Tarcil: Reconciling Scheduling Speed and Quality in Large, Shared Clusters

Abstract

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

We present Tarcil, a distributed scheduler that targets both scheduling speed and quality, making it appropriate for large, highly-loaded clusters running both short and long jobs. Tarcil uses an analytically derived sampling framework that dynamically adjusts the sample size based on load and provides guarantees on the quality of scheduling decisions with respect to resource heterogeneity and workload interference. It also implements admission control when sampling is unlikely to find suitable resources for a task. 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 workload scenarios, Tarcil increases the fraction of tasks that achieve near ideal performance by 4x and 2x compared to sampling-based and centralized scheduling respectively.

1. Introduction

An increasing and diverse set of applications is now hosted in private and public datacenters [4, 10, 18]. The large size of these clusters (up to tens of thousands of servers) and the high arrival rate of jobs (up to millions of tasks per second) make cluster scheduling quite challenging. The cluster scheduler must determine which hardware resources, e.g., specific servers and cores, should be used by each job. Ideally, scheduling decisions lead to three desirable properties. First, each workload receives resources that enable it to achieve *predictably high performance*. Second, jobs are packed tightly on available servers, achieving *high cluster utilization*. Third, decisions introduce *minimal scheduling overheads*, allowing the scheduler to handle 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 [14, 20]. In the second group, decisions are parallelized through two-level, distributed or shared-state designs. Two-level schedulers, such as Mesos and YARN, use a centralized coordinator to divide resources between frameworks like Hadoop and MPI [19, 34]. Each framework uses its own scheduler to assign resources to incoming 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* [13, 17, 24, 26, 39, 42]. For instance, Quasar [14] 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 [20] formulates scheduling as a cost opti-

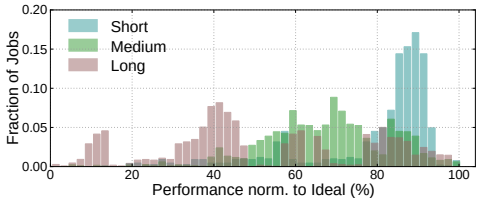


Fig. 1a: Sampling-based scheduling.

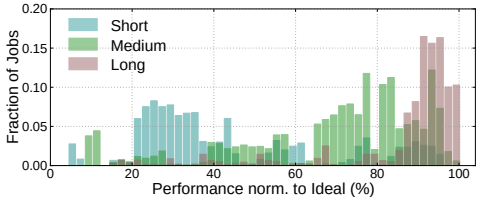


Fig. 1b: Centralized scheduling.

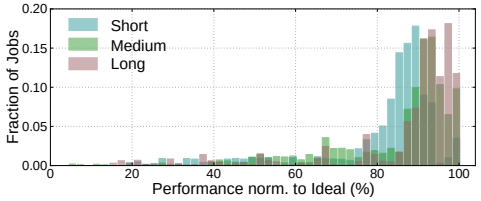


Fig. 1c: Tarcil.

Figure 1: Distribution of job performance on a 200-server cluster with concurrent, sampling-based [28] and centralized greedy [13] schedulers and Tarcil for three scenarios: 1) short, homogeneous Spark [40] tasks (100ms average duration), 2) Spark tasks of medium duration (1s–10s), and 3) long Hadoop analytics tasks (10s–10min). The ideal performance (100%) assumes no scheduling overheads and no performance degradation due to interference. The cluster utilization is 80%.

mization problem that accounts for preferences with respect to locality, fairness and starvation-freedom. Such schedulers make high quality decisions that lead to high application performance and high cluster utilization. Unfortunately, they need to greedily inspect the cluster state on every scheduling event. Their decision overhead can be prohibitively high for large clusters, and in particular for the very short jobs of real-time analytics (100ms - 10s) [28, 40]. Using multiple greedy schedulers improves scheduling throughput but not latency, and terminating the greedy search early typically lowers the 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 heterogeneity and interference preferences of jobs. Typically concurrent schedulers follow sampling schemes, while centralized systems are paired with sophisticated scheduling 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 of variable job duration using Sparrow [28] on a 200-server EC2 cluster. For very short jobs (100ms), fast scheduling allows most workloads to achieve 80% to 95% of the ideal performance on this cluster. In contrast, jobs with medium (1s–10s) or long duration (10s–1min) suffer significant degradation and achieve 50% to 30% of ideal performance. As duration increases, jobs become more heterogeneous in their resource 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 [14]. 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 propose *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. Similar to Quasar [13, 14], Tarcil starts with rich information on the resource preferences and interference sensitivity of incoming jobs. Similar to Sparrow [28], it uses sampling to avoid examining the whole cluster state on every decision. However, there are two key differences 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 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 of 100 and 400 servers on Amazon EC2 to show that Tarcil leads to low scheduling overheads and predictably high performance for a wide range of 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 2-3x faster. For a cluster running a wide range of applications from short Spark tasks to long Hadoop jobs and 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 [14]. Finally, Figure 1c, shows that Tarcil enables close to ideal performance for the vast majority of jobs of the three scenarios.

2. Background

Our work draws from related efforts to improve 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 [19, 31]. We assume a structure similar to Google's Omega [31], where multiple scheduling agents can access the whole cluster state. As long as these agents rarely attempt to assign work to the same servers (infrequent conflicts), they proceed concurrently without additional delays. Section 5 discusses conflict resolution.

Sampling-based scheduling: Based on results from randomized load balancing [25, 29], we can design sampling-based cluster schedulers [8, 15, 28]. Sampling the state of just a few servers reduces the latency of scheduling decisions and the probability of conflicts between concurrent scheduling agents, and is likely to find available resources in lightly- or medium-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 an incoming 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. In our evaluation we compare Tarcil with Sparrow.

Heterogeneity & interference-aware scheduling: Hardware heterogeneity occurs in large clusters because servers are populated and replaced over time [13, 39]. 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 [13, 17, 24, 27]. A scheduler can improve task performance significantly by taking into consideration its resource preferences. For instance, a particular

task may perform much better on 2.3GHz Ivy-Bridge cores compared to 2.6GHz Nehalem cores, while another task may be particularly sensitive to interference from cache-intensive workloads executing on the same server.

The key challenge in heterogeneity and interference-aware scheduling is knowing the preferences of incoming jobs. We start with a system like Quasar that automatically estimates resource preferences and interference sensitivity [13,14]. 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, 21, 30, 37]. Collaborative filtering projects the signal against all information available from previously-run jobs, identifying similarities in resource and interference preferences. These include examples such as the preferred core frequency and cache size for a job or the memory and network contention it generates. Quasar performs profiling and collaborative filtering online. We perform this analysis offline, given that workloads like real-time analytics are repeated multiple times, potentially over different data (e.g., daily or hourly).

3. The Tarcil Scheduler

3.1. Overview

Tarcil is a shared-state scheduler that allows multiple, concurrent agents to operate on the cluster state [31]. 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* [13, 14]. This information provides estimates of the relative performance on the different server platforms, as well as estimates of the interference the workload can tolerate and generate in shared resources (caches, memory, I/O channels). Next, Tarcil performs *admission control*. Given statistics on 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. Tarcil maintains coarse-grained statistics on available resources for admission control decisions. These statistics are updated as jobs begin and end execution.

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. This is useful for long-running workloads; for short jobs, the initial scheduling decision determines performance with little space for adjustments.

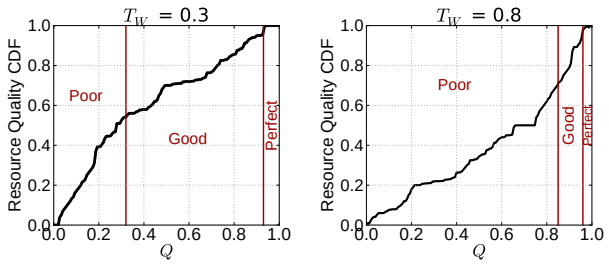


Figure 2: Distribution of resource quality Q for two workloads with $T_W = 0.3$ (left) and $T_W = 0.8$ (right).

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 [11]. Each RU consists of one core and an equally partitioned fraction of the server's memory and storage capacity and the provisioned network bandwidth. For example, a server with 16 cores, 64GB DRAM, 480GB of Flash and a 10GE NIC has 16 RUs, each with 1 core, 4 GB DRAM, 30GB of Flash and 625ME of network bandwidth. Because all our experiments are on public cloud providers where the network topology is unknown, in our evaluation we do not partition network bandwidth.

RU quality: The utility an application can extract from an RU depends on the hardware type (e.g., 2GHz vs 3GHz core) and the interference on shared resources from other jobs on the same server. Classification [13, 14] 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 [12]. Interference preferences capture, first, the amount of pressure t_i a job can tolerate in each shared resource i ($i \in [1,N]$), and second, the amount of pressure c_i it itself will generate in the same 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 in an RU.

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 [24], we reorder the elements of V_W by decreasing value of c_i and get $V'_W = [c_j, c_k, ..., c_n]$, with $c_j > c_k > ... > c_n$. 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 $V_{W_{enc}} = 8431$. The expres-

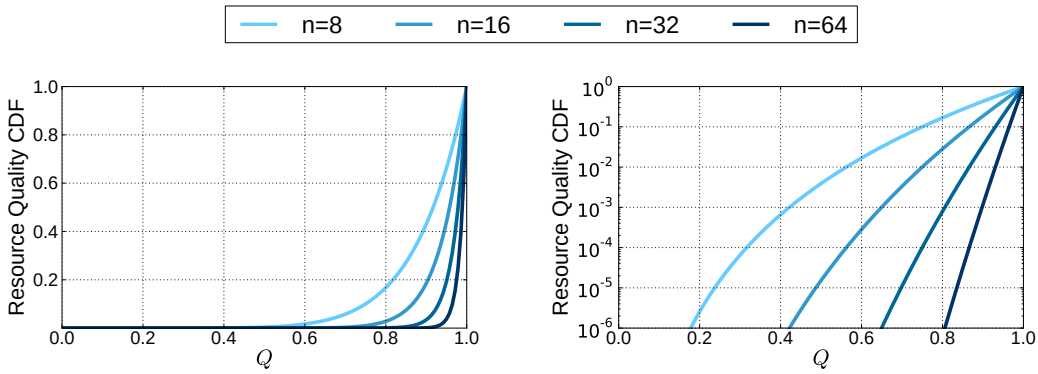


Figure 3: Resource quality CDFs under the uniformity assumption in linear and log scale for sample size R=8, 16, 32 and 64.

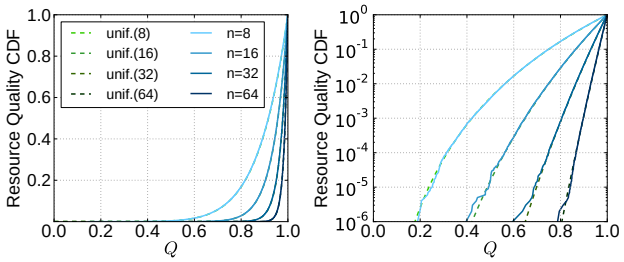


Figure 4: Comparison of resource quality CDFs under the uniformity assumption, and as measured in a 100-server cluster.

sion above is provably the most dense encoding that preserves the full entropy of the values of vector V'_W and their ordering, for general V'_W. Finally, for simplicity we normalize $V_{W_{enc}}$ in $[0,1]$ and derive the target resource quality for job W:

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

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:

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

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:

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

The higher the interference from co-located 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.

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

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

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 one hundred, 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 time. 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 incoming jobs based on T_W.

3.3. Sampling-based 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 cluster load, 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, and distribute them uniformly in $[0,1]$, i.e., for N_{RU} total RUs, $Q(i) = i/(N_{RU}-1)$, $i \in [0, N_{RU}-1]$. Because Q is now a *probability distribution* function of resource quality, 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 Q_i ($Q_1, Q_2, ..., Q_R \sim U[0,1]$) that are *uniformly distributed* by construction and statistically independent from each other (*i.i.d*), we can derive the distribution of quality Q after sampling. The cumulative distribution function (CDF) of the resource quality of each candidate is: $F_{Q_i}(x) = Prob(Q_i \leq x) = x, x \in [0,1]$[1]. Since the candidate with the highest quality

[1] This assumes Q_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.

is selected from the sampled set, its resource quality is the random variable $A = max\{Q_1, Q_2, ..., Q_R\}$, and its CDF is:

$$F_A(x) = Prob(A \leq x) = Prob(Q_1 \leq x \wedge ... \wedge Q_R \leq x)$$
$$= Prob(Q_i \leq x)^R = x^R, x \in [0,1] \quad (6)$$

This implies that the distribution of 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 $Q = 80\%$ ($Prob(Q < 0.8 | \forall\, RU) = 10^{-6}$).

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

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 allocation of 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 as shown in Figure 5b. Alternatively, if a job prefers distributing its resources across machines 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 $Pr[Q < 0.8] = 10^{-3}$ 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. $Pr[Q < 0.92] = 10^{-3}$, 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.

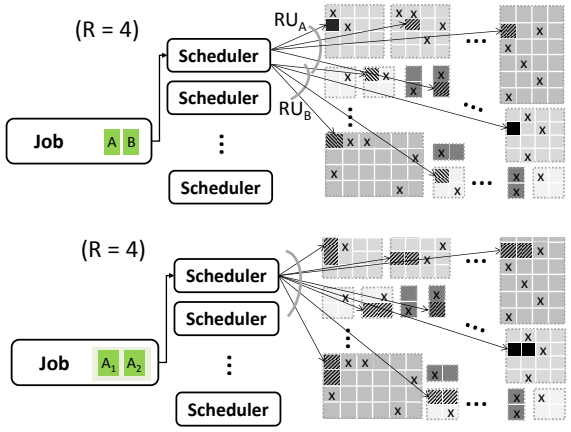


Figure 5: Batch sampling in Tarcil with sample size $R = 4$ for (a) a job with two independent tasks A and B, and (b) a job with two subtasks A_1 and A_2 that exhibit locality. x-marked RUs are already allocated, striped RUs are sampled, and solid black RUs are allocated to the incoming job after sampling.

4. Admission Control

4.1. Pre-scheduling Queueing

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

A simple indication to trigger job queueing is the count of available RUs in the cluster. This, however, does not yield sufficient insight into the quality of available RUs. If most RUs have poor quality for an incoming job, it may be better for it to wait. Unfortunately, a naïve quality check involves accessing the state of the whole cluster, which would introduce 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 for each RU to construct a contention score vector $[C_1\, C_2\, ...\, C_N]$ from the resource contention C_i it experiences in each of its resources, 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 also separate RUs by platform type to account for heterogeneity. We *only* keep a single count of available RUs for each bucket. The hash for an RU (and the counter of the corresponding bucket) needs to be recalculated upon instantiation or completion of a job in an RU. Updating the per-bucket counters is a fast operation, out of the critical path for scheduling. Note that excluding updates

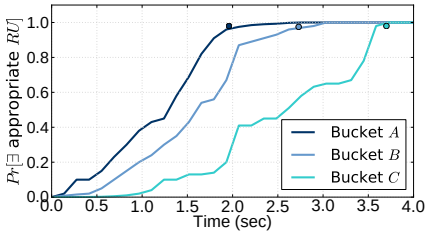


Figure 6: Actual and estimated (dot) probability for a target RU to exist as a function of waiting time for three buckets.

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. Queued applications wait until the probability that resources are freed increases or until an upper bound for waiting time is reached. To estimate waiting 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: Fig. 6 shows the probability that a desired RU will become available within time t for different buckets for a heterogeneous 100-server EC2 cluster running short Spark tasks and longer 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, which closely approximates the measured time for an appropriate RU to be freed (less than 8% deviation on average). In all experiments, we use 20 buckets, and history of the past 2 hours, which was sufficient to make accurate estimations of available resources. The number of buckets and/or history length may vary for different systems.

4.2. Post-scheduling Queueing

A job that exceeds the upper bound on queueing may still require a high sample size. To avoid excessive scheduling overheads, we cap the sample size at 32 and instead use late binding on the sampled servers until resources become available [28]. If the best two of the 32 sampled RUs are currently busy, the job is locally queued in both until the first RU is freed and is subsequently removed from the queue of the second RU. Note that local queueing is unlikely in practice.

5. Tarcil Implementation

5.1. Tarcil Components

Figure 7 shows the components of the scheduler. Tarcil is a distributed, shared-state scheduler and, unlike Quincy or Mesos, it does not have a central coordinator [19, 20]. Scheduling agents work in parallel, are load-balanced by the cluster front-end, and each agent has a local copy of the shared server state, which contains the list and status of all RUs in the cluster.

Since all schedulers have full access to the cluster state, conflicts are possible. Conflicts between agents are resolved using *lock-free optimistic concurrency* as discussed in [31]. The system maintains one resilient master copy of state. Each scheduling agent has a local copy of this state which is updated frequently. When an agent makes a scheduling decision it attempts to update the master copy of the state using an atomic write operation. While an agent performs this action no other agent can update these resources in the master copy. 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 (every 5-10sec) with the master. The timing of the updates includes a small random seed such that not all agents update their state at exactly the same time, making the master 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]. In the event where one 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 quality of RUs. When a new task is assigned to a server by a scheduling agent, the monitor updates the status of the RU in the master copy and notifies the scheduling agent and admission control. Finally, a per-RU *load monitor* evaluates performance in real time. When the monitor detects that a job's performance deviates from its expected target, it notifies the proper agent for a possible allocation adjustment. The load monitor also notifies agents of CPU or memory saturation, which triggers resource autoscaling (see Sec. 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 the network bandwidth. Containers can be merged to accommodate multicore workloads, using *cgroups*. Virtual machines (VMs) can also be

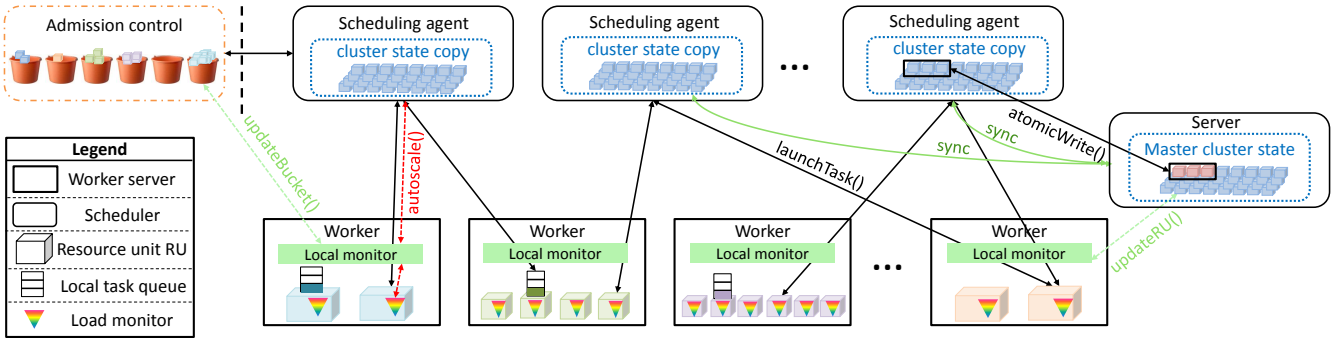


Figure 7: The different components of the scheduler and their interactions.

used to enable workload migration [27, 35, 36, 38], but would incur higher overheads.

Figure 8 traces a scheduling event. Once a job is submitted, admission control evaluates whether it should be queued or not. 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 that should be allocated to the job. The actual selection takes into account the resource quality and platform preferences, as well as any locality preferences of a task. The agent then attempts to update the master copy of the state. Upon a successful commit the agent 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 RPCs that update the master state and notify the agent and admission control; the scheduling agent then informs the cluster front-end.

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 can go through [22]. Similarly, we must consider cases where Tarcil makes a sub-optimal 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*, i.e., a local daemon running in each RU, 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 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.

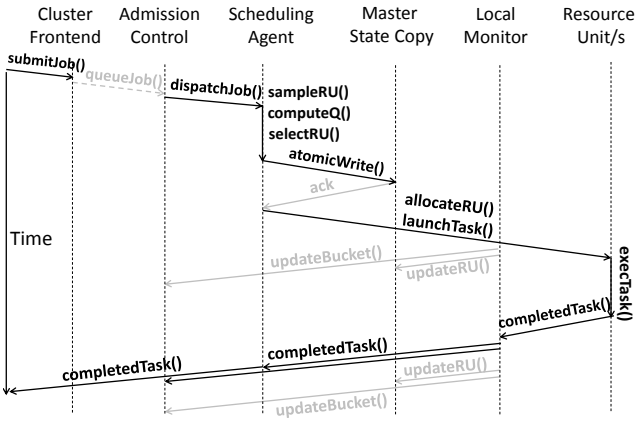


Figure 8: Trace of a scheduling event in Tarcil.

5.3. Fairness

Users can submit jobs with priorities. Jobs with higher priority will bypass others at admission control and preempt lower-priority jobs during resource selection. Tarcil also allows the user 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 from a job is scheduled. We leave the experimental evaluation of priorities and other policies to future work.

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: Fig. 9 shows the sensitivity of sampling overheads and response times to the sample size for homogeneous Spark jobs with 100msec duration and cluster loads varying from 10% to 90% on the 110-server EC2 cluster. All machines are r3.2xlarge memory-optimized instances (61GB of RAM). 10 servers are used by the scheduling agents, and the remaining 100 machines 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 in each boxplot shows the mean. As sample size increases, the overheads increase. Until $R = 32$ overheads are marginal even at

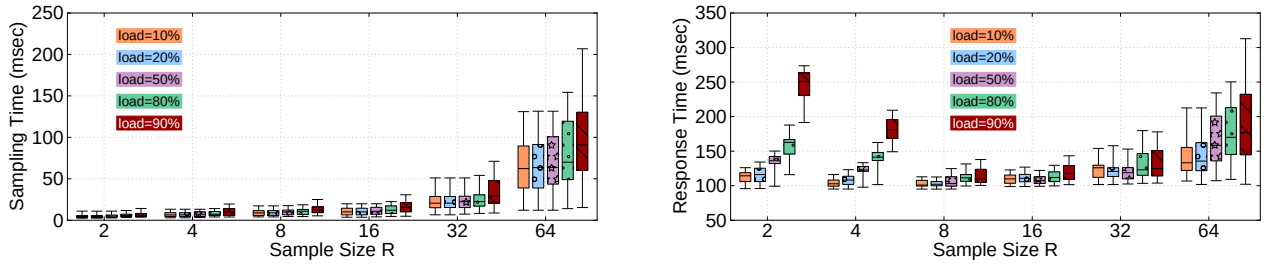


Figure 9: Sensitivity of sampling overheads and response times to sample size.

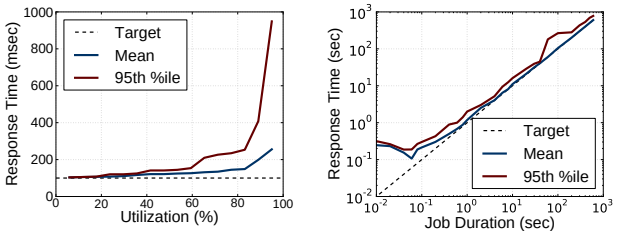


Figure 10: Response times when (a) increasing cluster load, and (b) when decreasing task duration with constant load.

high loads, but they increase substantially for $R \geq 64$, primarily due to the overhead of resolving conflicts between the 10 scheduling agents used. 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. Sample size of $R = 8$ is optimal for both low and high cluster loads, in this scenario.

Cluster load: Fig.10a 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 cluster load. The target performance of 100msec includes no scheduling overheads or degradation due to suboptimal scheduling. The reported response times include the task execution time 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 at high loads, but only exceeds 250msec at cluster loads of 80% or higher. Tasks with very high response times are typically those delayed by admission control until the wait-time threshold is reached. 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: Fig. 10b shows the average and 95th percentile response times as a function of task duration, which 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 or close to 100msec, the scheduling overhead dominates. Despite this, the mean and 95th percentile remain very close, which shows that performance unpredictability is limited. For long jobs, configuring and allocating large amounts of resources dominates the scheduling overheads, while for large numbers of short tasks, queueing delay dominates.

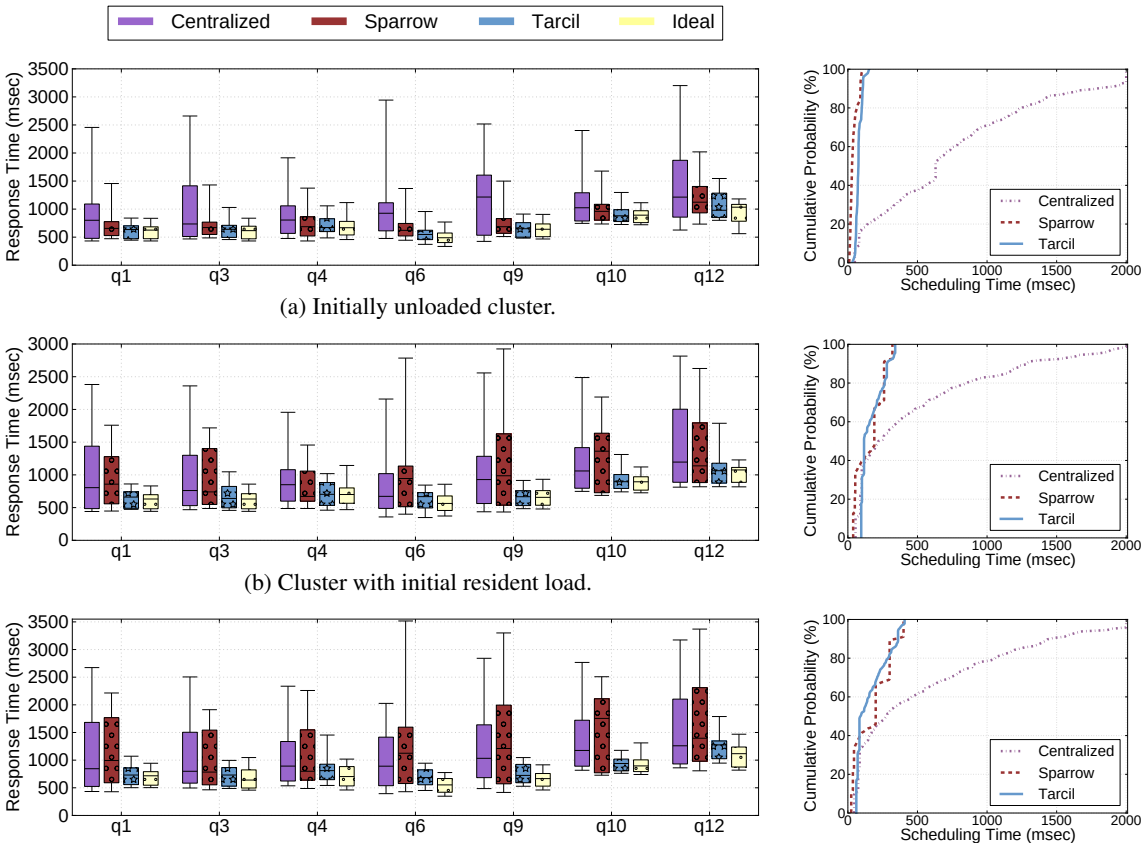
6.2. Comparison with Other Schedulers

Methodology: We compare Tarcil to Sparrow [28] and Quasar [14]. Sparrow uses multiple scheduling agents and 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 scheduling 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 (61GB of RAM). 10 servers are dedicated to the scheduling agents for Tarcil and Sparrow and a single server for Quasar. While we could replicate Quasar's scheduler for fault tolerance, it would not help with the latency of each scheduling decision. Additionally, Quasar schedules applications at job, not task, granularity (when applicable), which reduces its scheduling load. Unless otherwise specified, Tarcil uses sample sizes of $R = 8$ during low load.

6.2.1. TPC-H workload

We compare the three schedulers on the TPC-H decision support 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 [16], 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 present in the cluster. Figure 11a shows the response times for seven representative query types [41].



(a) Initially unloaded cluster.

(b) Cluster with initial resident load.

(c) Heterogeneous cluster with initial resident load.

Figure 11: Response times for different query types (left) and CDFs of scheduling overheads (right).

Response times include all scheduling overheads from sampling or the greedy selection, and queueing. Boundaries show 25th and 75th percentiles and whiskers the 5th and 95th percentiles. The ideal scheduler corresponds to a system that identifies the resources of optimal quality (including heterogeneity and interference preferences) with zero delay. Fig. 11a 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. To verify this, we also show the scheduling time CDF on the right of Fig. 11a. While Tarcil and Sparrow have tight bounds on scheduling overheads, the centralized scheduler adds up to 2 seconds of delay (timeout threshold). Comparing the query performance using 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 11b shows a setup where 40% of the cluster is busy servicing background applications, including other Spark jobs for machine learning processing, long Hadoop workloads, and latency-critical services like memcached. These jobs are *not* being scheduled by the examined schedulers. While the centralized scheduler still adds considerable overhead to each job (Fig.11b, 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 (poor predictability). 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. Fig.11c 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

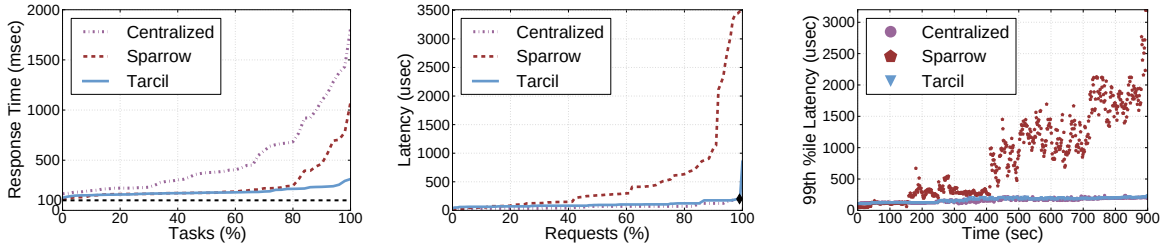


Figure 12: Performance of scheduled Spark tasks and resident memcached load (aggregate and over time).

despite its much higher scheduling overheads. Tarcil preserves response times close to those in the unloaded cluster and very close to those achieved with the ideal scheduler.

6.2.2. Impact on Resident Memcached Load

Finally, we examine the impact of scheduling decisions on resident cluster load. In the same heterogeneous cluster (110 nodes on EC2, 100 workers and 10 schedulers), we place long-running memcached instances as resident load. These instances serve read and write queries following the Facebook `etc` workload characteristics [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 total system capacity and has a QoS target of 200usec for the 99th percentile of response latency.

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. Fig. 12a shows the response times of the Spark tasks for the three schedulers. The centralized scheduler adds significant overheads, while Sparrow and Tarcil lead to 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 the memcached load. Fig.12b shows the latency CDF of the memcached requests. The black diamond depicts the QoS constraint of 200usec for the 99th request percentile. With Tarcil and the centralized scheduler, memcached does not suffer as both schedulers attempt to minimize interference. Sparrow, however, leads to large latency increases for memcached. Even though the performance of the short tasks is satisfactory, not accounting for resource preferences has an impact on the longer jobs in the cluster. Finally, Fig.12c shows how the 99th percentile of memcached requests changes throughout the execution of the experiment. 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 accounts for their resource preferences. Second, even if the long-running resident load was scheduled using Quasar, scheduling short tasks with Sparrow 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 the scheduling decision 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 [33], latency-critical services (memcached [23] and Cassandra [7]), and single-server benchmarks (SPECCPU2006, PARSEC [5], etc.). 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 quality of allocation decisions.

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 (sample size 2, batch sampling and late binding) for the remaining jobs.

Performance: Fig. 13a 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 in this cluster. 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 13b 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.

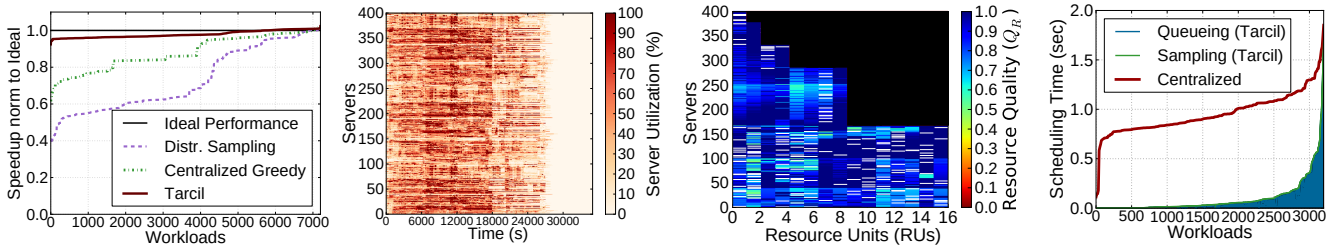


Figure 13: (a) Performance across 7,200 jobs on a 400-server EC2 cluster for the *Sampling-based* and *Centralized* schedulers and *Tarcil*, normalized to optimal performance, (b) cluster utilization achieved by Tarcil throughout the duration of the experiment, (c) quality of resource allocation across all RUs, and (d) scheduling overheads in *Tarcil* and the *Centralized* scheduler.

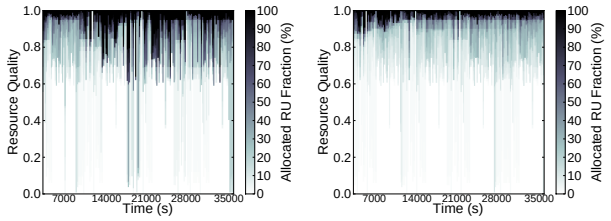


Figure 14: Resource quality CDFs for: (a) *Sampling-based*, (b) *Tarcil*.

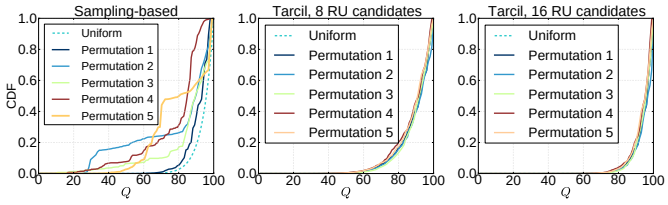


Figure 15: Resource quality distributions for the *Sampling-based* scheduler and *Tarcil* with $R = 8$ and 16 RUs across different permutations of the EC2 scenario.

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 13c 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 8,000s when all applications have arrived and the cluster operates at maximum utilization. White tiles correspond to unallocated resources. Dark blue tiles denote jobs with resources very close to their target quality. Lighter blue RUs correspond to jobs that received good but suboptimal resources. The graph shows that the majority of jobs are given appropriate resources. Note that high Q does not imply low server utilization. Utilization at the time of the snapshot is approximately 75%.

Scheduling overheads: Figure 13d shows the scheduling overheads for the Centralized scheduler and Tarcil. The results 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. The overheads for the *Sampling-based* scheduler are similar to Tarcil and are omitted from the graph for clarity.

Predictability: Figure 14 shows the fraction of allocated RUs that are over a certain resource quality at each point of the duration of the scenario. Results are shown for the *Sampling-based* scheduler (left) and *Tarcil* (right). Darker colors towards the bottom of the graph denote that a larger fraction of allocated RUs have poor quality. At time 16,000sec, 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 15 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 allocates resources of 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 the predictability of scheduling decisions.

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 both 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 excessive sampling and poor scheduling decisions at high load. We have compared Tarcil to existing parallel and centralized schedulers for a variety of workload scenarios on 100- to 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.

References

[1] Alexandr Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Communications of the ACM 51 (1): 117-122*.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of SIGMETRICS*. London, UK, 2012.

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

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

[5] 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.

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

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

[8] Hyeong Soo Chang, Robert Givan, and Edwin Chong. On-line scheduling via sampling. In *Proc. of Artificial Intelligence Planning and Scheduling (AIPS)*. 2000.

[9] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. of the 34th Annual ACM Symposium on Theory of Computing 2002*.

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

[11] Linux containers. `http://lxc.sourceforge.net/`.

[12] 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.

[13] 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.

[14] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proc. of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, USA, 2014.

[15] Xicheng Dong, Ying Wang, and Huaming 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)*. Tainan, 2011.

[16] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. Franklin, S. Shenker, and I. Stoica. Shark: Fast data analysis using coarse-grained distributed memory. In *Proc. of the 2012 ACM SIGMOD*. Scottsdale, Arizona, 2012.

[17] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proc. of the 2nd ACM Symposium on Cloud Computing*, pages 22:1–22:14, 2011.

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

[19] 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.

[20] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, MT, 2009.

[21] 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*.

[22] Christos Kozyrakis, Aman Kansal, Sriram Sankar, and Kushagra Vaid. Server engineering insights for large-scale online services. *IEEE Micro*, 30(4):8–19, July 2010.

[23] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proc. of EuroSys*. Amsterdam, The Netherlands, 2014.

[24] 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.

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

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

[27] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, 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.

[28] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13. Farminton, PA, 2013.

[29] Gahyun Park. A generalization of multiple choice balls-into-bins. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11. San Jose, CA, 2011.

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

[31] 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.

[32] Bikash Sharma, Victor Chudnovsky, Joseph L. Hellerstein, Rasekh Rifaat, and Chita R. Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Proc. of the 2nd ACM Symposium on Cloud Computing (SOCC)*. Cascais, Portugal, 2011.

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

[34] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proc. of the Symposium on Cloud Computing*. Santa Clara, CA, 2013.

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

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

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

[38] The xen project. `http://www.xen.org/`.

[39] 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.

[40] 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.

[41] Jianyong Zhang, Anand Sivasubramaniam, Hubertus Franke, Natarajan Gautam, Yanyong Zhang, and Shailabh Nagar. Synthesizing representative i/o workloads for tpc-h. In *Proc. of the 10th International Symposium on High Performance Computer Architecture (HPCA)*. Madrid, Spain, 2004.

[42] 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.