2 that it is quite difficult for a user or workload to translate a performance goal to a resource reservation. To right-size an allocation, we need to understand how scale-out, scale-up, heterogeneity in the currently available servers, and interference from the currently running jobs affect a workload with a specific dataset. Hence, separating resource allocation and assignment through reservations is bound to be suboptimal, either in terms of resource efficiency or in terms of workload performance (see Figure 1). Similarly, performing first allocation and then assignment in two separate steps is also suboptimal. Cluster management must handle both tasks in an integrated manner.

3. Quasar

3.1 Overview

Quasar differs from previous work in three ways. First, it shifts away from resource reservations and adopts a *performance-centric* approach. Quasar exports a high-level interface that allows users or the schedulers integrated in some frameworks (e.g., Hadoop or Spark) to express the performance constraint the workload should meet. The interface differentiates across workload types. For latency-critical workloads, constraints are expressed as a queries per second (QPS) target and a latency QoS constraint. For distributed frameworks like Hadoop, the constraint is execution time. For single node, single-threaded or multi-threaded workloads the constraint is a low-level metric of instructions-per-second (IPS). Once performance constraints are specified, it is up to Quasar to find a resource allocation and assignment that satisfies them.

Second, Quasar uses fast classification techniques to quickly and accurately estimate the impact different resource allocation and resource assignment decisions have on workload performance. Upon admission, an incoming workload and dataset is profiled on a few servers for a short period of time (a few seconds up to a few minutes - see Sec. 3.2). This limited profiling information is combined with information from the few workloads characterized offline and the many workloads that have been previously scheduled in the system using classification techniques. The result of classification is accurate estimates of application performance as we vary the type or number of servers, the amount of resources within a server, and the interference from other workloads. In other words, we estimate graphs similar to those shown in Figure 2. This classification-based approach eliminates the need for exhaustive online characterization and allows efficient scheduling of unknown or evolving workloads, or new datasets. Even with classification, exhaustively estimating performance for all allocation-assignment combinations would be infeasible. Instead, Quasar decomposes the problem to the four main components of allocation and assignment: resources per node and number of nodes for allocation, and server type and degree of interference for assignment. This dramatically reduces the complexity of the classification problem.

Third, Quasar uses the result of classification to jointly perform resource allocation and assignment, eliminating the inherent inefficiencies of performing allocation without knowing the assignment challenges. A greedy algorithm combines the result of the four independent classifications to select the number and specific set of resources that will meet (or get as close as possible to) the performance constraints. Quasar also monitors workload performance. If the constraint is not met at some point or resources are idling, either the workload changed (load or phase change), classification was incorrect, or the greedy scheme led to suboptimal results. In any case, Quasar adjusts the allocation and assignment if possible, or reclassifies and reschedules the workload from scratch.

Quasar uses similar classification techniques as those introduced in Paragon [20]. Paragon handles only resource assignment. Hence, its classification step can only characterize workloads with respect to heterogeneity (server type) and interference. In contrast, Quasar handles both resource allocation and assignment. Hence, its classification step also characterizes scale-out and scale-up issues for each workload. Moreover, the space of allocations and assignments that Quasar must explore is significantly larger than the space of assignments explored by Paragon. Finally, Quasar introduces an interface for performance constraints in order to decouple user goals from resource allocation and assignment. In Section 6, we compare Quasar to Paragon coupled with current resource allocation approaches to showcase the advantages of Quasar.

3.2 Fast and Accurate Classification

Collaborative filtering techniques are often used in recommendation systems with extremely sparse inputs [50]. One of their most publicized use was the Netflix Challenge [11], where techniques such as Singular Value Decomposition (SVD) and PQ-reconstruction [13, 35, 50, 66] were used to provide movie recommendations to users that had only rated a few movies themselves, by exploiting the large number of ratings from other users. The input to SVD in this case is a very sparse matrix A with users as rows, movies as columns and ratings as elements. SVD decomposes A to the product of the matrix of singular values Σ that represents similarity concepts in A, the matrix of left singular vectors U that represents correlation between rows of A and similarity concepts, and the matrix of right singular vectors V that represents the correlation between columns of A and similarity concepts ($A = U \cdot \Sigma \cdot V^T$). A similarity concept can be that users that liked “Lord of the Rings 1” also liked “Lord of the Rings 2”. PQ-reconstruction with Stochastic Gradient Descent (SGD), a simple latent-factor model [13, 66], uses Σ, U, and V to reconstruct the missing entries in A. Starting with the SVD output, P^T is initialized to ΣV^T and Q to U which provides an initial reconstruction of A. Subsequently, SGD iterates over all elements of the reconstructed matrix R=Q·P^T until convergence.

For each element r_{ui} of R:

$$\epsilon_{ui} = r_{ui} - \mu - b_u - q_i \cdot {p_u}^T$$
$$q_i \leftarrow q_i + \eta(\epsilon_{ui} p_u - \lambda q_i)$$
$$p_u \leftarrow p_u + \eta(\epsilon_{ui} q_i - \lambda p_u)$$

until $|\epsilon|_{L_2} = \sqrt{\sum_{u,i} |\epsilon_{ui}|^2}$ becomes marginal. η is the learning rate and λ the regularization factor of SGD and their values are determined empirically. In the above model, we also include the average rating μ and a user bias b_u that account for the divergence of specific users from the norm. Once the matrix is reconstructed, SVD is applied once again to generate movie recommendations by quantifying the correlation between new and existing users. The complexity of SVD is $O(min(N^2M, M^2N))$, where M, N the dimensions of A, and the complexity of PQ-reconstruction with SGD is $O(N \cdot M)$.

In Paragon [20], collaborative filtering was used to quickly classify workloads with respect to interference and heterogeneity. A few applications are profiled exhaustively offline to derive their performance on different servers and with varying amounts of interference. An incoming application is profiled for one minute on two of the many server configurations, with and without interference in two shared resources. SVD and PQ-reconstruction are used to accurately estimate the performance of the workload on the remaining server configurations and with interference on the remaining types of resources. Paragon showed that collaborative filtering can quickly and accurately classify unknown applications with respect to tens of server configurations and tens of sources of interference.

The classification engine in Quasar extends the one in Paragon in two ways. First, it uses collaborative filtering to estimate the impact of resource scale-out (more servers) and scale-up (more resources per server) on application performance. These additional classifications are necessary for resource allocation. Second, it tailors classifications to different workload types. This is necessary because different types for workloads have different constraints and allocation knobs. For instance, in a webserver we can apply both scale-out and scale-up and we must monitor queries per second (QPS) and latency. For Hadoop, we can also configure workload parameters such as the number of mappers per node, heapsize, and compression. For a single-node workload, scaling up might be the only option while the metric of interest can be instructions per second. The performance constraints interface of Quasar allows users to specify the type of submitted applications.

Overall, Quasar classifies for scale-up, scale-out, heterogeneity, and interference. The four classifications are done independently and in parallel to reduce complexity and overheads. The greedy scheduler combines information from all four. Because of the decomposition of the problem the matrix dimensions decrease, and classification becomes fast enough that it can be applied on every workload submission, even if the same workload is submitted multiple times with different datasets. Hence there is no need to classify for dataset sensitivity.

Scale-up classification: This classification explores how performance varies with the amount of resources used within a server. We currently focus on compute cores, memory and storage capacity. We will address network bandwidth in future work. We perform scale-up classification on the highest-end platform, which offers the largest number of scale-up options. When a workload is submitted, we profile it briefly with two randomly-selected scale-up allocations. The parameters and duration of profiling depend on workload type. Latency-critical services, like memcached are profiled for 5-10 seconds under live traffic, with two different core/thread counts and memory allocations (see the validation section for a sensitivity analysis on the number of profiling runs). For workloads like Hadoop, we profile a small subset (2-6) of map tasks with two different allocations and configurations of the most important framework parameters (e.g., mappers per node, JVM heapsize, block size, memory per task, replication factor, and compression). Profiling lasts until the map tasks reach at least 20% of completion, which is typically sufficient to estimate the job's completion time using its progress rate [70] and assuming uniform task duration [32]. Section 4.3 addresses the issue of non-uniform task duration distribution and stragglers. Finally, for stateful services like Cassandra [14], Quasar waits until the service's setup is complete before profiling the input load with the different allocations. This takes at most 3-5 minutes, which is tolerable for long-running services. Section 4.2 discusses how Quasar guarantees side-effect free application copies for profiling runs.

Profiling collects performance measurements in the format of each application's performance goal (e.g., expected completion time or QPS) and inserts them into a matrix A with workloads as rows and scale-up configurations as columns. A configuration includes compute, memory, and storage allocations or the values of the framework parameters for a workload like Hadoop. To constrain the number of columns, we quantize the vectors to integer multiples of cores and blocks of memory and storage. This may result into somewhat suboptimal decisions, but the deviations are small in practice. Classification using SVD and PQ-reconstruction then derive the workload's performance across all scale-up allocations.

Scale-out classification: This type of classification is only applicable to workloads that can use multiple servers, such as distributed frameworks (e.g., Hadoop or Spark), stateless (e.g., webserving) or stateful (e.g., memcached or Cassandra) distributed services, and distributed computations (e.g., MPI jobs). Scale-out classification requires one more run in

	Default density constraint: 2 entries per row, per classification												*8 entries per row*		
	scale-up			**scale-out**			**heterogeneity**			**interference**			*exhaustive*		
Classification err.	*avg*	*90 %ile*	*max*	*avg*	*90 %ile*	*max*	*avg*	*90 %ile*	*max*	*avg*	*90 %ile*	*max*	*avg*	*90 %ile*	*max*
Hadoop (10 Jobs)	5.2%	9.8%	11%	5.0%	14.5%	17%	4.1%	4.6%	5.0%	1.8%	5.1%	6%	14.1%	15.8%	16%
Memcached (10)	6.3%	9.2%	11%	6.6%	10.5%	12%	5.2%	5.7%	6.5%	7.2%	9.1%	10%	14.1%	16.5%	18%
Webserver (10)	8.0%	10.1%	13%	7.5%	11.6%	14%	4.1%	5.1%	5.2%	3.2%	8.1%	9%	16.5%	17.6%	18%
Single-node (413)	4.0%	8.1%	9%	-	-	-	3.5%	6.9%	8.0%	4.4%	9.2%	10%	11.6%	12.1%	13%

Table 2: Validation of Quasar's classification engine. We present average, 90^{th} percentile and maximum errors between estimated values and actual values obtained with detailed characterization. We also compare the classification errors of the four parallel classification to a single, exhaustive classification that accounts for all combinations of resource allocation and resource assignment jointly.

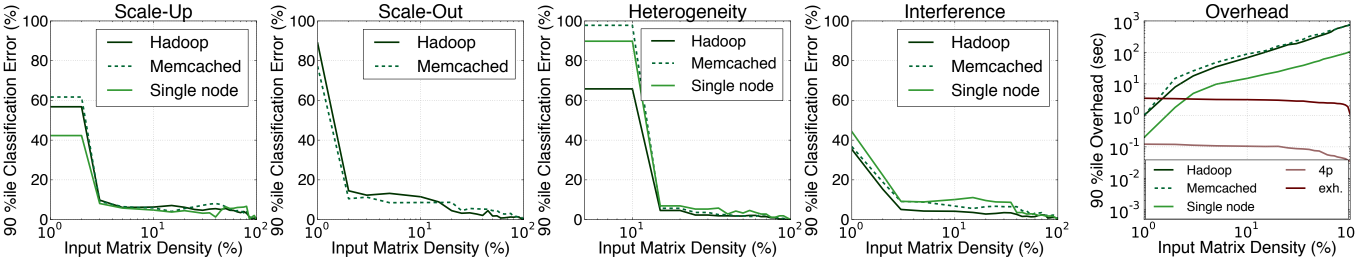

Figure 3: Sensitivity of classification accuracy to input matrix density constraints (Fig. 3(a-d)). Fig. 3e shows the profiling and decision overheads for different density constraints of the input matrices, assuming constant hardware resources for profiling.

addition to single-node runs done for scale-up classification. To get consistent results, profiling is done with the same parameters as one of the scale-up runs (e.g., JVM heapsize) and the same application load. This produces two entries for matrix A, where rows are again workloads and columns are scale-out allocations (numbers of servers). Collaborative filtering then recovers the missing entries of performance across all node counts. Scale-out classification requires additional servers for profiling. To avoid increasing the classification overheads when the system is online, applications are only profiled on one to four nodes for scale-out classification. To accurately estimate the performance of incoming workloads for larger node counts, in offline mode, we have exhaustively profiled a small number of different workload types (20-30) against node counts 1 to 100. These runs provide the classification engine with dense information on workload behavior for larger node counts. This step does not need to repeat unless there are major changes in the cluster's hardware or application structure.

Heterogeneity classification: This classification requires one more profiling run on a different and randomly-chosen server type using the same workload parameters and for the same duration as a scale-up run. Collaborative filtering estimates workload performance across all other server types.

Interference classification: This classification quantifies the sensitivity of the workload to interference caused and tolerated in various shared resources, including the CPU, cache hierarchy, memory capacity and bandwidth, and storage and network bandwidth. This classification does not require an extra profiling run. Instead, it leverages the first copy of the scale-up classification to inject, one at a time, two microbenchmarks that create contention in a specific shared resource [19]. Once the microbenchmark is injected, Quasar tunes up its intensity until the workload performance drops below an acceptable level of QoS (typically 5%). This point is recorded as the workload's sensitivity to this type of interference in a new row in the corresponding matrix A. The columns of the matrix are the different sources of interference. Classification is then applied to derive the sensitivities to the remaining sources of interference. Once the profiling runs are complete the different types of classification reconstruct the missing entries and provide recommendations on efficient allocations and assignments for each workload. Classification typically takes a few msec even for thousands of applications and servers.

Multiple parallel versus single exhaustive classification: Classification is decomposed to the four components previously described for both accuracy and efficiency reasons. The alternative design would consist of a single classification that examines all combinations of resource allocations and resource assignments at the same time. Each row in this case is an incoming workload, and each column is an allocation-assignment vector. Exhaustive classification addresses pathological cases that the four simpler classifications estimate poorly. For example, if TCP incast occurs for a specific allocation, *only* on a specific server platform that is not used for profiling, its performance impact will not be identified by classification. Although these cases are rare, they can result in unexpected performance results. On the other hand, the exponential increase in the column count in the exhaustive scheme increases the time required to perform classification [35, 50, 66] (note that this occurs at every application arrival). Moreover, because the number of columns now exceeds the number of rows, classification accuracy decreases, as SVD finds fewer similarities with high confidence [26, 49, 64].

To address this issue without resorting to exhaustive classification, we introduce a simple feedback loop that updates the matrix entries when the performance measured at runtime deviates from the one estimated through classification. This loop addresses such misclassifications, and additionally assists with scaling to server counts that exceed the capabilities of profiling, i.e., more than 100 nodes.

Validation: Table 2 summarizes a validation of the accuracy of the classification engine in Quasar. We use a 40-server cluster and applications from Hadoop (10 data-mining jobs), latency-critical services (10 memcached jobs, and 10 Apache webserver loads), and 413 single-node benchmarks from SPEC, PARSEC, SPLASH-2, BioParallel, Minebench and SpecJbb. The memcached and webserving jobs differ in their query distribution, input dataset and/or incoming load. Hadoop jobs additionally differ in terms of the application logic. Details on the applications and systems can be found in Section 5. We show average, 90^{th} percentile and maximum errors for each application and classification type. The errors show the deviation between estimated and measured performance or sensitivity to interference. On average, classification errors are less than 8% across all application types, while maximum errors are less than 17%, guaranteeing that the information that drives cluster management decisions is accurate. Table 2 also shows the corresponding errors for the exhaustive classification. In this case, average errors are slightly higher, especially for applications arriving early in the system [49], however, the deviation between average and maximum errors is now lower, as the exhaustive classification can accurately predict performance for the pathological cases that the four parallel classifications miss.

We also validate the selected number of profiling runs, i.e., how classification accuracy changes with the density of the input matrices. Figure 3(a-d) shows how the 90^{th} percentile of errors from classification changes as the density of the corresponding input matrix increases. For clarity, we omit the plots for Webserver, which has similar patterns to memcached. For all four classification types, a single profiling run per classification results in high errors. Two or more entries per input row result in decreased errors, although the benefits reach the point of diminishing returns after 4-5 entries. This behavior is consistent across application types, although the exact values of errors may differ. Unless otherwise specified, we use 2 entries per row in subsequent experiments. Figure 3e shows the overheads (profiling and classification) for the three application classes (Hadoop, memcached, single node) as input matrix density increases. We assume that the hardware resources used towards profiling and classification are kept constant. Obviously as the number of profiling runs increases the overheads increase significantly, without equally important accuracy improvements. The figure also shows the overheads from classification only (excluding profiling) for the four parallel classifications (*4p*) and the exhaustive scheme (*exh.*). As expected, the increase in column count corresponds in an increase in decision time, often by two orders of magnitude.

3.3 Greedy Allocation and Assignment

The classification output is given to a greedy scheduler that jointly determines the amount, type, and exact set of allocated resources. The scheduler's objective is to allocate the *least amount of resources needed to satisfy a workload's performance target*. This greatly reduces the space the scheduler traverses, allowing it to examine higher quality resources first, as smaller quantities of them will meet the performance constraint. This approach also scales well to many servers.

The scheduler uses the classification output, to first rank the available servers by decreasing resource quality, i.e., high performing platforms with minimal interference first. Next, it sizes the allocation based on available resources until the performance constraint is met. For example, if a webserver must meet a throughput of 100K QPS with 10msec 99^{th} percentile latency and the highest-ranked servers can achieve at most 20K QPS, the workload would need five servers to meet the constraints. If the number of highest-ranked servers available is not sufficient, the scheduler will also allocate lower-ranked servers and increase their number. The feedback between allocation and assignment ensures that the amount and quality of resources are accounted for jointly. When sizing the allocation, the algorithm first increases the per-node resources (scale-up) to better pack work in few servers, and then distributes the load across machines (scale-out). Nevertheless, alternative heuristics can be used based on the workload's locality properties or to address fault tolerance concerns.

The greedy algorithm has $O(M \cdot logM + S)$ complexity, where the first component accounts for the sorting overhead and the second for the examination of the top S servers, and in practice takes a few msec to determine an allocation/assignment even for systems with thousands of servers. Despite its greedy nature, we show in Section 6 that the decision quality is quite high, leading to both high workload performance and high resource utilization. This is primarily due to the accuracy of the information available after classification. A potential source of inefficiency is that the scheduler allocates resources on a per-application basis in the order workloads arrive. Suboptimal assignments can be detected by sampling a few workloads (e.g., based on job priorities if they are available) and adjusting their assignment later on as resources become available when other workloads terminate. Finally, the scheduler employs admission control to prevent oversubscription when no resources are available.

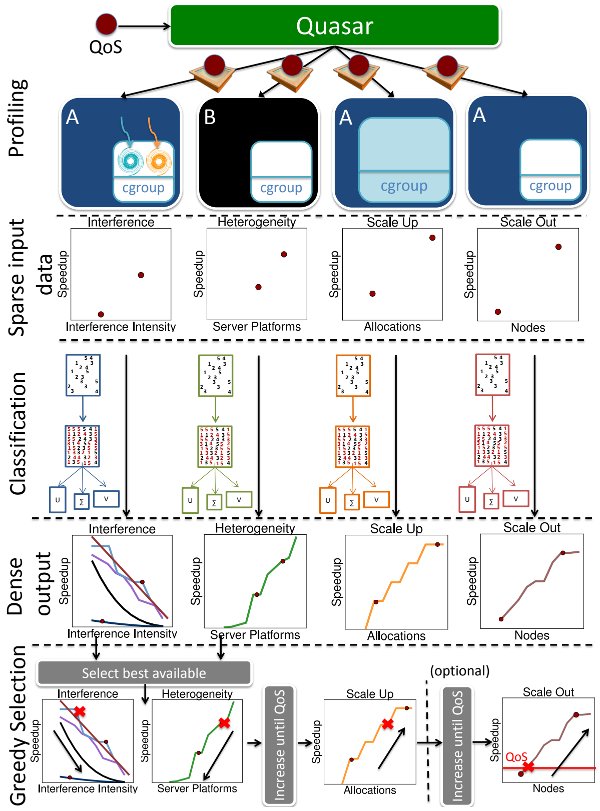

Figure 4: The steps for cluster management with Quasar. Starting from the top, short runs using sandboxed workload copies produce the initial profiling signal that classification techniques expand to information about relationship between performance and scale-up, scale-out, heterogeneity, and interference. Finally, the greedy scheduler uses the classification output to find the number and type of resources that maximize utilization and application performance.

3.4 Putting it All Together

Figure 4 shows the different steps of cluster management in Quasar. Upon arrival of a workload, Quasar collects profiling data for scale-out and scale-up allocations, heterogeneity, and interference. This requires up to four profiling runs that happen in parallel. All profiling copies are sandboxed (as explained in Section 4.2), the two platforms used are A and B (two nodes of A are used for the scale-out classification) and each profiling type produces two points in the corresponding speedup graph of the workload. The profiling runs happen with the actual dataset of the workload. The total profiling overhead depends on the workload type and is less than 5 min in all cases we examined. For non-stateful services, e.g., small batch workloads that are a large fraction of DC workloads [54], the complete profiling takes 10-15 seconds. Note that for stateful services, e.g., Cassandra, where setup is necessary, it only affects *one* of the profiling runs. Once the service is warmed-up, subsequent profiling only requires a few seconds to complete. Once the profiling results are available, classification provides the full workload characterization (speedup graph). Next, the greedy scheduler assigns specific servers to the workload. Overall, Quasar's overheads are quite low even for short-running applications (batch, analytics) or long running online services.

Quasar maintains per-workload and per-server state. Per-workload state includes the classification output. For a cluster with 10 server types and 10 sources of interference, we need roughly 256 bytes per workload. The per-server state includes information on scheduled applications and their cumulative resource interference, roughly 128B in total. The per-server state is updated on each workload assignment. Quasar also needs some storage for the intermediate classification results and for server ranking during assignment. Overall, state overheads are marginal and scale linearly with the number of workloads and servers. In our experiments, a single server was sufficient to handle the total state and computation of cluster management. Additional servers can be used for fault-tolerance.

4. Implementation

We implemented a prototype for Quasar in about 6KLOC of C, C++, and Python. It runs on Linux and OS X and currently supports applications written in C/C++, Java, and Python. The API includes functions to express the performance constraints and type of submitted workloads, and functions to check job status, revoke it, or update the constraints. We have used Quasar to manage analytics frameworks such as Hadoop, Storm, and Spark, latency-critical services such as NoSQL workloads, and conventional single-node workloads. There was no need to change any applications or frameworks. The framework-specific code in Quasar is 100-600 LOC per framework. In the future, we plan to merge the Quasar classification and scheduling algorithms in a cluster management framework like OpenStack or Mesos.

4.1 Dynamic Adaptation

Some workloads change behavior during their runtime, either due to phase changes or due to variation in user traffic. Quasar detects such changes and adjusts resource allocation and/or assignment to preserve the performance constraints.

Phase detection: Quasar continuously monitors the performance of all active workloads in the cluster. If a workload runs below its performance constraint, it either went through a phase change or was incorrectly classified or assigned. In any case, Quasar reclassifies the application at its current state and adjusts its resources as needed (see discussion below). We also proactively test for phase changes and misclassifications/misscheduling by periodically sampling a few active workloads and injecting interfering microbenchmarks to them. This enables partial interference classification in place. If there is a significant change compared to the original classification results, Quasar signals a phase change.

Proactive detection is particularly useful for long-running workloads that may affect co-located workloads when entering a phase change. We have validated the phase detection schemes with workloads from SPECCPU2006, PARSEC, Hadoop and memcached. With the reactive-only scheme, Quasar detects 94% of phase changes. By sampling 20% of active workloads every 10 minutes, we detect 78% of changes proactively with 8% probability of false positives.

Allocation adjustment: Once the phase has been detected or load increases significantly for a user-facing workload, Quasar changes the allocation to provide more resources or reclaim unused resources. Quasar adjusts allocations in a conservative manner. It first scales up or down the resources given to the workload in each of the servers it currently occupies. If needed, best-effort (low priority) workloads are evicted from these servers. If possible, a scale-up adjustment is the simplest option as it typically requires no state migration. If scale-up is not possible or cannot address the performance needs, scale-out and/or migration to other servers is used. For stateless services (e.g., adding/removing workers to Hadoop or scaling a webserver), scale-out is straightforward. For stateful workloads, migration and scale-out can be expensive. If the application is organized in microshards [16], Quasar will migrate a fraction of the load from each server to add capacity at minimum overhead. At the moment, Quasar does not employ load prediction for user-facing services [29, 46]. In future work, we will use such predictors as an additional signal to trigger adjustments for user-facing workloads.

4.2 Side Effect Free Profiling

To acquire the profiling data needed for classification, we must launch multiple copies of the incoming application. This may cause inconsistencies with intermediate results, duplicate entries in databases, or data corruption on file systems. To eliminate such issues, Quasar uses sandboxing for the training copies during profiling. We use *Linux containers* [8] with `chroot` to sandbox profiling runs and create a copy-on-write filesystem snapshot so that files (including framework libraries) can be read and written as usual [72]. Containers enable full control over how training runs interact with the rest of the system, including limiting resource usage through *cgroups*. Using virtual machines (VMs) for the same purpose is also possible [47, 62, 63, 68], but we chose containers as they incur lower overheads for launching.

4.3 Stragglers

In frameworks like Hadoop or Spark, individual tasks may take much longer to complete for reasons that range from poor work partitioning to network interference and machine instability [4]. These straggling tasks are typically identified and relaunched by the framework to ensure timely job completion [1, 3, 4, 17, 23, 37, 70]. We improve straggler detection in Hadoop in the following manner. Quasar calls the `TaskTracker` API in Hadoop and checks for underperforming tasks (at least 50% slower than the median). For such tasks, Quasar injects two contentious microbenchmarks in the corresponding servers and reclassifies them with respect to interference caused and tolerated. If the results of the in-place classification differ from the original by more than 20%, we signal the task as a straggler and notify the Hadoop JobTracker to relaunch it on a newly assigned server. This allows Quasar to detect stragglers 19% earlier than Hadoop, and 8% earlier than LATE [70] for the Hadoop applications described in the first scenario in Section 5.

4.4 Discussion

Cost target: Apart from a performance target, a user could also specify a cost constraint, priorities, and utility functions for a workload [55]. These can either serve as a limit for resource allocation or to prioritize allocations during very high load. We will consider these issues in future work.

Resource partitioning: Quasar does not explicitly partition hardware resources. Instead, it reduces interference by co-locating workloads that do not contend on the shared resources. Resource partitioning is orthogonal. If mechanisms like cache partitioning or rate limiting at the NIC are used, interference can be reduced and more workload colocations will be possible using Quasar. In that case, Quasar will have to determine the settings for partitioning mechanisms, in the same way it determines the number of cores to use for each workload. We will consider these issues in future work.

Fault tolerance: We use master-slave mirroring to provide fault-tolerance for the server that runs the Quasar scheduler. All system state (list of active applications, allocations, QoS guarantees) is continuously replicated and can be used by hot-standby masters. Quasar can also leverage frameworks like ZooKeeper [5] for more scalable schemes with multiple active schedulers. Quasar does not explicitly add to the fault tolerance of frameworks like MapReduce. In the event of a failure, the cluster manager relies on the individual frameworks to recover missing worker data. Our current resource assignment does not account for fault zones. However, this is a straight forward extension for the greedy algorithm.

5. Methodology

Clusters: We evaluated Quasar on a 40-server local cluster and a 200-server cluster on EC2. The ten platforms of the local cluster range from dual core Atom boards to dual socket 24 core Xeon servers with 48GB of RAM. The EC2 cluster has 14 server types ranging from small to x-large instances. All servers are dedicated and managed only by Quasar, i.e., there is no interference from external workloads.

The following paragraphs summarize the workload scenarios used to evaluate Quasar. Scenarios include batch and latency-critical workloads and progressively evaluate different aspects of allocation and assignment. Unless otherwise specified experiments are run 7 times for consistency and we report the average and standard deviation.

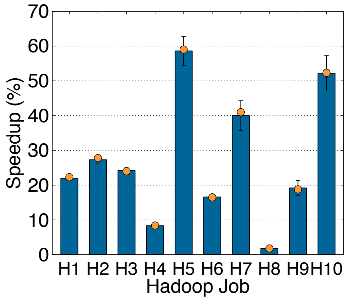

Figure 5: Performance of the ten Hadoop jobs.

Parameter	Quasar	Hadoop
Block size	64MB	64MB
Compression rate	7.6(gzip)	5.1(lzo)
Heapsize	0.75GB	1GB
Replication	2	2
Mappers per node	12	8
Server type	E-F	A-E

Table 3: Parameters setting for Hadoop job H8 by Quasar and the Hadoop scheduler.

Single Batch Job: Analytics frameworks like Hadoop [30], Storm [57], and Spark [71] are large consumers of resources on private and public clouds. Such frameworks have individual schedulers that set the various framework parameters (e.g., mappers per node and block size) and determine resource allocation (number of servers used). The allocations made by each scheduler are suboptimal for two reasons. First, the scheduler does not have full understanding of the complexity of the submitted job and dataset. Second, the scheduler is not aware of the details of available servers (e.g., heterogeneity), resulting in undersized or overprovisioned allocations. In this first scenario, a single Hadoop job is running at a time on the small cluster. This simple scenario allows us to compare the resource allocation selected by Hadoop to the allocation/assignment of Quasar on a single job basis. We use ten Hadoop jobs from the Mahout library [39] that represent data mining and machine learning analyses. The input datasets vary between 1 and 900GB. Note that there is no workload co-location in this scenario.

Multiple Batch Jobs: The second scenario represents a realistic setup for batch processing clusters. The cluster is shared between jobs from multiple analytics frameworks (Hadoop, Storm, and Spark). We use 16 Hadoop applications running on top of the Mahout library, four workloads for real-time text and image processing in Storm, and four workloads for logical regression, text processing and machine learning in Spark. These jobs arrive in the cluster with 5 sec inter-arrival times. Apart from the analytics jobs, a number of single-server jobs are submitted to the cluster. We use workloads from SPECCPU2006, PARSEC [12], SPLASH-2 [67], BioParallel [34], Minebench [43] and 350 multiprogrammed 4-app mixes from SPEC [53]. These single-server workloads arrive with 1 second inter-arrival times and are treated as best-effort (low priority) load that fills any cluster capacity unused by analytics jobs. There are not guarantees on performance of best-effort tasks, which may be migrated or killed at any point to provide resources for analytics tasks.

We compare Quasar to allocations done by the frameworks themselves (Hadoop, Spark, Storm schedulers) and assignments by a least-loaded scheduler that accounts for core and memory use but not heterogeneity or interference.

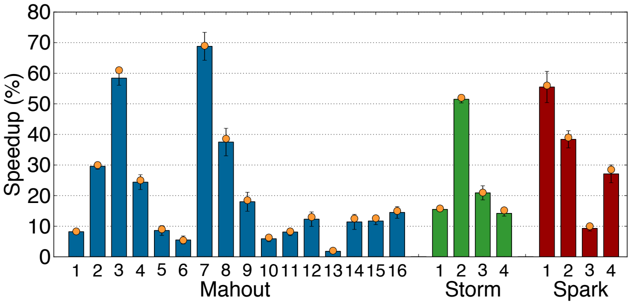

Figure 6: Performance speedup for the Hadoop, Storm and Spark jobs with Quasar.

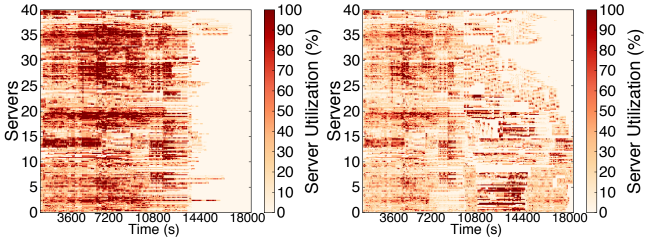

Figure 7: Cluster utilization with Quasar (left) and the framework schedulers (right).

Low-Latency Service: Latency-critical services are also major tenants in cloud facilities. We constructed a webserving scenario using the HotCRP conference management system [33], which includes the Apache webserver, application logic in PHP, and data stored in MySQL. The front- and back-end run on the same machine, and the installation is replicated across several machines. The database is kept purposefully small (5GB) so that it is cached in memory and emphasis is placed on compute, cache, memory and networking issues, and not on disk performance. HotCRP traffic includes requests to fill in paper abstracts, update author information, and upload or read papers. Apart from throughput constraints, HotCRP requires a 100msec per-request latency.

We use three traffic scenarios: flat, fluctuating, and large spike. Apart from satisfying HotCRP constraints, we want to use any remaining cluster capacity for single-node, best-effort tasks (see description in previous scenario). We compare Quasar to a system that uses an auto-scaling approach to scale HotCRP between 1 and 8 servers based on the observed load of the servers used [6]. Auto-scale allocates an additional, least-loaded server for HotCRP when current load exceeds 70% [7] and redirects a fair share of the traffic to the new server instance. Load balancing happens on the workload generator side. Best-effort jobs are assigned by a least-loaded (LL) scheduler. Quasar deals with load changes in HotCRP by either scaling-up existing allocations or scaling-out (more servers) based on how the two affect performance.

Stateful Latency-Critical Services: This scenario extends the one above in two ways. First, there are multiple low-latency services. Second, these services involve significant volumes of state. Specifically, we examine the deployment

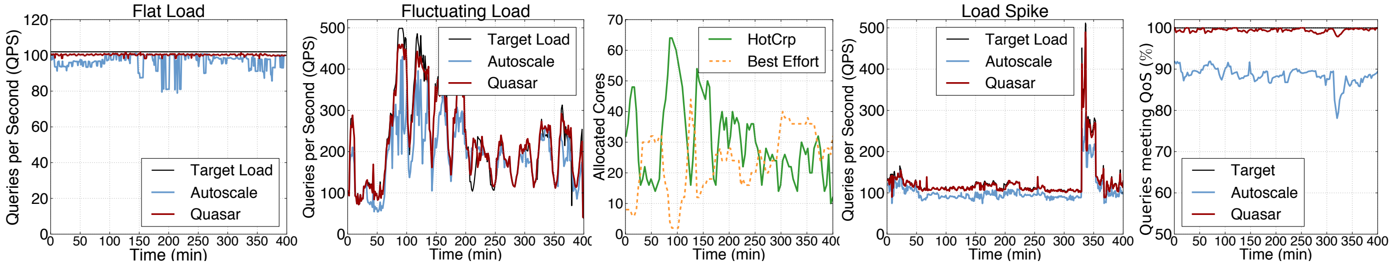

Figure 8: Throughput for HotCRP under (a) flat, (b) fluctuating and (d) spiking load. Fig. 8 (c) shows the core allocation in Quasar for the fluctuating load, and (e) shows the fraction of queries meeting the latency constraint for the load with spike.

of memory-based memcached [22] and disk-based Cassandra [14], two latency-critical NoSQL services. Memcached (1TB state) is presented with load that fluctuates following a diurnal pattern with maximum aggregate throughput target of 2.4M QPS and a 200usec latency constraint. The disk-bound Cassandra (4TB state) has a lower load of 60K QPS of maximum aggregate throughput and a 30 msec latency constraint. Any cluster capacity unused by the two services is utilized for best-effort workloads which are submitted with 10sec inter-arrival times. To show the fluctuation of utilization with load, and since scaling now involves state migration, this scenario runs over 24 hours and is repeated 3 times for consistency. Similarly to the previous scenario, we compare Quasar with the auto-scaling approach and measure performance (throughput and latency) for the two services and overall resource utilization. Scale-out in this case involves migrating one (64MB) or more microshards to a new instance, which typically takes a few msec.

Large-Scale Cloud Provider: Finally, we bring everything together in a general case where 1200 workloads of all types (analytics batch, latency-critical, and single-server jobs) are submitted in random order to a 200-node cluster of dedicated EC2 servers with 1 sec inter-arrival time. All applications have the same priority and no workload is considered best-effort (i.e., all paying customers have equal importance). The scenario is designed to use almost all system cores at steady-state, without causing oversubscription, under ideal resource allocation. We do, however, employ admission control to prevent machine oversubscription, when allocation is imperfect [18]. Wait time due to admission control counts towards scheduling overheads. Quasar handles allocation and assignment for all workloads. For comparison, we use an auto-scale approach for resource allocation of latency-critical workloads. For frameworks like Hadoop and Storm, the framework estimates its resource needs and we treat that as a reservation. For resource assignment, we use two schedulers: a least-loaded scheduler that simply accounts for core and memory availability and Paragon that, given a resource allocation, can do heterogeneity- and interference-aware assignment. The latter allows us to demonstrate the benefits of jointly solving allocation and assignment over separate (although optimized) treatment of the two.

6. Evaluation

6.1 Single Batch Job

Performance: Fig. 5 shows the reduction in execution time of ten Hadoop jobs when resources are allocated by Quasar instead of Hadoop itself. We account for all overheads, including classification and scheduling. Quasar improves performance for all jobs by an average of 29% and up to 58%. This is significant given that these Hadoop jobs take two to twenty hours to complete. The yellow dots show the execution time improvement needed to meet the performance target the job specified at submission. Targets are set to the best performance achieved after a parameter sweep on the different server platforms. Quasar achieves performance within 5.8% of the constraint on average, leveraging the information of how resource allocation and assignment impact performance. When resources are allocated by Hadoop, performance deviates from the target by 23% on average.

Efficiency: Table 3 shows the different parameter settings selected by Quasar and by Hadoop for the H8 Hadoop job, a recommendation system that uses Mahout with a 20GB dataset [39]. Apart from the block size and replication factor, the two frameworks set job parameters differently. Quasar detects that interference between mappers is low and increases the mappers per node to 12. Similarly, it detects that heap size is not critical for this job and reduces its size, freeing resources for other workloads. Moreover, Quasar allocates tasks to the two most suitable server types (E and F), while Hadoop chooses from all available server types.

6.2 Multiple Batch Frameworks

Performance: Fig. 6 shows the reduction in execution times for Hadoop, Storm, and Spark jobs when Quasar manages resource allocation and assignment. On average, performance improves by 27% and comes within 5.3% of the provided constraint, a significant improvement over the baseline. Apart from sizing and configuring jobs better, Quasar can aggressively co-locate them. For example, it can detect when two memory-intensive Storm and Spark jobs interfere and when they can efficiently share a system. Quasar allows the remaining cluster capacity to be used for best-effort jobs without disturbing the primary jobs because it is interference-aware. Best-effort jobs come within 7.8% on

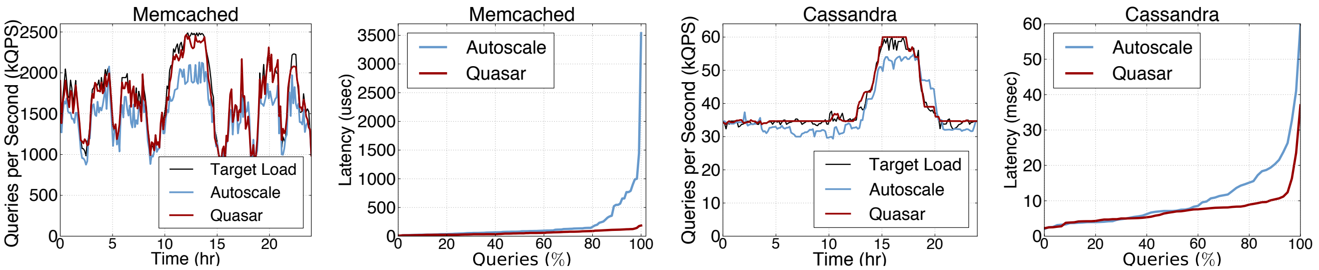

Figure 9: Throughput and latency for memcached and Cassandra in a cluster managed by Quasar or an auto-scaling system.

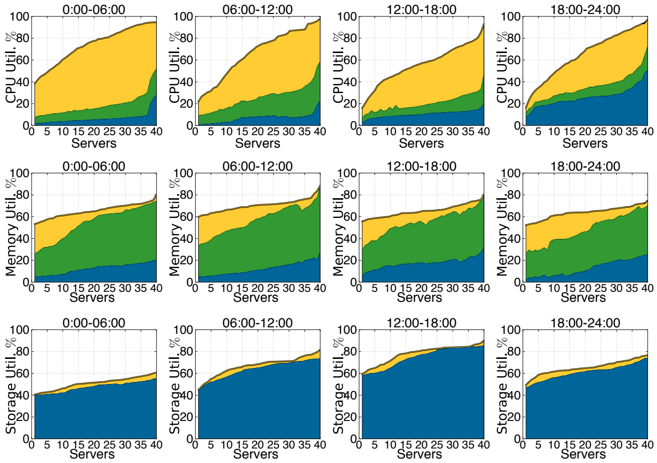

Figure 10: Average resource usage across all servers for four 6-hour snapshots. The cluster is running memcached (green), Cassandra (blue), and best-effort tasks (yellow) and is managed by Quasar.

average of the peak performance each job could achieve if it was running alone on the highest performing server type.

Utilization: Fig. 7 shows the per-server CPU utilization (average across all cores) over time in the form of a heatmap. Utilization is sampled every 5 sec. In addition to improving individual job performance, Quasar increases utilization, achieving 62% on average versus 34% with the individual framework schedulers (right heatmap). Because performance is now higher the whole experiment completes faster. Workloads after $t = 14400$ are mostly best-effort jobs that take longer than the main analytics workloads to complete.

6.3 Low-Latency Service

Performance: Fig. 8a shows the aggregate throughput for HotCRP achieved with Quasar and the auto-scaling system when the input traffic is flat. While the absolute differences are small, it is important to note that the auto-scaling manager causes frequent QPS drops due to interference from best-effort workloads using idling resources. With Quasar, HotCRP runs undisturbed and the best-effort jobs achieve runtimes within 5% of minimum, while with auto-scale, they achieve runtimes within 24% of minimum. When traffic varies (Fig. 8b), Quasar tracks target QPS closely, while autoscale provides 18% lower QPS on average, both due to interference and suboptimal scale-up configuration. Quasar's smooth behavior is due to the use of both scale-out and scale-up to best meet the new QPS target, leaving the highest number of cores possible for best-effort jobs (Fig. 8c). For the load with the sharp spike, Quasar tracks QPS within 4% on average (Fig. 8d) and meets the latency QoS for nearly all requests (Fig. 8e). When the spike arrives, Quasar first scales up each existing allocation, and then only uses two extra servers of suitable type to handle remaining traffic. The autoscaling system observes the load increase when the spike arrives and allocates four more servers. Due to the higher latency of scale-out and the fact that auto-scaling is not aware of heterogeneity or interference, it fails to meet the latency guarantees for over 20% of requests around the spike arrival.

6.4 Stateful Latency-Critical Services

Performance: Fig. 9 shows the throughput of memcached and Cassandra over time and the distribution of query latencies. Quasar tracks throughput targets closely for both services, while the auto-scaling manager degrades throughput by 24% and 12% on average for memcached and Cassandra respectively. The differences in latency are larger between the two managers. Quasar meets latency QoS for memcached for 98.8% of requests, while auto-scaling only for 80% of requests. For Cassandra, Quasar meets the latency QoS for 98.6% of requests, while the auto-scaling for 93% of requests. Memcached is memory-based and has an aggressive latency QoS, making it more sensitive to suboptimal resource allocation and assignment on a shared cluster.

Utilization: Fig. 10 shows the utilization of CPU, memory capacity, and disk bandwidth across the cluster servers when managed by Quasar over 24h. Each column is a snapshot of average utilization over 6 hours. Since memcached and Cassandra have low CPU requirements, excluding the period 18:00-24:00 when Cassandra performs garbage collection, most of the CPU capacity is allocated to best-effort jobs. The number of best-effort jobs varies over time because the exact load of memcached and Cassandra changes. Most memory is used to satisfy the requirements of memcached, with small amounts needed for Cassandra and best-effort jobs. Cassandra is the nearly exclusive user of disk I/O. Some servers do not exceed 40-50% utilization for most of the experiment's

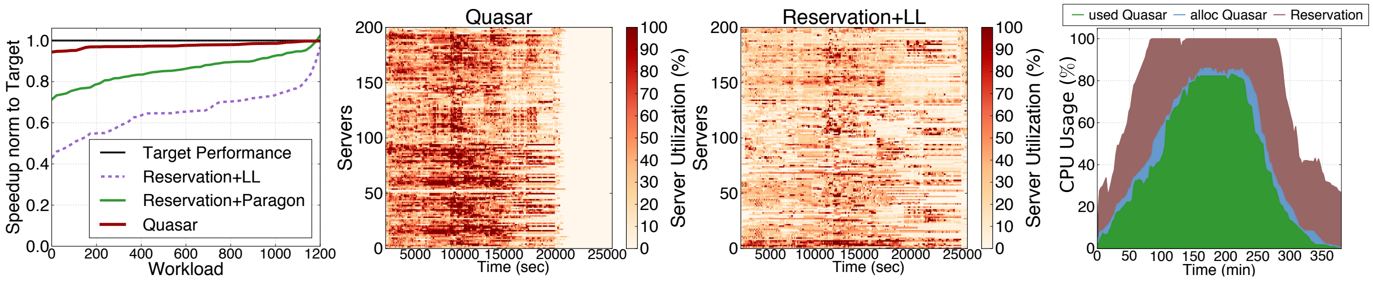

Figure 11: (a) Performance and cluster utilization for 1200 workloads on 200 EC2 servers with (b) Quasar and (c) the reservation+LL system. Fig. 11(d) shows the allocated versus used resources for Quasar and allocated for reservation+LL.

duration. These are low-end machines, for which higher utilization dramatically increases the probability of violating QoS constraints for latency-critical services. In general, the cluster utilization is significantly higher than if each service was running in dedicated machines.

6.5 Large-Scale Cloud Provider

Performance: Figure 11 presents the overall evaluation of Quasar managing a 200-node cluster running all previously-discussed types of workloads. We compare to resource allocation based on reservations (e.g., expressed by the Hadoop scheduler or an auto-scaling system) and resource assignment on least-loaded machines (LL) or based on the interference and heterogeneity-aware Paragon. Figure 11a shows the performance of the 1,200 workloads ordered from worst-to best-performing, normalized to their performance target. Quasar achieves 98% of the target on average, while the reservation-based system with Paragon achieves 83%. This shows the need to perform allocation and assignment together; the intelligent resource assignment by Paragon is not sufficient. Using reservations and LL assignment performs quite poorly, only achieving 62% of the target on average.

Utilization: Figures 11b-c show the per-server CPU utilization throughout the scenario's execution for Quasar and the reservation+LL system. Average utilization is 62% with Quasar, while meeting performance constraints for both batch and latency-critical workloads. The reservation+LL manager achieves average utilization of 15%, 47% lower than Quasar. Figure 11d shows the allocated and used resources for Quasar compared to the resources reserved by the reservation+LL manager over time. Overprovisioning with Quasar is low, with the difference between allocated and used being roughly 10%. This is significantly lower than the resources reserved by the reservation-based manager, which exceed the capacity of the cluster during most of the scenario. Because Quasar has detailed information on how different allocations/assignments affect performance, it can rightsize the allocations more aggressively, while meeting the performance constraints without QoS violations.

Cluster management overheads: For most applications the overheads of Quasar from profiling, classification, greedy selection and adaptation are low, 4.1% of execution time on average. For short-lived batch workloads, overheads are up to 9%. The overheads are negligible for any long-running service, and even for jobs lasting a few seconds, they only induce single-digit increases in execution time. In contrast with reservation+LL, Quasar does not introduce any wait time overheads due to oversubscription.

7. Conclusions

We have presented Quasar, a cluster management system that performs coordinated resource allocation and assignment. Quasar moves away from the reservation-based standard for cluster management. Instead of users requesting raw resources, they specify a performance target the application should meet and let the manager size resource allocations appropriately. Quasar leverages robust classification techniques to quickly analyze the impact of resource allocation (scale-up and scale-out), resource type (heterogeneity), and interference on performance. A greedy algorithm uses this information to allocate the least amount of resources necessary to meet performance constraints. Quasar currently supports distributed analytics frameworks, web-serving applications, NoSQL datastores, and single-node batch workloads. We evaluated Quasar over a variety of workload scenarios and compared it to reservation/auto-scaling-based resource allocation systems and schedulers that use similar classification techniques for resource assignment (but not resource allocation). We showed that Quasar improves aggregate cluster utilization and individual application performance.

Acknowledgements

We sincerely thank Rob Benson, Jonathan Blandford, Chris Lambert, Brian Wickman, Ben Hindman and the entire Mesos and Aurora teams at Twitter for allowing us to include data from their production system and for helping with the collection of the measurements. We would also like to thank John Ousterhout, Mendel Rosenblum, Daniel Sanchez, Jacob Leverich, David Lo, and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was partially supported by a Google directed research grant on energy proportional computing. Christina Delimitrou was supported by a Stanford Graduate Fellowship.

References

[1] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. London, UK, 2012.

[2] Amazon ec2. `http://aws.amazon.com/ec2/`.

[3] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Lombard, IL, 2013.

[4] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proc. of the 9th USENIX conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, CA, 2010.

[5] Apache zookeeper. `http://zookeeper.apache.org/`.

[6] Autoscale. `https://cwiki.apache.org/CLOUDSTACK/autoscaling.html`.

[7] AWS Autoscaling. `http://aws.amazon.com/autoscaling/`.

[8] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. OSDI, 1999.

[9] L. Barroso. Warehouse-scale computing: Entering the teenage decade. *ISCA Keynote, SJ, June 2011*.

[10] Luiz Barroso and Urs Hoelzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[11] R. Bell, M. Koren, and C. Volinsky. The BellKor 2008 Solution to the Netflix Prize. Technical report, AT&T Labs, 2008.

[12] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Toronto, CA, October, 2008.

[13] Leon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proc. of the International Conference on Computational Statistics (COMPSTAT)*. Paris, France, 2010.

[14] Apache cassandra. `http://cassandra.apache.org/`.

[15] McKinsey & Company. Revolutionizing data center efficiency. In *Uptime Institute Symposium, 2008*.

[16] Jeffrey Dean and Luiz Andre Barroso. The tail at scale. In *Communications of the ACM, Vol. 56 No. 2, Pages 74-80*.

[17] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of OSDI*, pages 10–10, 2004.

[18] Christina Delimitrou, Nick Bambos, and Christos Kozyrakis. QoS-Aware Admission Control in Heterogeneous Datacenters. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*. San Jose, June 2013.

[19] Christina Delimitrou and Christos Kozyrakis. iBench: Quantifying Interference for Datacenter Workloads. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC)*. Portland, OR, September 2013.

[20] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proc. of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, TX, USA, 2013.

[21] Eucalyptus cloud services. `http://www.eucalyptus.com/`.

[22] Brad Fitzpatrick. Distributed caching with memcached. In *Linux Journal, Volume 2004, Issue 124, 2004*.

[23] R. Gandhi and A. Sabne. Finding stragglers in hadoop. In *Tech. Report*. 2011.

[24] Gartner says efficient data center design can lead to 300 percent capacity growth in 60 percent less space. `http://www.gartner.com/newsroom/id/1472714`.

[25] Google compute engine. `http://cloud.google.com/products/compute-engine.html`.

[26] Z. Ghahramani and M. Jordan. Learning from incomplete data. In *Lab Memo No. 1509, CBCL Paper No. 108, MIT AI Lab*.

[27] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proc. of the 8th USENIX conference on Networked systems design and implementation (NSDI)*. Boston, MA, 2011.

[28] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Workload analysis and demand prediction of enterprise data center applications. In *Proc. of the 10th IEEE International Symposium on Workload Characterization*. Boston, 2007.

[29] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Proc. of the International Conference on Network and Service Management (CNSM)*. Niagara Falls, ON, 2010.

[30] Apache hadoop. `http://hadoop.apache.org/`.

[31] J Hamilton. Cost of power in large-scale data centers. `http://perspectives.mvdirona.com`.

[32] Ben 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 *Proc. of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, MA, 2011.

[33] Hotcrp conference management system. `http://read.seas.harvard.edu/~kohler/hotcrp/`.

[34] Aamer Jaleel, Matthew Mattina, and Bruce L. Jacob. Last level cache (llc) performance of data mining workloads on a cmp - a case study of parallel bioinformatics workloads. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*. Austin, Texas, 2006.

[35] K.C. Kiwiel. Convergence and efficiency of subgradient methods for quasiconvex minimization. In *Mathematical Programming (Series A) (Berlin, Heidelberg: Springer) 90 (1): pp. 1-25, 2001.*

[36] Jacob Leverich and Christos Kozyrakis. On the energy (in)efficiency of hadoop clusters. In *Proc. of HotPower*. Big Sky, MT, 2009.

[37] J. Lin. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce. In *Proc. of LSDS-IR Workshop*. Boston, MA, 2009.

[38] Host server cpu utilization in amazon ec2 cloud. `http://huanliu.wordpress.com/2012/02/17/host-server-cpu-utilization-in-amazon-\ec2-cloud/`.

[39] Mahout. `http://mahout.apache.org/`.

[40] Jason Mars and Lingjia Tang. Whare-map: heterogeneity in "homogeneous" warehouse-scale computers. In *Proc. of the 40th Annual International Symposium on Computer Architecture (ISCA)*. Tel-Aviv, Israel, 2013.

[41] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proc. of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, 2011.

[42] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 319–330, 2011.

[43] Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Gokhan Memik, and Alok N. Choudhary. Minebench: A benchmark suite for data mining workloads. In *Proceedings of the 9th IEEE International Symposium on Workload Characterization (IISWC)*. San Jose, California, 2006.

[44] R. Nathuji, C. Isci, and E. Gorbatov. Exploiting platform heterogeneity for power efficient data centers. In *Proc. of ICAC'07, FL, 2007*.

[45] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proc. of EuroSys France, 2010*, 2010.

[46] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proc. of the USENIX International Conference on Automated Computing (ICAC'13)*. San Jose, CA, 2013.

[47] Dejan Novakovic, Nedeljko Vasic, Novakovic, Stanko, Dejan Kostic, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proc. of the USENIX Annual Technical Conference (ATC'13)*. San Jose, CA, 2013.

[48] Openstack cloud software. `http://www.openstack.org/`.

[49] Nathan Parrish, Hyrum Anderson, Maya Gupta, and Dun Yu Hsaio. Classifying with confidence from incomplete information. In *Proc. of the Journal Machine Learning Research (JMLR)*. 2013.

[50] A. Rajaraman and J. Ullman. *Textbook on Mining of Massive Datasets*. 2011.

[51] Charles Reiss, Alexey Tumanov, Gregory Ganger, Randy Katz, and Michael Kozych. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. of the Third ACM Symposium on Cloud Computing (SOCC)*. San Jose, CA, 2012.

[52] Rightscale. `https://aws.amazon.com/solution-providers/isv/rightscale`.

[53] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proc. of the 38th annual International Symposium in Computer Architecture (ISCA-38)*. San Jose, CA, June, 2011.

[54] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proc. of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, April 2013.

[55] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. A cost-aware elasticity provisioning system for the cloud. In *Proc. of the 2011 31st International Conference on Distributed Computing Systems (ICDCS)*. Minneapolis, MN, 2011.

[56] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proc. of the 2nd ACM Symposium on Cloud Computing (SOCC)*. Cascais, Portugal, 2011.

[57] Storm. `https://github.com/nathanmarz/storm/`.

[58] Torque resource manager. `http://www.adaptivecomputing.com/products/open-source/torque/`.

[59] Arunchandar Vasan, Anand Sivasubramaniam, Vikrant Shimpi, T. Sivabalan, and Rajesh Subbiah. Worth their watts? an empirical study of datacenter servers. In *Proc. of the 16th International Symposium on High Performance Computer Architecture (HPCA)*. Bangalore, India, 2010.

[60] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. Dejavu: accelerating resource allocation in virtualized environments. In *Proc. of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. London, UK, 2012.

[61] Vmware vcloud suite. `http://www.vmware.com/products/vcloud`.

[62] Virtualbox. `https://www.virtualbox.org/`.

[63] Vmware virtual machines. `http://www.vmware.com/`.

[64] C. Wang, X. Liao, L. Carin, and D. B. Dunson. Classification with incomplete data using dirichlet process priors. In *Journal of Machine Learning Research (JMLR), 2010*.

[65] Windows azure. `http://www.windowsazure.com/`.

[66] Ian H. Witten, Eibe Frank, and Geoffrey Holmes. *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd Edition.

[67] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *Proc. of the 22nd International Symposium on Computer Architecture (ISCA)*. Santa Margherita Ligure, Italy, 1995.

[68] The xen project. http://www.xen.org/.

[69] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: precise online qos management for increased utilization in warehouse scale computers. In *Proc. of the 40th Annual International Symposium on Computer Architecture (ISCA)*. Tel-Aviv, Israel, 2013.

[70] M Zaharia, A Konwinski, A.D Joseph, R Katz, and I Stoica. Improving mapreduce performance in heterogeneous environments. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, CA, 2008.

[71] Matei Zaharia, M Chowdhury, T Das, A Dave, J Ma, M McCauley, M.J Franklin, S Shenker, and I Stoica. Spark: Cluster computing with working sets. In *Proc. of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. San Jose, CA, 2012.

[72] Zfs. http://www.freebsd.org/doc/handbook/filesystems-zfs.html.

[73] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proc. of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 2013.