Tarcil: reconciling scheduling speed and quality in large shared clusters

Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters

Christina Delimitrou†, Daniel Sanchez* and Christos Kozyrakis†

†Stanford University, *MIT

cdel@stanford.edu, sanchez@csail.mit.edu, kozyraki@stanford.edu

Abstract

Scheduling diverse applications in large, shared clusters is particularly challenging. Recent research on cluster scheduling focuses either on scheduling speed, using sampling to quickly assign resources to tasks, or on scheduling quality, using centralized algorithms that search for the resources that improve both task performance and cluster utilization.

We present Tarcil, a distributed scheduler that targets both scheduling speed and quality. Tarcil uses an analytically derived sampling framework that adjusts the sample size based on load, and provides statistical guarantees on the quality of allocated resources. It also implements admission control when sampling is unlikely to find suitable resources. This makes it appropriate for large, shared clusters hosting short- and long-running jobs. We evaluate Tarcil on clusters with hundreds of servers on EC2. For highly-loaded clusters running short jobs, Tarcil improves task execution time by 41% over a distributed, sampling-based scheduler. For more general scenarios, Tarcil achieves near-optimal performance for 4× and 2× more jobs than sampling-based and centralized schedulers respectively.

Categories and Subject Descriptors: D.4.1 [*Process Management*]: Scheduling

Keywords: Cloud computing, datacenters, scheduling, QoS, resource-efficiency, scalability

1. Introduction

An increasing and diverse set of applications is now hosted in private and public datacenters [4, 17, 24]. The large size of these clusters (tens of thousands of servers) and the high arrival rate of jobs (up to millions of tasks per second) make scheduling quite challenging. The cluster scheduler must determine which hardware resources, e.g., specific servers and cores, should be used by each job. Ideally, schedulers should have three desirable properties. First, each workload should receive the resources that enable it to achieve *predictable, high performance*. Second, jobs should be tightly packed on available servers to achieve *high cluster utilization*. Third, scheduling overheads should be minimal to allow the scheduler to *scale to large clusters and high job arrival rates*.

Recent research on cluster scheduling can be examined along two dimensions: *scheduling concurrency (throughput)* and *scheduling speed (latency)*.

With respect to scheduling concurrency, there are two groups of work. In the first, scheduling is serialized, with a centralized scheduler making all decisions [13, 19]. In the second, decisions are parallelized through two-level or distributed designs. Two-level schedulers, such as Mesos and YARN, use a centralized coordinator to divide resources between frameworks like Hadoop and MPI [18, 35]. Each framework uses its own scheduler to assign resources to tasks. Since neither the coordinator nor the framework schedulers have a complete view of the cluster state and all task characteristics, scheduling is suboptimal [31]. Shared-state schedulers like Omega [31] allow multiple schedulers to concurrently access the whole cluster state using atomic transactions. Finally, Sparrow uses multiple concurrent, stateless schedulers to sample and allocate resources [28].

With respect to the speed at which scheduling decisions happen, there are again two groups of work. The first group examines most of (or all) the cluster state to determine the most suitable resources for incoming tasks, in a way that addresses the performance impact of *hardware heterogeneity* and *interference in shared resources* [12, 16, 23, 26, 33, 40, 43]. For instance, Quasar [13] uses classification to determine the resource preferences of incoming jobs. Then, it uses a greedy scheduler to search the cluster state for resources that meet the application's demands on servers with minimal contention. Similarly, Quincy [19] formulates scheduling as a cost optimization problem that accounts for job preferences with respect to locality, fairness and starvation-freedom. These schedulers make high quality decisions that lead to high application performance and cluster utilization. However, they inspect the full cluster state on every scheduling event. Their decision overhead can be prohibitively high for large clusters, especially for very short

SoCC '15, August 27-29, 2015, Kohala Coast, HI, USA.

ACM 978-1-4503-3651-2/15/08...$15.00.

DOI: http://dx.doi.org/10.1145/2806777.2806779

jobs like real-time analytics (100 ms–10 s) [28, 41]. Using multiple greedy schedulers improves scheduling throughput but not latency, and terminating the greedy search early hurts decision quality, especially at high cluster loads.

The second group improves the speed of each scheduling decision by only examining a small number of machines. Sparrow reduces scheduling latency through resource sampling [28]. The scheduler examines the state of two randomly-selected servers for each required core and selects the one that becomes available first. While Sparrow improves scheduling speed, its decisions can be poor because it ignores the resource preferences of jobs. Typically concurrent schedulers follow sampling schemes, while centralized systems are paired with sophisticated algorithms.

Figure 1 illustrates the tradeoff between scheduling speed and quality. Figure 1a shows the probability distribution function (PDF) of application performance for three scenarios with analytics jobs of variable duration using Sparrow [28] on a 200-server EC2 cluster. For very short jobs (100 ms ideal duration, i.e., no scheduling delay, and no suboptimal execution due to reasons like interference), fast scheduling allows most workloads to achieve 80% to 95% of the target performance. In contrast, jobs with medium (1–10 s) or long duration (10 s–10 min) suffer significant degradation and achieve 50% to 30% of their ideal performance. As duration increases, jobs become more heterogeneous in their requirements (e.g., preference for high-end cores), and interference between jobs sharing a server matters. In contrast, the scheduling decision speed is not as critical.

Figure 1b shows the PDF of job performance using the Quasar scheduler that accounts for heterogeneity and interference [13]. The centralized scheduler leads to near-optimal performance for long jobs. In contrast, medium and short jobs are penalized by the latency of scheduling decisions, which can exceed the execution time of the shortest jobs. Even if we use multiple schedulers to increase the scheduling throughput [31], the per-job overhead remains prohibitively high.

We present *Tarcil*, a scheduler that achieves the best of both worlds: *high-quality and high-speed* decisions, making it appropriate for large, highly-loaded clusters that host both short and long jobs. Tarcil starts with rich information on the resource preferences and interference sensitivity of incoming jobs [12, 13]. The scheduler then uses sampling to avoid examining the whole cluster state on every decision. There are two main insights in Tarcil's architecture. First, Tarcil uses sampling not merely to find available resources but to identify resources that best match a job's resource preferences. The sampling scheme is derived using analytical methods that provide statistical guarantees on the quality of scheduling decisions. Tarcil additionally adjusts the sample size dynamically based on the quality of available resources. Second, Tarcil uses admission control to avoid scheduling a job that is unlikely to find appropriate resources. To handle

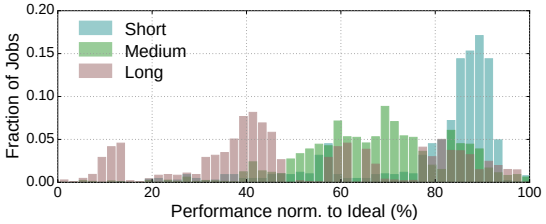


Figure 1a: Sampling-based scheduling.

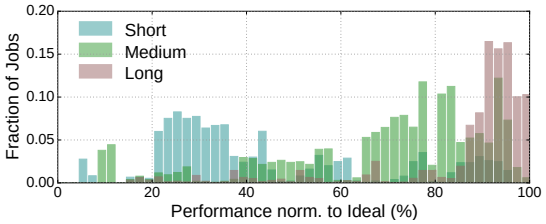


Figure 1b: Centralized scheduling.

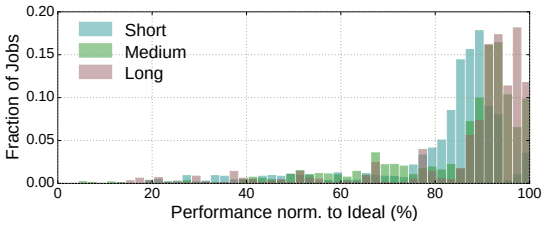


Figure 1c: Tarcil.

Figure 1: Distribution of job performance on a 200-server cluster with concurrent, sampling-based [28] and centralized greedy [12] schedulers and Tarcil for: 1) short, homogeneous Spark [41] tasks (100ms ideal duration), 2) Spark tasks of medium duration (1s–10s), and 3) long Hadoop tasks (10s–10min). Ideal performance (100%) assumes no scheduling overheads and no degradation due to interference. The cluster utilization is 80%.

the tradeoff between long queueing delays and suboptimal allocations, Tarcil uses a small amount of coarse-grain information on the quality of available resources.

We use two clusters with 100 and 400 servers on Amazon EC2 to show that Tarcil leads to low scheduling overheads and predictable, high performance for different workload scenarios. For a heavily-loaded, heterogeneous cluster running short Spark jobs, Tarcil improves average performance by 41% over Sparrow [28], with some jobs running 2–3× faster. For a cluster running diverse applications from various analytics jobs to low-latency services, Tarcil achieves near-optimal performance for 92% of jobs, in contrast with only 22% of jobs with a distributed, sampling-based scheduler and 48% with a centralized greedy scheduler [13]. Finally, Figure 1c shows that Tarcil enables close to ideal performance for the vast majority of jobs of the three scenarios.

2. Background

We now discuss related work on improving the scheduling speed and quality in large, shared datacenters.

Concurrent scheduling: Scheduling becomes a bottleneck for clusters with thousands of servers and high workload churn. An obvious solution is to schedule multiple jobs in parallel [18, 31]. Google's Omega [31], for example, has multiple scheduling agents that can access the whole cluster state concurrently. As long as these agents rarely attempt to assign work to the same servers (infrequent conflicts), they proceed concurrently without delays.

Sampling-based scheduling: Based on results from randomized load balancing [25, 29], we can design sampling-based cluster schedulers [8, 14, 28]. Sampling the state of just a few servers reduces the latency of each scheduling decision and the probability of conflicts between concurrent agents, and is likely to find available resources in non heavily-loaded clusters. The recently-proposed Sparrow scheduler uses *batch sampling* and *late binding* [28]. Batch sampling examines the state of two servers for each of m required cores by a new job and selects the m best cores. If the selected cores are busy, tasks are queued locally in the sampled servers and assigned to the machine where resources become available first.

Heterogeneity & interference-aware scheduling: Hardware heterogeneity occurs in large clusters because servers are populated and replaced progressively during the lifetime of the system [12, 40]. Moreover, the performance of tasks sharing a server may degrade significantly due to interference on shared resources such as caches, memory, and I/O channels [12, 16, 23, 27]. A scheduler can improve performance significantly by taking into consideration the job's resource preferences. For instance, a particular task may perform much better on 2.3 GHz Ivy Bridge cores compared to 2.6 GHz Nehalem cores, while another task may be particularly sensitive to interference from cache-intensive workloads executing on the same socket.

The key challenge in heterogeneity and interference-aware scheduling is determining the preferences of incoming jobs. We start with a system like Quasar that automatically estimates resource preferences and interference sensitivity [12, 13]. Quasar profiles each incoming job for a few seconds on two server types, while two microbenchmarks place pressure on two shared resources. The sparse profiling signal on resource preferences is transformed into a dense signal using collaborative filtering [6, 20, 30, 38], which projects the signal against all information available from previously run jobs. This process identifies similarities in resource and interference preferences, such as the preferred core frequency and cache size for a job, or the memory and network contention it generates. Profiling requires 5-10 seconds, and in Quasar it is performed every time a new application arrives to account for input load changes. Because in this work we also consider real-time analytics applications, which are repeated multiple times, potentially over different data (e.g., daily or hourly), profiling for Quasar and Tarcil is performed only the first time a new application is submitted.

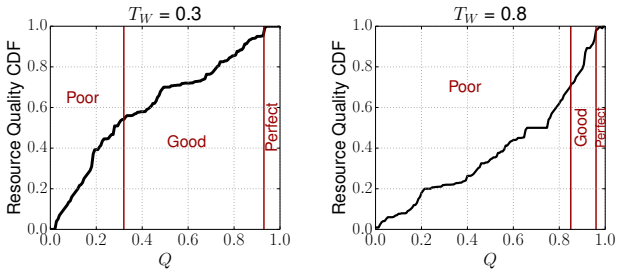


Figure 2: Distribution of resource quality Q for two workloads with $T_W = 0.3$ (left) and $T_W = 0.8$ (right).

3. The Tarcil Scheduler

3.1 Overview

Tarcil is a shared-state scheduler that allows multiple, concurrent agents to operate on the cluster state. In this section, we describe the operation of a single agent.

The scheduler processes incoming workloads as follows. Upon submission, Tarcil first *looks up the job's resource and interference sensitivity preferences* [12, 13]. This information provides estimates of the relative performance on the different server platforms, and of the interference the workload can tolerate and generate in shared resources (caches, memory, I/O channels). Next, Tarcil performs *admission control*. Given coarse-grained statistics of the cluster state, it determines whether the scheduler is likely to quickly find resources of satisfactory quality for a job, or whether it should queue it for a while. Admission control is useful when the cluster is highly loaded. A queued application waits until it has a high probability of finding appropriate resources or until a queueing-time threshold is reached (see Section 4). Statistics on available resources are updated as jobs start and complete.

For admitted jobs, Tarcil performs sampling-based scheduling with the sample size adjusted to satisfy statistical guarantees on the quality of allocated resources. The scheduler also uses batch sampling if a job requests multiple cores. Tarcil examines the quality of sampled resources to select those best matching the job's preferences. It additionally monitors the performance of running jobs. If a job runs significantly below its expected performance, the scheduler adjusts the scheduling decisions, first locally and then via migration. This is useful for long-running workloads; for short jobs, the initial scheduling decision determines performance with little room for adjustments.

3.2 Analytical Framework

We use the following framework to design and analyze sampling-based scheduling in Tarcil.

Resource unit (RU): Tarcil manages resources at RU granularity using Linux containers [10]. Each RU consists of one core and an equally partitioned fraction of the server's memory and storage capacity, and provisioned network bandwidth. For example, a server with 16 cores, 64 GB DRAM,

480 GB of Flash and a 10 Gbps NIC has 16 RUs, each with 1 core, 4 GB DRAM, 30 GB of Flash and 625 Mbps of network bandwidth.

RU quality: The utility an application can extract from an RU depends on the hardware type (e.g., 2 GHz vs 3 GHz core) and the interference on shared resources from other jobs on the same server. The scheduler [12, 13] obtains the interference preferences of an incoming job using a small set of microbenchmarks to inject pressure of increasing intensity (from 0 to 99%) on one of ten shared resources of interest [11]. Interference preferences capture, first, the amount of pressure t_i a job can tolerate in each shared resource $i \in [1, N]$, and second, the amount of pressure c_i it itself will generate in that resource. High values of t_i or c_i imply that a job will tolerate or cause a lot of interference on resource i. t_i and c_i take values in $[0, 99]$. In most cases, jobs that cause a lot of interference in a resource are also sensitive to interference on the same resource. Hence, to simplify the rest of the analysis we assume that $t_i = 99 - c_i$ and express resource quality as a function of caused interference.

Let W be an incoming job and V_W the vector of interference it will cause in the N shared resources, $V_W = [c_1, c_2, ..., c_N]$. To capture the fact that different jobs are sensitive to interference on different resources [23], we reorder the elements of V_W by decreasing value of c_i and get $V'_W = [c_j, c_k, ..., c_n]$, with $c_j \geq c_k \geq ... \geq c_n$. Finally, we obtain a single value for the resource requirements of W using an order-preserving encoding scheme that transforms V'_W to a concatenation of its elements:

$$V_{W_{enc}} = c_j \cdot 10^{(2\cdot(N-1))} + c_k \cdot 10^{(2\cdot(N-2))} + ... + c_n \quad (1)$$

For example, if $V'_W = [84, 31]$ then $V_{W_{enc}} = 8431$. This compactly encodes the values of vector V'_W and preserves their order. Finally, for simplicity we normalize $V_{W_{enc}}$ in $[0, 1]$ and derive the target resource quality for job W:

$$T_W = \frac{V_{W_{enc}}}{10^{2N} - 1},\ T_W \in [0, 1] \quad (2)$$

A high value for the quality target T_W implies that job W is resource-intensive. Its performance will depend a lot on the quality of the scheduling decision.

We now need to find RUs that closely match this target quality. To determine if an available resource unit H is appropriate for job W, we calculate the interference caused on this RU by all other jobs occupying RUs on the same server. Assuming M resource units in the server, the total interference H experiences on resource i is:

$$C_i = \frac{\sum_{m \neq H} c_i}{M - 1} \quad (3)$$

Starting with vector $V_H = [C_1, C_2, ..., C_N]$ for H and using the same reordering and order-preserving encoding as for T_W, we calculate the quality of resource H as:

$$U_H = 1 - \frac{V_{H_{enc}}}{10^{2N} - 1},\ U_H \in [0, 1] \quad (4)$$

The higher the interference from colocated tasks, the lower U_H will be. Resources with low U_H are more appropriate for jobs that can tolerate a lot of interference and vice versa. Note that the ordering of C_i in $V_{H_{enc}}$ follows the resource ordering in vector V'_W of the application we want to schedule, i.e., resource j which is the most critical resource for job W is ordered first. This ensures that RUs are judged with respect to the resources a new application is most sensitive to.

Comparing U_H for an RU against T_W allows us to judge the quality of resource H for incoming job W:

$$Q = \begin{cases} 1 - (U_H - T_W) & \text{, if } U_H \geq T_W \\ T_W - U_H & \text{, if } U_H < T_W \end{cases} \quad (5)$$

If Q equals 1, we have an ideal assignment with the server tolerating as much interference as the new job generates. If Q is within $[0, T_W]$, selecting RU H will degrade the job's performance. If Q is within $(T_W, 1)$, the assignment will preserve the workload's performance but is suboptimal. It would be better to assign a more demanding job on this resource unit.

Resource quality distribution: Figure 2 shows the distribution of Q for a 100-server cluster with ∼800 RUs (see Section 6 for cluster details) and 100 10-min Hadoop jobs as resident load (50% cluster utilization). For a non-demanding new job with $T_W = 0.3$ (left), there are many appropriate RUs at any point. In contrast, for a demanding job with $T_W = 0.8$, only a small number of resources will lead to good performance. Obviously, the scheduler must adjust the sample size for new jobs based on T_W.

3.3 Scheduling with Guarantees

We can now derive the sample size that provides statistical guarantees on the quality of scheduling decisions.

Assumptions and analysis: To make the analysis independent of application characteristics, we make Q an absolute ordering of RUs in the cluster. Starting with Equation 5, we sort RUs based on Q for incoming job W, breaking any ties in quality with a fair coin. Subsequently, we compute a uniformly distributed rank of resource quality D. For the i^{th} value of sorted Q, D is $D(i) = i/(N_{RU} - 1)$, with $i \in [0, N_{RU} - 1]$ and N_{RU} the number of total RUs. Because D is now *uniformly distributed*, we can derive the sample size in the following manner.

Assume that the scheduler samples R RU candidates for each RU needed by an incoming workload. If we treat the qualities of these R candidates as random variables D_i ($D_1, D_2, ..., D_R \sim U[0, 1]$) that are *uniformly distributed* by construction and statistically independent from each other ($i.i.d$), we can derive the distribution of quality D after sampling. The cumulative distribution function (CDF) of the resource quality of each candidate is:

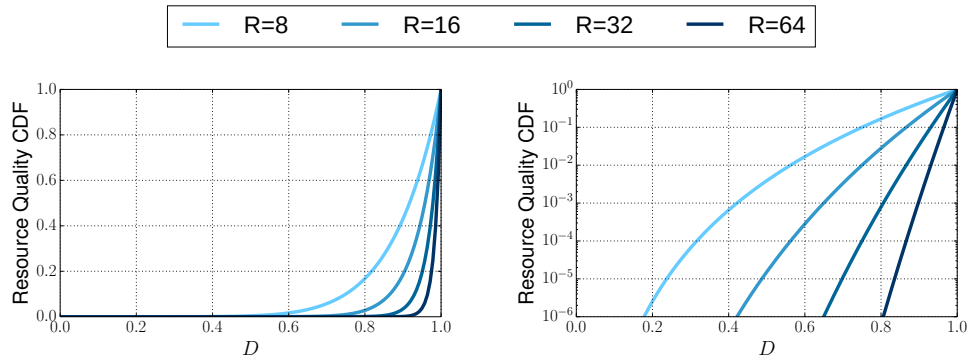


Figure 3: Resource quality CDFs under the uniformity assumption in linear and log scale for R=8, 16, 32 and 64.

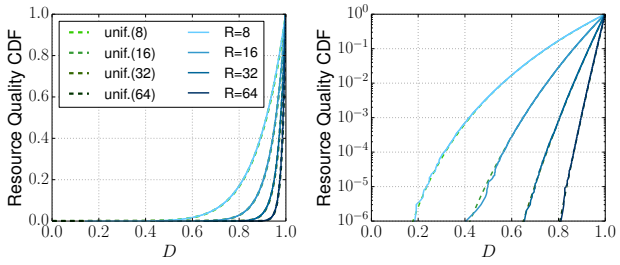


Figure 4: Comparison of resource quality CDFs under the uniformity assumption, and as measured in a 100-server cluster.

$F_{D_i}(x) = Prob(D_i \leq x) = x, x \in [0,1]$[1]. Since the candidate with the highest quality is selected from the sampled set, its resource quality is the max order random variable $A = max\{D_1, D_2, ..., D_R\}$, and its CDF is:

$$F_A(x) = Prob(A \leq x) = Prob(D_1 \leq x \wedge ... \wedge D_R \leq x)$$
$$= Prob(D_i \leq x)^R = x^R,\ x \in [0,1] \quad (6)$$

This implies that the distribution of resource quality after sampling *only* depends on the sample size R. Figure 3 shows CDFs of resource quality distributions under the uniformity assumption, for sample sizes $R = \{8, 16, 32, 64\}$. The higher the value of R, the more skewed to the right the distribution is, hence the probability of finding only candidates of low quality quickly diminishes to 0. For example, for $R = 64$ there is a 10^{-6} probability that none of the sampled RUs will have resource quality of at least $D = 80\%$, i.e., $Prob(D < 0.8 | \forall\ RU) = 10^{-6}$.

Figure 4 *validates the uniformity assumption* on a 100-server EC2 cluster running short Spark tasks (100 ms ideal duration) and longer Hadoop jobs (1-10 min). The cluster load is 70-75% (see methodology in Sec. 6). In all cases, the deviation between the analytically derived and measured distributions of D is minimal, which shows that the analysis above holds in practice. In general, the larger the cluster, the more closely the distribution approximates uniformity.

[1] This assumes D_i to be continuous variables, although in practice they are discrete. This makes the analysis independent of the cluster size N_{RU}. The result holds for the discretized version of the equation.

Large jobs: For jobs that need multiple RUs, Tarcil uses *batch sampling* [28, 29]. For m requested units, the scheduler samples $R \cdot m$ RUs and selects the m best among them as shown in Figure 5a. Some applications experience locality between sub-tasks or benefit from allocating all resources in a small set of machines (e.g., within a single rack). In such cases, for each sampled RU, Tarcil examines its neighboring resources and makes a decision based on their aggregate quality (Figure 5b). Alternatively, if a job prefers distributing its resources the scheduler will allocate RUs in different machines, racks and/or cluster switches, assuming knowledge of the cluster's topology. Placement preferences for reasons such as security [32] can also be specified in the form of attributes at submission time by the user.

Sampling at high load: Equation 6 estimates the probability of finding near-optimal resources accurately when resources are not scarce. When the cluster operates at high load, we must increase the sample size to guarantee the *same probability* of finding a *candidate of equally high quality*, as when the system is unloaded. Assume a system with $N_{RU} = 100$ RUs. Its discrete CDF is $F_A(x) = P[A \leq x] = x, x = 0, 0.01, 0.02, ..., 1$. For sample size R, this becomes: $F_A(x) = x^R$, and a quality target of $Pr[D < 0.8] = 10^{-3}$ is achieved with $R = 32$. Now assume that 60% of the RUs are already busy. If, for example, only 8 of the top 20 candidates for this task are available at this point, we need to set R s.t. $Pr[D < 0.92] = 10^{-3}$, which requires a sample size of $R = 82$. Hence, the sample size for a highly loaded cluster can be quite high, degrading scheduling latency. In the next section, we introduce an admission control scheme that bounds sample size and scheduling latency, while still allocating high-quality resources.

4. Admission Control

When available resources are plentiful, jobs are immediately scheduled using the sampling scheme described in Section 3. However, when load is high, the sample size needed to find resources of sufficient quality may become quite large. Tarcil employs a simple admission control scheme that queues jobs until appropriate resources become available and estimates how long an application should wait at admission control.

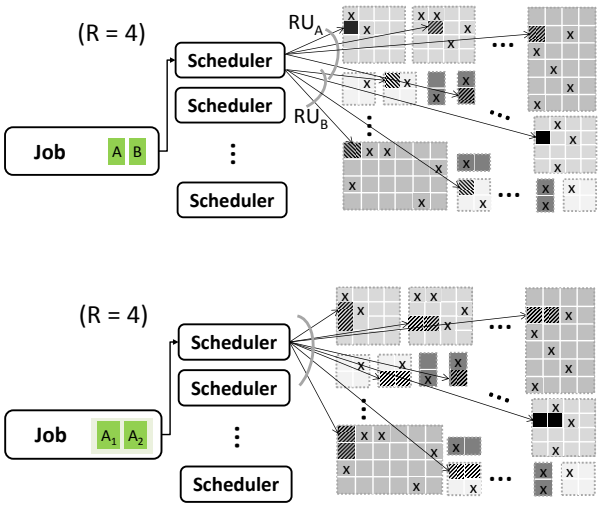


Figure 5: Batch sampling in Tarcil with $R = 4$ for (a) a job with two independent tasks A and B, and (b) a job with two subtasks A_1 and A_2 that exhibit locality. x-marked RUs are already allocated, striped RUs are sampled, and solid black RUs are allocated to the new job.

A simple indication to trigger job queueing is the count of available RUs. This, however, does not yield sufficient insight into the quality of available resources. If most RUs have poor quality for an incoming job, queueing may be preferable. Unfortunately, a naïve quality check involves accessing the whole cluster state, with prohibitive overheads. Instead, we maintain a small amount of coarse-grain information which allows for a fast check. We leverage the information on contention scores that is already maintained per RU to construct a contention score vector $[C_1 C_2 ... C_N]$ from the contention C_i the RU experiences in each resource, due to interference from neighboring RUs. We use *locality sensitive hashing* (LSH) based on random selection to hash these vectors into a small set of buckets [1, 9, 30]. LSH computes the cosine distance between vectors and assigns RUs with similar contention scores in the respective resources to the same bucket. We *only* keep a single count of available RUs for each bucket. The hash for an RU (and the counter of that bucket) is recalculated upon instantiation or completion of a job. Updating the per-bucket counters is a fast operation, out of the critical path for scheduling. Note that excluding updates in RU status, LSH is only performed once.

Admission control works as follows. We check the bucket(s) that correspond to the resources with quality that matches the incoming job's preferences. If these buckets have counters close to the number of RUs the job needs, the application is queued. Admission control may penalize resource-demanding jobs, for which RUs take longer to be freed. To ensure this penalty is not severe, we impose an upper limit for queueing time. Therefore, queued applications wait until the probability that resources are freed increases or until an upper bound for waiting time is reached. To estimate waiting

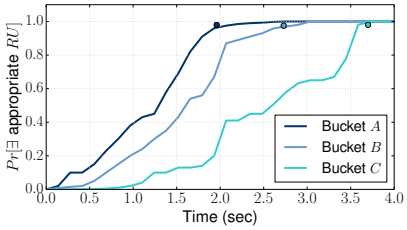


Figure 6: Actual and estimated (dot) probability for a target RU to exist as a function of waiting time.

time, Tarcil records the rate at which RUs of each bucket became available in recent history. Specifically, it uses a simple feedback loop to estimate when the probability that an appropriate RU exists approximates 1 for a target bucket. The distribution is updated every time an RU from that bucket is freed. Tarcil also sets an upper bound for waiting time at $\mu + 2 \cdot \sigma$, where μ and σ are the mean and standard deviation of the corresponding "time-until-free" PDF. If the estimated waiting time is less than the upper bound, the job waits for resources to be freed; otherwise it is scheduled to avoid excessive queueing. Although admission control adds some complexity, in practice it only delays workloads at very high cluster utilizations (over 80%-85%).

Validation of waiting time estimation: Figure 6 shows the probability that a desired RU will become available within time t for different buckets in a heterogeneous 100-server EC2 cluster running short Spark tasks and long Hadoop jobs. The cluster utilization is approximately 85%. We show the probabilities for r3.2xlarge (8 vCPUs) instances with CPU contention (A), r3.2xlarge instances with network contention (B), and c3.large (2 vCPUs) instances with memory contention (C). The distributions are obtained from recent history and vary across buckets. The dot in each line shows the estimated waiting time by Tarcil. There is less than 8% deviation between estimated and measured time for an appropriate RU to be freed. In all experiments we use 20 buckets and a 2-hour history. This was sufficient to make accurate estimations of available resources, however, the bucket count and/or history length may vary for other systems.

5. Tarcil Implementation

5.1 Tarcil Components

Figure 7 shows the components of the scheduler. Tarcil is a distributed, shared-state scheduler so, unlike Quincy or Mesos, it does not have a central coordinator [18, 19]. Scheduling agents work in parallel, are load-balanced by the cluster front-end, and have a local copy of the shared server state, which contains the list and status of all RUs.

Since all schedulers have full access to the cluster state, conflicts are possible. Conflicts between agents are resolved using *lock-free optimistic concurrency* [31]. The system maintains one resilient master copy of state in a separate server. Each scheduling agent has a local copy of this

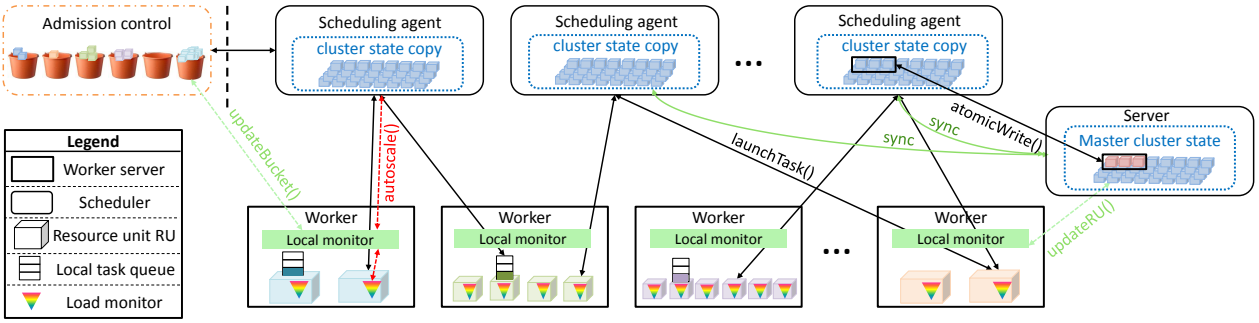


Figure 7: The different components of the scheduler and their interactions.

state which is updated frequently. When an agent makes a scheduling decision it attempts to update the master state copy using an atomic write operation. Once the commit is successful the resources are yielded to the corresponding agent. Any other agent with conflicting decisions needs to resample resources. The local copy of state of each agent is periodically synced with the master (every 5-10 s). The timing of the updates includes a small random offset such that the master does not become the bottleneck. When the sample size is small, decisions of scheduling agents rarely overlap and each scheduling action is fast ($\sim 10 - 20$ms, for a 100-server cluster and $R = 8$, over an order of magnitude faster than centralized approaches). When the number of sampled RUs increases beyond $R = 32$ for very large jobs, conflicts can become more frequent, which we resolve using incremental transactions on the non-conflicting resources [31]. If a scheduling agent crashes, an idle cluster server resumes its role, once it has obtained a copy of the master state.

Each worker server has a *local monitor* module that handles scheduling requests, federates resource usage in the server, and updates the RU quality. When a new task is assigned to a server by a scheduling agent, the monitor updates the RU status in the master copy and notifies the agent and admission control. Finally, a per-RU *load monitor* evaluates performance in real time. When a job's performance deviates from its expected target, the monitor notifies the proper agent for a possible allocation adjustment. The load monitor also informs agents of CPU or memory saturation, which triggers resource autoscaling (see Section 5.2).

We currently use Linux containers to partition servers into RUs [3]. Containers enable CPU, memory, and I/O isolation. Each container is configured to a single core and a fair share of the memory and storage subsystem, and network bandwidth. Containers can be merged to accommodate multicore workloads, using *cgroups*. Virtual machines (VMs) could also be used to enable migration [27, 36, 37, 39], but would incur higher overheads.

Figure 8 traces a scheduling event. Once a job is submitted, admission control evaluates whether it should be queued. Once the assigned scheduling agent sets the sample size according to the job's constraints, it samples the shared cluster state for the required number of RUs. Sampling happens locally in each agent. The agent computes the resource quality of sampled resources and selects the ones allocated to the job. The selection takes into account the resource quality and platform preferences, and any locality preferences of a task. The agent then attempts to update the master copy of the state. Upon a successful commit it notifies the local monitor of the selected server(s) over RPC and launches the task in the target RU(s). The local monitor notifies admission control, and the master copy to update their state. Once the task completes, the local monitor issues updates to the master state and notifies the agent and admission control; the scheduling agent then informs the cluster front-end.

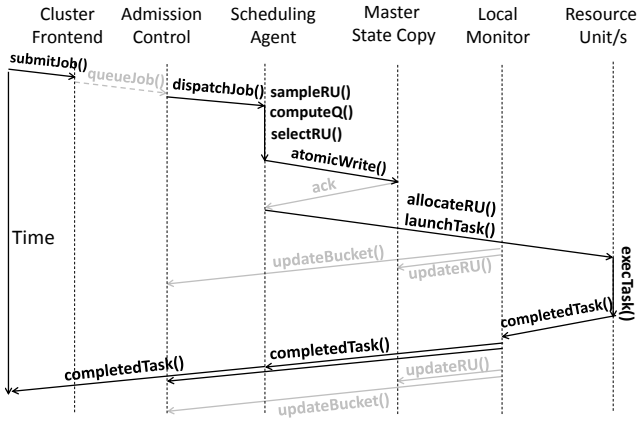


Figure 8: Trace of a scheduling event in Tarcil.

5.2 Adjusting Allocations

For short-running tasks, the quality of the initial assignment is particularly important. For long-running tasks, we must also consider the different phases the program goes through [21]. Similarly, we must consider cases where Tarcil makes a suboptimal allocation due to inaccurate classification, deviations from fully random selection in the sampling process, or a compromise in resource quality at admission control. Tarcil uses the per-server *load monitor* to measure the performance of active workloads in real time. This can correspond to instructions per second (IPS), packets per second or a high-level application metric, depending on the application type. Tarcil compares this metric to any

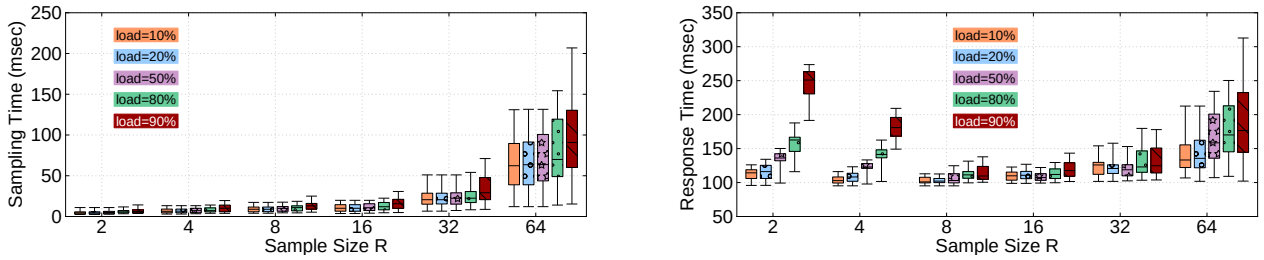


Figure 9: Sensitivity of sampling overheads and response times to sample size.

performance targets the job provides or are available from previous runs of the same application. If there is a large deviation, the scheduler takes action. Since we are using containers, the primary action we take is to avoid scheduling other jobs on the same server. For scale-out workloads, the system also employs a simple *autoscale* service which allocates more RUs (locally or not) to improve the job's performance.

5.3 Priorities

Users can submit jobs with priorities. Jobs with high priority bypass others during admission and preempt lower-priority jobs during resource selection. Tarcil also allows users to select between incremental scheduling, where tasks from a job get progressively scheduled as resources become available and all-or-nothing gang scheduling, where either all or no task is scheduled. We evaluate priorities in Section 6.1.

6. Evaluation

6.1 Tarcil Analysis

We first evaluate Tarcil's scalability and its sensitivity to parameters such as the sample size and task duration.

Sample size: Figure 9 shows the sensitivity of sampling overheads and response times to the sample size for homogeneous Spark tasks with 100 ms duration and cluster loads varying from 10% to 90% on the 110-server EC2 cluster. All machines are r3.2xlarge memory-optimized instances (61 GB of RAM). 10 servers are used by the scheduling agents, and the remaining 100 serve incoming load. The boundaries of the boxplots depict the 25th and 75th percentiles, the whiskers the 5th and 95th percentiles and the horizontal line shows the mean. As sample size increases, the overheads increase. Until $R = 32$ overheads are marginal even at high loads, but they increase substantially for $R \geq 64$, primarily due to the overhead of resolving conflicts between the 10 scheduling agents. Hence, we cap sample size to $R = 32$ even under high load. Response times are more sensitive to sample size. At low load, high quality resources are plentiful and increasing R makes little difference to performance. As load increases, sampling with $R = 2$ or $R = 4$ is unlikely to find good resources. $R = 8$ is optimal for both low and high cluster loads in this scenario.

Number of scheduling agents: We now examine how the number of agents that perform concurrent scheduling actions affects the quality and latency of scheduling. Figure 10a shows how scheduling latency changes as we increase the number of scheduling agents. The cluster load varies again from 10% to 90%, and the load is the same homogeneous Spark tasks with 100msec optimal duration, as before. We set the sample size to $R = 8$, which was the optimal, based on the previous experiment. When the number of schedulers is very small (below 3), latency suffers at high loads due to limited scheduling parallelism. As the number of agents increases latency drops, until 12 agents. Beyond that point, latency slowly increases due to increasing conflicts among agents. For larger cluster sizes, the same number of agents would not induce as many conflicts. Figure 10b shows how the fraction of tasks that meet QoS changes as the number of scheduling agents increases. As previously seen, if the number of agents is very small, many jobs experience increased response times. As more agents are added, the vast majority of jobs meet their QoS until high cluster loads. When cluster load exceeds 80%, QoS violations are caused primarily due to queueing at admission control, instead of limited scheduling concurrency. In general, 3 scheduling agents are sufficient to get the minimum scheduling latency; in following comparisons with Sparrow we use 10 agents to ensure a fair comparison, since Sparrow uses a 10:1 worker to agent ratio.

Cluster load: Figure 11a shows the average and 95th percentile response times when we scale the cluster load in the 110-server EC2 cluster. The incoming jobs are homogeneous Spark tasks with 100msec target duration. We increase the task arrival rate to increase the load. The target performance of 100msec includes no overheads or degradation due to suboptimal scheduling. The reported response times include the task execution and all overheads. The mean of response times with Tarcil remains almost constant until loads over 85%. At very high loads, admission control and the large sample size increase the scheduling overheads, affecting performance. The 95th percentile is more volatile, but only exceeds 250msec at loads over 80%. Tasks with very high response times are typically those delayed by admission control. Sampling itself adds marginal overheads until 90% load. At very high loads scheduling overheads are dominated by queueing time and increased sample sizes.

Task duration: Figure 11b shows the average and 95th percentile response times as a function of task duration, which

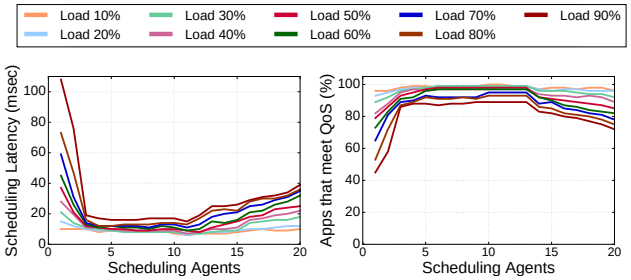


Figure 10: Sensitivity to the number of concurrent scheduling agents. Figure 10a shows the scheduling latency, and Figure 10b the fraction of jobs that meet QoS.

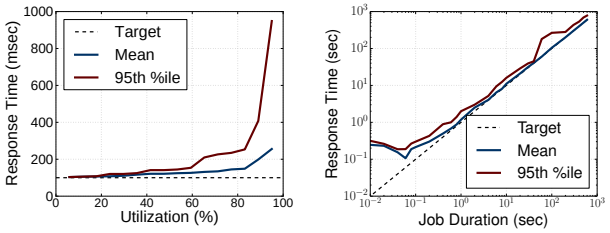


Figure 11: Task response times with (a) system load, and (b) task duration at constant load.

ranges from 10msec to 600sec. The cluster load is 80% in all cases. For long tasks the mean and 95th percentile closely approximate the target performance. When task duration is below 100msec, the scheduling overhead dominates. Despite this, the mean and tail latency remain very close, which shows that performance unpredictability is limited.

Priorities: We now examine the impact of task priorities on response time. Figure 12a shows the distribution of response times for the same 100msec homogeneous Spark tasks when increasing the fraction of tasks with high priority. 10% of high priority tasks corresponds to a set of randomly-selected tasks that bypass all prior-enqueued tasks. System load is at 80%. Figure 12b shows the impact of bypassing on the remaining low priority tasks. In all cases, we compare response times to the system with no priorities. The workload scenarios are identical, meaning that the task comparison between systems with and without priorities is one-to-one. Since high priority tasks can bypass others, their response time is reduced, as seen in the left figure. On the other hand low priority tasks may experience increased queueing time, especially when more than a third of tasks have high priority.

6.2 Comparison Between Schedulers

Methodology: We compare Tarcil to Sparrow [28] and Quasar [13]. Sparrow uses a sampling ratio of $R = 2$ servers for every core required, as recommended in [28]. Quasar has a centralized greedy scheduler that searches the cluster state with a timeout of 2 seconds. Sparrow does not take into account heterogeneity or interference preferences for incoming jobs, while Tarcil and Quasar do. We evaluate these schedulers on the same 110-server EC2 cluster with r3.2xlarge memory-optimized instances. 10 servers are dedicated to the scheduling agents for Tarcil and Sparrow and a single server for Quasar. Note that utilization in the scheduling agents for Tarcil is very low, less than 10%; we select 10 agents to ensure a fair comparison with Sparrow. While we could replicate Quasar's scheduler for fault tolerance, it would not improve the latency of each scheduling decision. Additionally, Quasar schedules applications at job, not task granularity, which reduces its scheduling load. Unless otherwise specified, Tarcil uses sample sizes of $R = 8$ during low load.

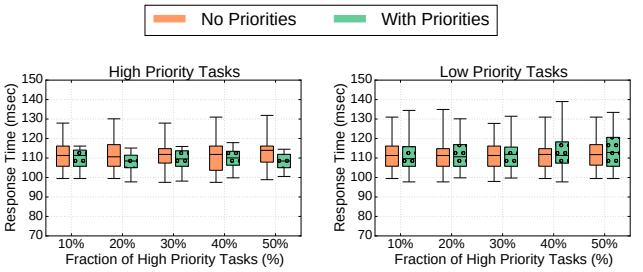
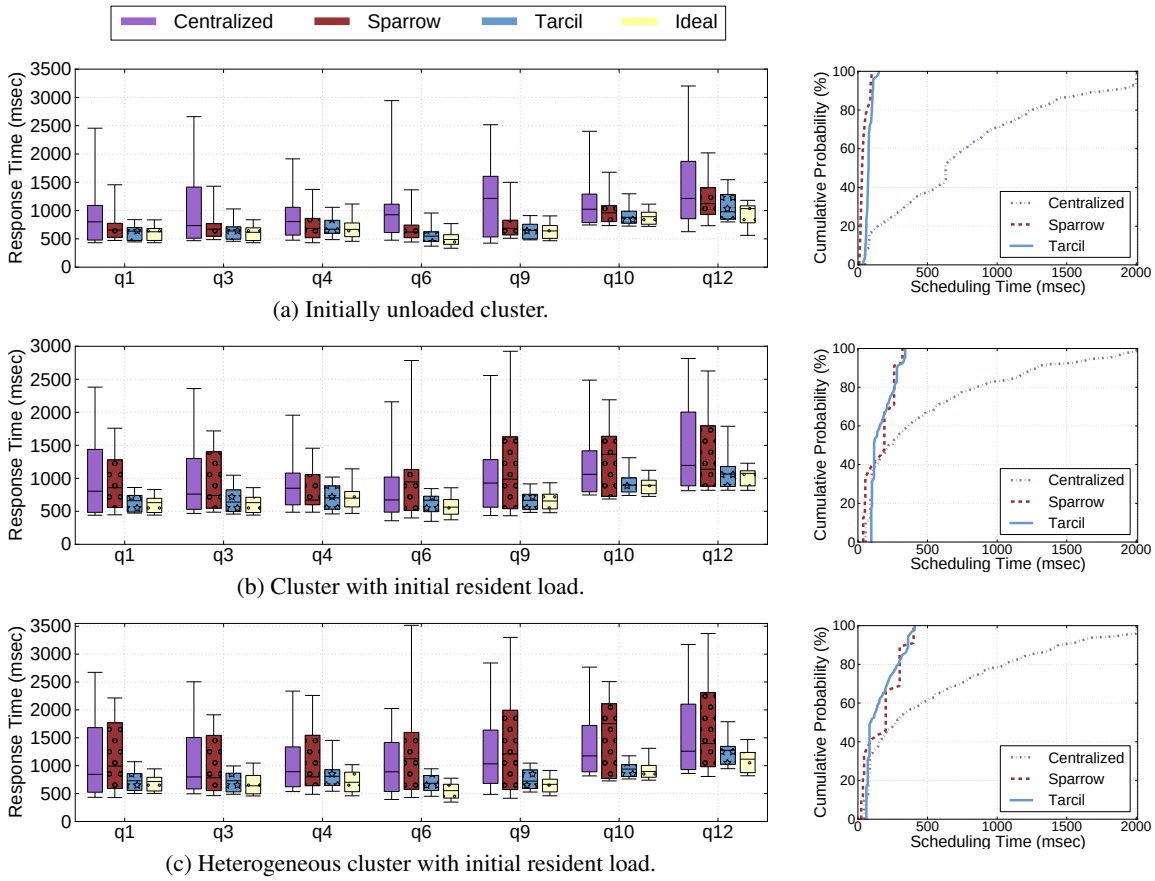


Figure 12: Task response time in the presence of priorities. Figure 12a shows the response time of the high priority tasks, and Figure 12b for the low priority tasks.

6.2.1 TPC-H workload

We compare the three schedulers on the TPC-H benchmark. TPC-H is a standard proxy for ad-hoc, low-latency queries that comprise a large fraction of load in shared clusters. We use a similar setup as the one used to evaluate Sparrow [28]. TPC-H queries are compiled into Spark tasks using Shark [15], a distributed SQL data analytics platform. The Spark plugin for Tarcil is 380 lines of code in Scala. Each task triggers a scheduling request for the distributed schedulers (Tarcil and Sparrow), while Quasar schedules jointly all tasks from the same computation stage. We constrain tasks in the first stage of each query to the machines holding their input data (3-way replication). All other tasks are unconstrained. We run each experiment for 30 minutes, with multiple users submitting randomly-ordered TPC-H queries to the cluster. The results discard the initial 10 minutes (warm-up) and capture a total of 40k TPC-H queries and approximately 134k jobs. Utilization at steady state is 75-82%.

Unloaded cluster: We first examine the case where TPC-H is the only workload in the cluster. Figure 13a shows the response times for seven representative query types [42]. Response times include all scheduling and queueing overheads. Boundaries show the 25th and 75th percentiles and whiskers the 5th and 95th percentiles. The ideal scheduler corresponds to a system that identifies resources of optimal quality with zero delay. Figure 13a shows that the centralized scheduler experiences the highest variability in performance. Although some queries complete very fast because they receive high quality resources, most experience high scheduling delays.



(a) Initially unloaded cluster.

(b) Cluster with initial resident load.

(c) Heterogeneous cluster with initial resident load.

Figure 13: Response times for different query types (left) and CDFs of scheduling overheads (right).

To verify this, we also show the scheduling time CDF on the right of Figure 13a. While Tarcil and Sparrow have tight bounds on scheduling overheads, the centralized scheduler adds up to 2 seconds of delay (timeout threshold). Comparing query performance between Sparrow and Tarcil, we see that the difference is small, 8% on average. Tarcil approximates the ideal scheduler more closely, as it accounts for each task's resource preferences. Additionally, Tarcil constrains performance unpredictability. The 95th percentile is reduced by 80%-2.4x compared to Sparrow.

Cluster with resident load: The difference in scheduling quality becomes more clear when we introduce cross-application interference. Figure 13b shows a setup where 40% of the cluster is busy servicing background applications, including other Spark jobs, long Hadoop workloads, and latency-critical services like memcached. These jobs are *not* scheduled by the examined schedulers. While the centralized scheduler still adds considerable overhead to each job (Figure 13b, right), its performance is now comparable to Sparrow. Since Sparrow does not account for sensitivity to interference, the response time of queries that experience resource contention is high. Apart from average response time, the 95th percentile also increases significantly. In contrast, Tarcil accounts for resource preferences and only places tasks on machines with acceptable interference levels. It maintains an average performance only 6% higher compared to the unloaded cluster across query types. More importantly, it preserves the low performance jitter by bounding the 95th percentile of response times.

Heterogeneous cluster with resident load: Next, in addition to interference, we also introduce hardware heterogeneity. The cluster size remains constant but 75% of the worker machines are replaced with less or more powerful servers, ranging from general purpose medium and large instances to quadruple compute- and memory-optimized instances. Figure 13c shows the new performance for the TPC-H queries. As expected, response times increase, since some of the high-end machines are replaced by less powerful servers. More importantly, performance unpredictability increases when the resource preferences of incoming jobs are not accounted for. In some cases ($q9$, $q10$), the centralized scheduler now outperforms Sparrow despite its higher scheduling overheads. Tarcil achieves response times close to the unloaded cluster and very close to the ideal scheduler.

6.2.2 Impact on Resident Memcached Load

Finally, we examine the impact of scheduling on resident load. In the same heterogeneous cluster (110 EC2 nodes with

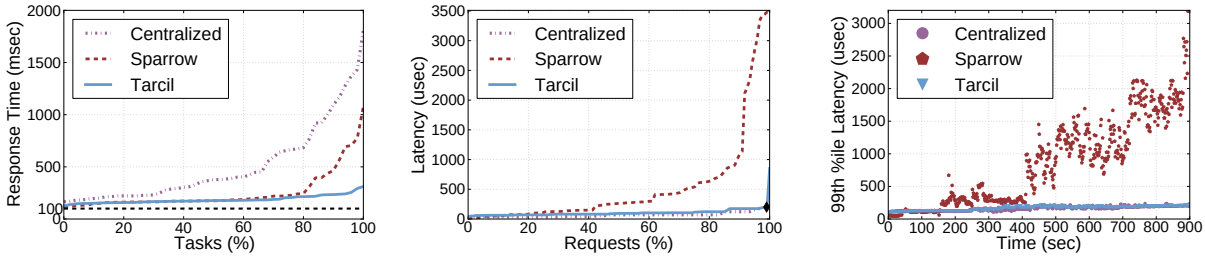


Figure 14: Performance of scheduled Spark tasks and resident memcached load (aggregate and over time).

10 schedulers), we place long-running memcached instances as resident load. These instances serve read and write queries following the Facebook `etc` workload [2]. `etc` is the large memcached deployment in Facebook, has a 3:1 read:write ratio, and a value distribution between 1B and 1KB. Memcached occupies about 40% of the system capacity and has a QoS target of 200usec for the 99th latency percentile.

The incoming jobs are homogeneous, short Spark tasks (100msec ideal duration, 20 tasks per job) that perform logistic regression. A total of 300k jobs are submitted over 900 seconds. Figure 14a shows the response times of the Spark tasks for the three schedulers. The centralized scheduler adds significant overheads, while Sparrow and Tarcil add small overheads and behave similarly for 80% of the tasks. For the remaining tasks, Sparrow increases response times significantly, as it is unaware of the interference induced by memcached. Tarcil maintains low response times for most tasks.

It is also important to consider the impact on memcached. Figure 14b shows the latency CDF of memcached requests. The black diamond depicts the QoS constraint of tail latency. With Tarcil and the centralized scheduler, memcached behaves well as both schedulers minimize interference. Sparrow, however, leads to large latency increases for memcached, since it does not account for resource preferences. Figure 14c shows how the 99th percentile of memcached changes as load increases. Initially memcached meets its QoS for all three schedulers. As the cluster becomes more loaded the tail latency increases significantly for Sparrow.

Note that a naïve coupling of Sparrow – for short jobs – with Quasar – for long jobs – is inadequate for three reasons. First, Tarcil achieves higher performance for short tasks because it considers their resource preferences. Second, even if the long jobs were scheduled with Quasar, using Sparrow for short tasks would degrade its performance. Third, while the difference in execution time achieved by Quasar and Tarcil for long jobs is small, scheduling overheads are significantly reduced, without sacrificing scheduling quality.

6.3 Large-Scale Evaluation

Methodology: We also evaluated Tarcil on a 400-server EC2 cluster with 10 server types ranging from 4 to 32 cores. The total core count in the cluster is 4,178. All servers are dedicated and managed only by the examined schedulers and there is no external interference from other workloads.

We use applications including short Spark tasks, longer Hadoop jobs, streaming Storm jobs [34], latency-critical services (memcached [22] and Cassandra [7]), and single-server benchmarks (SPECCPU2006, PARSEC [5]). In total, 7,200 workloads are submitted with 1-second inter-arrival times. These applications stress different resources, including CPU, memory and I/O (network, storage). We measure job performance (from submission to completion), cluster utilization, scheduling overheads and scheduling quality.

We compare Tarcil, Quasar and Sparrow. Because this scenario includes long-running jobs, such as memcached, that are not supported by the open-source implementation of Sparrow, we use Sparrow when applicable (e.g., Spark) and a Sampling-based scheduler that follows Sparrow's principles (batch sampling and late binding) for the remaining jobs.

Performance: Figure 15a shows the performance (time between submission and completion) of the 7,200 workloads ordered from worst to best-performing, and normalized to their optimal performance. Optimal corresponds to the performance on the best available resources and zero scheduling delay. The Sampling-based scheduler degrades performance for more than 75% of jobs. While Centralized behaves better, achieving an average of 82% of optimal, it still violates QoS for a large fraction of applications, particularly short-running workloads (0-3900 for this scheduler). Tarcil outperforms both schedulers, leading to 97% average performance and bounding maximum performance degradation to 8%.

Cluster utilization: Figure 15b shows the system utilization across the 400 servers of the cluster when incoming jobs are scheduled with Tarcil. CPU utilization is averaged across the cores of each server, and sampled every 2 sec. Utilization is 70% on average at steady-state (middle of the scenario), when there are enough jobs to keep servers load-balanced. The maximum in the x-axis is set to the time it takes for the Sampling-based scheduler to complete the scenario ($\sim 35,000$ sec). The additional time corresponds to jobs that run on suboptimal resources and take longer to complete.

Core allocation: Figure 15c shows a snapshot of the RU quality across the cluster as observed by the job that is occupying each RU when using Tarcil. The snapshot is taken at $8,000$s when all jobs have arrived and the cluster operates at maximum utilization. White tiles correspond to unallocated resources. Dark blue tiles denote jobs with resources

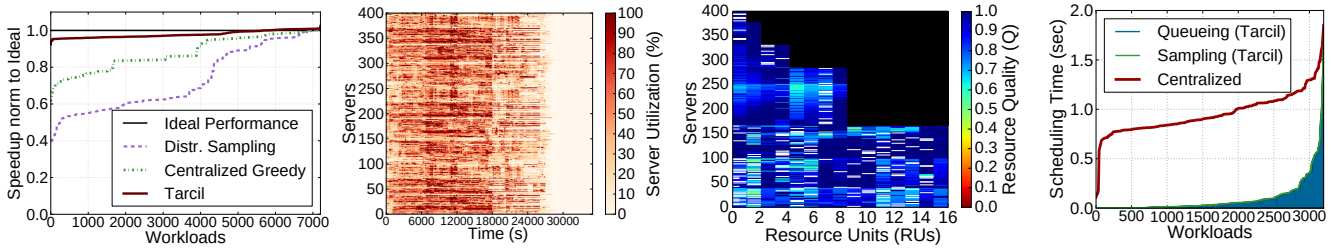


Figure 15: (a) Performance across 7,200 jobs on a 400-server EC2 cluster for the Sampling-based and Centralized schedulers and Tarcil, normalized to optimal performance, (b) cluster utilization with Tarcil throughout the duration of the experiment, (c) quality of resource allocation across all RUs, and (d) scheduling overheads in Tarcil and the Centralized scheduler.

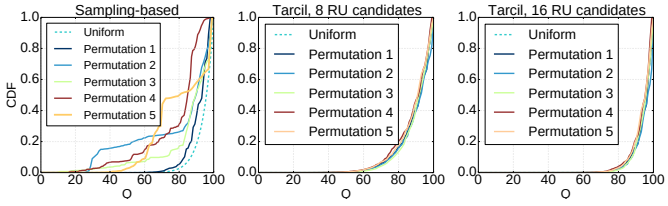


Figure 16: Resource quality distributions for the Sampling-based scheduler and Tarcil with $R = 8$ and 16 RUs across different permutations of the EC2 scenario.

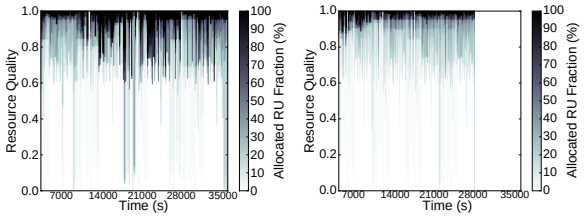


Figure 17: Resource quality CDFs for: (a) the Sampling-based scheduler, and (b) Tarcil.

very close to their target quality. Lighter blue corresponds to jobs that received good but suboptimal resources. The graph shows that the majority of jobs are given suitable resources. Note that high Q does not imply low server utilization. Cluster utilization at the time of the snapshot is 75%.

Scheduling overheads: Figure 15d shows the scheduling overheads for the Centralized scheduler and Tarcil. The results on scheduling overheads are consistent with the TPC-H experiment in Section 6.2. The overheads of the Centralized scheduler increase significantly with scale, adding approximately 1 sec to most workloads. Tarcil keeps overheads low, adding less than 150msec to more than 80% of workloads. This is essential for scalability. At high load, Tarcil increases the sample size to preserve the statistical guarantees and/or resorts to local queueing. Beyond the first 4,000 workloads, most scheduling overheads are due to queueing. The overheads for the Sampling-based scheduler are similar to Tarcil.

Predictability: Figure 17 shows the fraction of allocated RUs that are over a certain resource quality at each point during the scenario. Results are shown for the Sampling-based scheduler (left) and Tarcil (right). Darker colors towards the bottom of the graph denote that most of allocated RUs have poor quality. At time $16,000$sec, when the cluster is highly-loaded, the Sampling-based scheduler leads to 70% of allocated cores having quality less than 0.4. For Tarcil, only 18% of cores have less than 0.9 quality. Also note that, as the scenario progresses, the Sampling-based scheduler starts allocating resources of worse quality, while Tarcil maintains almost the same quality throughout the experiment.

Figure 16 explains this dissimilarity. It shows the CDF of resource quality for this scenario, and 5 random permutations of it (different job submission order). We show the CDF for the Sampling-based scheduler and Tarcil with 8 and 16 candidates. We omit the centralized scheduler which achieves high quality most of the time. The sampling-based scheduler deviates significantly from the uniform distribution, since it does not account for the quality of allocated resources. In contrast, Tarcil closely follows the uniform distribution, improving performance predictability.

7. Conclusions

We have presented Tarcil, a cluster scheduler that improves both scheduling speed and quality, making it appropriate for large, highly-loaded clusters running short and long jobs. Tarcil uses an analytically-derived sampling framework that provides guarantees on the quality of allocated resources, and adjusts the sample size to match application preferences. It also employs admission control to avoid poor scheduling decisions at high load. We have compared Tarcil to existing parallel and centralized schedulers for various workloads on 100- and 400-server clusters on Amazon EC2. We have showed that it provides low scheduling overheads, high application performance, and high cluster utilization. Moreover, it reduces performance jitter, improving predictability in large, shared clusters.

Acknowledgments

The authors sincerely thank David Shue, Mendel Rosenblum, John Ousterhout, and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was supported by the Stanford Experimental Datacenter Lab and NSF grant CNS-1422088. Christina Delimitrou was supported by a Facebook Graduate Fellowship.

References

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proc. of the 47th annual IEEE symp. on Foundations of Computer Science*, 2006.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, 2012.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of the 3rd USENIX symp. on Operating Systems Design and Implementation*, 1999.

[4] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3), 2013.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. of the 17th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2008.

[6] L. Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In *Proc. of the International Conference on Computational Statistics (COMPSTAT)*, 2010.

[7] Apache Cassandra. `http://cassandra.apache.org/`.

[8] H. S. Chang, R. Givan, and E. Chong. On-line Scheduling via Sampling. In *Proc. of Artificial Intelligence Planning and Scheduling (AIPS)*, 2000.

[9] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, 2002.

[10] Linux Containers. `http://lxc.sourceforge.net/`.

[11] C. Delimitrou and C. Kozyrakis. iBench: Quantifying Interference for Datacenter Workloads. In *Proc. of the IEEE Intl. Symp. on Workload Characterization*, 2013.

[12] C. Delimitrou and C. Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proc. of the 18th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2013.

[13] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2014.

[14] X. Dong, Y. Wang, and H. Liao. Scheduling Mixed Real-Time and Non-real-Time Applications in MapReduce Environment. In *Proc. of the IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*, 2011.

[15] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[16] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proc. of the 2nd ACM Symposium on Cloud Computing*, 2011.

[17] J. Hamilton. Cost of Power in Large-Scale Data Centers. `http://perspectives.mvdirona.com`, 2008.

[18] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. 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 Symp. on Networked Systems Design and Implementation*, 2011.

[19] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proc. of the 22nd Symp. on Operating System Principles*, 2009.

[20] K. C. Kiwiel. Convergence and efficiency of subgradient methods for quasiconvex minimization. *Mathematical programming*, 90(1), 2001.

[21] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid. Server engineering insights for large-scale online services. *IEEE Micro*, 30(4), 2010.

[22] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *Proc. of the EuroSys Conf.*, 2014.

[23] J. Mars and L. Tang. Whare-map: heterogeneity in "homogeneous" warehouse-scale computers. In *Proc. of the 40th annual Intl. Symp. on Computer Architecture*, 2013.

[24] McKinsey & Company. Revolutionizing Data Center Efficiency. In *Uptime Institute Symposium*, 2008.

[25] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10), 2001.

[26] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds. In *Proc. of the EuroSys Conf.*, 2010.

[27] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proc. of the USENIX Annual Technical Conference*, 2013.

[28] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proc. of the 24th Symp. on Operating System Principles*, 2013.

[29] G. Park. A generalization of multiple choice balls-into-bins. In *Proc. of the 30th annual ACM SIGACT-SIGOPS symp. on Principles of Distributed Computing*, 2011.

[30] A. Rajaraman and J. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011.

[31] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proc. of the EuroSys Conf.*, 2013.

[32] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. Modeling and Synthesizing Task Placement Constraints in Google Compute Clusters. In *Proc. of the 2nd ACM Symposium on Cloud Computing (SOCC)*, 2011.

[33] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proc. of the 10th USENIX Conference on Operating Systems Design*

and Implementation, OSDI'12, pages 349–362, Berkeley, CA, USA, 2012. USENIX Association.

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

[35] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. of the 4th Symposium on Cloud Computing*, 2013.

[36] VirtualBox. `https://www.virtualbox.org/`.

[37] VMware Virtual Machines. `http://www.vmware.com/`.

[38] I. H. Witten, E. Frank, and G. Holmes. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[39] The Xen Project. `http://www.xen.org/`.

[40] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers. In *Proc. of the 40th annual Intl. Symp. on Computer Architecture*, 2013.

[41] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. of the 2nd conf. on Hot Topics in Cloud Computing*, 2010.

[42] J. Zhang, A. Sivasubramaniam, H. Franke, N. Gautam, Y. Zhang, and S. Nagar. Synthesizing representative I/O workloads for TPC-H. In *Proc. of the 10th IEEE intl. symp. on High Performance Computer Architecture*, 2004.

[43] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI 2: CPU performance isolation for shared compute clusters. In *Proc. of the EuroSys Conf.*, 2013.