Batch Sampling: Low Overhead Scheduling for Parallel Jobs

Paper # 34

Abstract

Large-scale data analytics frameworks are shifting towards ever shorter task durations and larger degrees of parallelism to provide low latency. As these frameworks start being used for more interactive queries and to power business decision making applications, they pose a major scaling challenge for cluster schedulers, which will need to handle millions of tasks per second while offering low latency and high availability. In this paper, we show that a decentralized load balancing approach, based on a generalization of the power of two choices, provides a viable alternative to scaling up centralized schedulers, and performs within a few percent of optimal for common workloads. We evaluate this approach through analytical results and an implementation called Sparrow that can dramatically outperform the Mesos cluster scheduler for sub-second tasks.

1 Introduction

Today's clusters are running ever shorter and higher-fanout jobs. Spurred by demand for lower-latency interactive data processing, efforts in research and industry alike have produced frameworks (e.g., Dremel [12], Spark [23], Hadapt [2]) that stripe work across thousands of machines or store data in memory in order to complete jobs in seconds. We expect this trend to continue with a new generation of frameworks targeting sub-second response times. Bringing response times into the 100ms range will enable data in large clusters to be queried by automatic business intelligence applications (e.g., to update product prices based on real-time demand), or to be accessed by user-facing services which run sophisticated parallel computations on a per-query basis, such as language translation and highly personalized search.

Providing sub-second response times for parallel jobs that execute on thousands of machines poses a significant scheduling challenge. Careful placement of a job's tasks is fundamental to achieving good performance: data locality can significantly reduce a task's completion time, while good load balancing is critical to minimize the overall job's response time. Parallel jobs are especially sensitive to scheduling decisions, as a job's performance is determined by the *last* task to complete. Thus, scheduling systems for these workloads will need to make high-quality placement decisions at very high throughput: a cluster containing ten thousand 16-core machines and running 100ms tasks may require well over 1 million scheduling decisions per second. Scheduling decisions must also be fast: for 100ms tasks, scheduling delays above tens of milliseconds are intolerable. Finally, as large-scale data processing frameworks start to power user-facing services, high availability becomes an operating requirement.

One approach to address this problem would be to optimize current state-of-the-art centralized schedulers to support much lower latency jobs. This presents a difficult engineering challenge, as such a scheduler would need to handle two orders of magnitude higher throughput than the fastest general-purpose schedulers (e.g., Mesos [9], Hadoop NextGen [16], SLURM [11]) by making scheduling decisions in as little as 1μs on average. Additionally, achieving high availability would require the replication or recovery of large amounts of state in sub-second time.

In this paper, we explore a radically different approach for scheduling low-latency jobs, inspired by multi-tier web architectures where multiple front ends balance incoming requests across service nodes. These architectures provide excellent scaling and availability, but they are not designed to work for highly *parallel* jobs, where a single delayed task affects the response time of the entire job. To address this challenge, we propose *batch sampling*, a parallel sampling approach that extends the well-known power-of-two-choices method [13]. In a nutshell, to schedule a job containing m parallel tasks, the scheduler probes dm nodes (where $d > 1$), and picks the m least loaded of these nodes to run the job. Since this sampling technique does not need to maintain global state at the scheduler, it can easily be implemented in a decentralized manner by having multiple front ends schedule jobs independently. Decentralization simplifies both scaling and availability, as operators can improve throughput by simply adding more front ends, and clients can respond to failures by redirecting their queries to other front ends.

Batch sampling performs surprisingly well for common scheduling workloads. Using mathematical analysis, we demonstrate that batch sampling places 99% of jobs on idle machines at up to 70% utilization. Simulation results verify the analysis, and demonstrate that batch sampling performs within 1% of an optimal scheduler at up to 60%

utilization. Furthermore, for jobs with no task-level constraints, we show that batch sampling performs significantly better than applying the power of two choices independently to each task. Batch sampling performs increasingly well with the number of cores per machine, making it particularly well suited for today's clusters.

We evaluate batch sampling using a scheduler implementation called Sparrow. We port Spark [23] to run on top of Sparrow in order to compare Sparrow to Mesos, a recent centralized scheduler. Using Spark to run short TPC-H queries, we find that Mesos performs poorly for low-latency jobs; in a 100-node cluster, Sparrow outperforms Mesos at all utilizations. Unlike other schedulers, whose complexity grows with the number of users and the size of the cluster, Sparrow can easily distribute the workload across multiple frontends and consistently provide a low scheduling latency of several milliseconds. Finally, we discuss the limitations of batch sampling, and how Sparrow can co-exist with traditional batch workloads that employ significantly longer tasks.

2 Target Setting

We design a scheduler specifically for scheduling high-fanout, low-latency jobs. We do not attempt to build a general cluster scheduler, and we anticipate that our scheduler will run tasks in a statically or dynamically allocated portion of the cluster that may have been allocated by a more general scheduler.

To meet the needs of emerging high-fanout, low-latency frameworks, a scheduler must support:

Low latency: To ensure that scheduling delay is not a substantial fraction of job completion time, the scheduler must provide *millisecond-scale scheduling delay*.

High throughput: To handle clusters with tens of thousands of nodes (and correspondingly hundreds of thousands of cores), the scheduler must support *millions of task scheduling decisions per second*.

High availability: Cluster operators already go to great lengths to increase the availability of centralized batch schedulers. Low-latency frameworks will be used to power user-facing services, making *high availability an operating requirement*.

In order to provide a highly scalable scheduling mechanism, we do not support certain types of job constraints. We assume that data-processing frameworks will have some constraints, including task-level constraints (e.g., each task needs to be co-resident with input data) and job-level constraints (e.g., all tasks must be placed on machines with GPUs). However, we do not support inter-job constraints (e.g., the tasks for job A should not be run on any of the same servers as the tasks in job B).

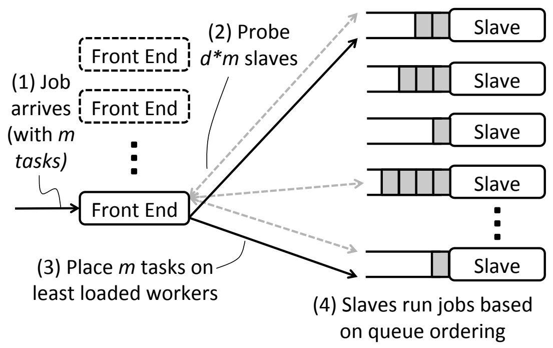
We also assume that tasks have mostly homogeneous length and complete in no more than hundreds of milliseconds. These tasks are allocated to machines that map them to *slots* , and execute them in parallel. The "homogeneous length" assumption allow us to reasonable estimate when a currently busy machine will be able to launch a new task. While this assumption holds for emerging low-latency frameworks, as shown in §7.2, we also discuss how to remove this assumption in §8.3.

This setting applies for the data processing frameworks we have considered. In Spark, for example, a front end compiles functional query definitions into multi-phase parallel jobs, where the first phase is constrained to execute on machines that contain partitions of the cached input data set (task-level constraints) and the remaining phases can execute anywhere. Spark does not have inter-job constraints. Since tasks operate on in-memory data sets with fixed-sized partitions, they often complete in tens to hundreds of milliseconds, which is homogeneous enough to make our sampling technique effective (as shown in the evaluation in §7). Because the tasks are small, their marginal resource requirements tend to be small, meaning that slots provide an adequate isolation mechanism on worker nodes. Future low-latency frameworks are likely to share these characteristics: tasks will consistently complete in tens to hundreds of milliseconds, and inter-job constraints will not be necessary. Our target setting fits well with these types of applications.

We do not consider stragglers, i.e., tasks that take considerably more time than expected. Instead, our focus is on task scheduling to minimize the job response times given the *expected* waiting and completion times of each task. In contrast, straggler mitigation techniques, such as task speculation [24, 4, 8], deal with the *nondeterministic* variation in task execution due to unpredictable causes (e.g., resource contention and failures). In §5.5, we show that our scheduling technique significantly improves the response time of a job even in a system that implements an idealized speculation algorithm. Thus, the scheduling technique we present in this paper is orthogonal and complementary to the straggler mitigation ones.

3 Sample-Based Scheduling

In order to support the scheduling needs of emerging low-latency frameworks, we investigate a scheduling architecture that relies on a stateless, randomized probing approach to placing tasks. Because our approach does not rely on shared state, it can be distributed over a large number of front ends that place tasks in parallel; such a distributed architecture dramatically improves throughput and availability.



(1) Job arrives (with *m* tasks)
Front End
Front End
Front End
(2) Probe *d*m* slaves
Slave
Slave
Slave
Slave
Slave
(3) Place *m* tasks on least loaded workers
(4) Slaves run jobs based on queue ordering

Figure 1: Placing the parallel tasks in a job using random sampling.

3.1 Batch Sampling

We propose scheduling using stateless front ends that obtain instantaneous load information by performing random sampling before placing a job's tasks. Our approach, which we call *batch sampling*, generalizes Mitzenmacher's Power-of-Two-Choices load balancing technique [13]. When a job arrives with m tasks, a front-end first randomly selects a set of dm candidate nodes ($d \geq 1$), where d is a tunable parameter. The front end then samples the dm slave machines in parallel to instantaneously estimate the completion time of a task on the particular slave.[1] Once all machines have replied, the front end chooses the set of machines which collectively offer the lowest response time, and launches tasks on then. This process is illustrated in Figure 1.

Front ends assign tasks to machines, even though those machines may not be idle. This strategy pushes the responsibility for queueing tasks onto the slave nodes. We assume each slave node will run at most c concurrent tasks (c is typically set to be the number of cores on the machine), and will queue remaining tasks.

3.2 Handling Placement Constraints

We support jobs with two types of constraints. First, we support job-level constraints; for example, tasks are restricted to run on certain types of machines. For these jobs, the scheduler can directly use batch sampling.

Our mechanism also supports task-level constraints. Consider a typical first phase of a job, where each task needs to be co-located with a particular chunk of input data that is replicated on 3 machines. In this case, the scheduler performs *per-task sampling*: it chooses an average of d machines to sample for each task, selected from the 3 possible machines where the task can run, and places the task on the least loaded machine. Assuming 3 replicas, for $d = 3$, our distributed schedulers perform nearly identically to a centralized scheduler, since the probes reveal load information about all possible machines on which the task can run.

In general, for constrained jobs, our approach closely matches the behavior of a centralized scheduler. Constraints limit the set of machines on which the job can run, which means that our load sample, of size dn, represents a sizable proportion of the machines that could run the job. Thus, for constrained jobs, sampling gives the scheduler a near-complete view of the load on the machines where the job could run, allowing the distributed schedulers to make a near-optimal scheduling decision.

3.3 Resource Allocation Policies

Cluster schedulers seek to allocate resources according to a specific policy when aggregate demand for resources exceeds capacity. Popular cluster schedulers such as Hadoop [22] and Dryad [10] opt for *weighted fair sharing* or proportional sharing, which allocates each user a share of the cluster proportional to her weight. When a user is not using her share, it is distributed evenly amongst other users. Proportional sharing has global semantic, applying to the aggregate resource usage across a cluster.

Despite the lack of a central vantage point from which to enforce shares, our design can still provide proportional fairness by pushing the share enforcement to slaves. Each slave performs weighted fair queueing at user level granularity using a separate queue for each user. As a result, two users competing on the same slave will get shares that are roughly proportional to their weights. By extension, two unconstrained jobs or two jobs that share the same constraints (i.e., can run their tasks on the same set of machines) will get their shares on the aggregate resources they use. The fair sharing mechanism provides naturally isolation, as it ensures a user will get its share irrespective of the demand of other users at a node level.

When sending a probe, the scheduler includes the user whose job the probe corresponds to in the message to the slave. The slave responds with the estimated number of tasks that will be run before a task for the given user, and the scheduler greedily places tasks on the machine where they are expected to complete soonest. Figure 2 depicts a simple example, where all users have equal weights and a master is attempting to schedule a task for user c.

In addition, our design can easily provide a strict priority service by maintaining per-user queues at each slave, and scheduling the tasks according to the users' priorities.

[1]Network delay may lead to some staleness in the information acquired through sampling. To mitigate this problem, we use temporary reservations: slave machines account for probes received in the last network RTT when replying to a probe.

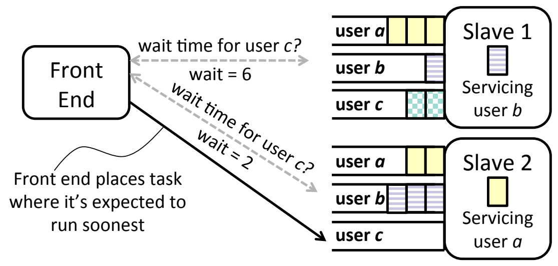


Figure 2: Task placement using a weighted fairness policy.

n	Number of servers in the cluster
λ	Load
m	Tasks per Job
d	Probes per Task
t	Mean task service time

Table 1: Summary of notation.

4 Analysis

This section presents analytical intuition for the performance of parallel sampling in comparison with other techniques. Scheduling parallel jobs is difficult, because the response time of a job is determined by the worst response time of its constituent tasks. This makes techniques which attempt to minimize *mean* task response time generally ineffective, particularly for larger jobs. We quantify the expected job-level performance time under (i) purely random approaches, (ii) direct extensions of existing sample-based techniques and (iii) a new sampling technique called batch sampling.

4.1 Model

We consider a model where jobs arrive in the system as a Poisson process. Each job consists of m tasks that are allocated to servers (see Table 1 for a summary of notation). The mean job arrival rate is $\frac{\lambda n}{mt}$, where n represents the number of servers in the cluster, t denotes the mean service time of a task, and we refer to $\lambda < 1$ as the load. Servers run a single task at a time, and queue remaining tasks. We assume zero network delay (an assuption we revisit in §6) and consider a system with an infinitely large number n of servers (an assumption we validate in §4.3).

4.2 Probability of Zero Wait Time

To provide intuition for the performance of different load balancing approaches, we begin by considering the probability that a job experiences zero wait time, which happens when all tasks in the job are placed on idle servers. Since an ideal, omniscient scheduler can always place tasks on idle machines when the cluster is under 100% utilized, quantifying how often our approach places jobs on all idle machines describes how far we are from optimal. The probability of experiencing zero wait time using random placement, per-task sampling, and batch sampling is summarized in Table 2, graphed in Figure 3, and described below.

Random Placement	$(1-\lambda)^m$
Per-Task Sampling	$(1-\lambda^d)^m$
Batch Sampling Exact Probability Lower Bound	 $\sum_{i=m}^{d\cdot m}(1-\lambda)^i\lambda^{d\cdot m-i}\binom{d\cdot m}{i}$ $1-e^{-\left((1-\lambda)d+\frac{1}{(1-\lambda)d}-2\right)m/2}$

Table 2: Probability that a job will experience zero wait time under three different task placement approaches.

One insight we exploit in this analysis is that the fraction of non-idle machines in the cluster will be equal to the load λ on the cluster. For $\lambda < 1$, basic queueing theory results dictate that the cluster will not experience infinite queueing, so at steady state, the rate at which tasks are processed must equal the rate at which they are arriving. Thus, we expect λn machines to be running one task (with potentially additional tasks queued), and the remaining $(1-\lambda)n$ machines to be idle.

When tasks are randomly assigned to machines, the probability that all m tasks in a job are assigned to idle machines is $(1-\lambda)^m$. Figure 3 (top) plots this probability for a variety of job sizes, and demonstrates that this probability is quite small, even for modest job sizes and low loads.

Previous work has demonstrated that for scheduling tasks, choosing the less loaded of two randomly sampled queues yields drastically improved performance over random placement [13]. We use this technique to schedule parallel jobs by sampling for each task independently, which we call per-task sampling. The probability that all tasks will be placed on idle machines is shown in Table 2; note that with $d = 1$, this reduces to the random case given above.

Increasing the amount of sampling improves results significantly, as shown in Figure 3, which compares sampling two tasks per machine to random placement. When a job's tasks are constrained to run on a small number of nodes, per-task sampling closely approximates optimal performance, because even an ideal is constrained to only a few choices for each task. Indeed, the "optimal" allocation is much worse for highly constrained jobs than for unconstrained ones.

Per-task sampling remains exponentially bad with respect to m; when jobs have no task-level constraints, we can dramatically improve on this performance by employing *batch sampling*. As described in §3.1, batch sampling samples a single batch of dm machines and places a job's m tasks on the least loaded machines in the batch. Table 2

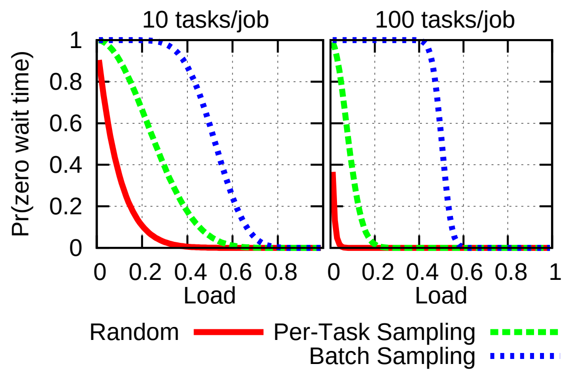


Figure 3: Probability that a job will experience zero wait time in a single-core environment, using random placement, sampling 2 servers/task, and sampling $2m$ machines to place an m-task job.

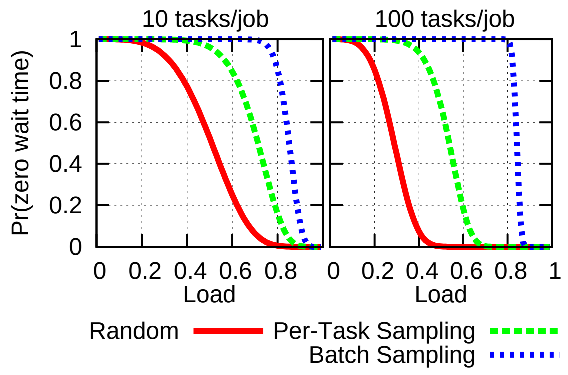


Figure 4: Probability that a job will experience zero wait time in a system of 4-core servers, using both per-task and batch sampling.

describes the exact probability of placing all tasks on idle machines using this approach. As shown in Figure 3, the improvement from using batch sampling increases with the number of tasks per job.

Critically, this bound *improves* exponentially with higher numbers of tasks per job, whereas using per-task sampling, has the opposite property, as demonstrated by the results for multiple job sizes shown in Figure 3. This improvement stems from the fact that the sample size increases with the job size, so the variance in the fraction of sampled machines that are idle correspondingly decreases. Note that the probability for batch sampling is an upper bound, where as the probabilities for the other two schemes are exact results.

We also use a Chernoff tail bound ([7, 15]) on the number of idle machines expected in each batch of probes to obtain a lower bound on the probability that a job will be placed on entirely idle machines; the result is given in Table 2 and derived in detail in our technical report [3]. The closed-form expression for the lower bound is quite helpful for understanding performance for different probing ratios (d) and for different numbers of jobs per job (m). In particular, the bound allows us to determine how to set d for different job sizes in order to get the same expected performance across a variety of job sizes.

4.2.1 Multicore

Operating in multicore environments, where multiple tasks can run concurrently on a single machine, dramatically improves the performance of our approach (keeping cluster load fixed). Consider a model where each server can run up to c tasks concurrently, without degraded performance compared to running the task on an idle machine, and any remaining tasks are queued. As before, the scheduler places at most 1 task per job per machine, even if multiple cores are available. The dramatic multicore improvement (shown in Figure 4) stems from the fact that each probe returns information about c processing units, improving the chance of finding an idle processing unit on which a task can run.

To analyze the performance in multicore environments, we assume that the distribution of idle cores is independent of whether cores reside on the same machine. We validated this assumption using our simulation and found that this assumption gives an accurate indicator of the percentage of tasks with zero queuing, as shown in detail in our technical report [3]. Based on that assumption, the probability that a given machine does not contain at least one empty core is λ^c. Replacing λ in Table 2 with λ^c leads to the updated probabilities depicted in Figure 4. These results improve dramatically on the single-core results: for batch sampling with just 4-cores per machine and 10 tasks per job, batch sampling achieves near perfect performance (99% of jobs are expected to experience zero wait time) at up to 70% load. As the number of tasks per job increases, batch sampling achievers near perfect performance for even higher loads.

4.3 Expected Wait Times

In addition to computing the probability that a job experiences zero wait time, we have also derived results for the expected wait time under each scheduling scheme [3]. We extended an existing queueing analysis of the power of two choices [13], which models the distribution of queue lengths in an infinitely large cluster using differential equations. Due to space limitations we omit the analysis in this paper, but Figure 5 shows the results from numerically solving the system, and compares them to simulation results for a 2500-node cluster. Both plots are for a job composed of 50 tasks; each task has exponentially distributed length with mean t. The analytically derived wait times closely match simulated wait times at all loads.

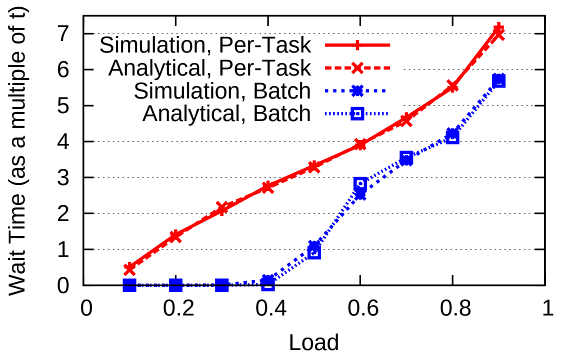


Figure 5: Comparison of analytically computed wait times and simulated wait times for 50-task jobs, using both per-task sampling and batch sampling. Both sampling approaches sample 2 tasks per job ($d = 2$). Wait time is given as a multiple of t, (mean task run time).

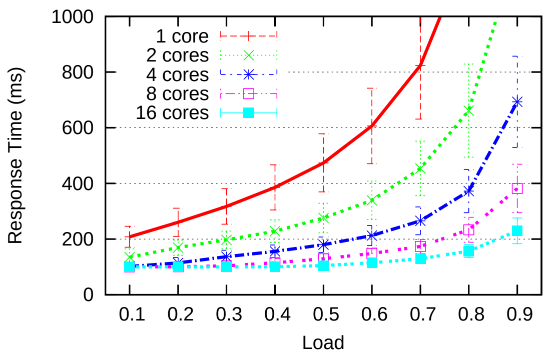


Figure 7: Response time in a $10,000$-core cluster using random placement improves substantially as the cores per machine increase.

5 Simulation Results

In order to understand system behavior under a wide variety of parameterizations and in a more realistic environment, we built an event-based simulator. Response times given by our simulator match those in our deployment, as shown in §7.1. The simulation also models fairness, which is not captured by our analytical results.

Simulated results demonstrate that batch sampling provides near-optimal response times: response times using batch sampling are within 1% of an optimal scheduler at up to 60% load. Batch sampling performs increasingly well as the number of cores per machine increases, which bodes well given current trends towards many-core machines. Simulated results also demonstrate that decentralized fair sharing provides isolation between users.

Unless otherwise specified, the simulations use a mean task duration of 100ms, network delay of 2ms (consistent with delay measured in our deployment), and run for 100 seconds of simulated time. We use 100-task jobs (we find that job size does not have a significant impact on response time).

5.1 Impact of Probing Ratio (*d*)

The number of probes needed to ensure low response times is the key parameter of our system. Figure 6(a) shows the impact of the probing ratio d on job response time, for a 10,000 core cluster (2500 4-core machines) running 100-task jobs consisting of 100ms fixed duration tasks. The "global" line shows response time with an ideal scheduler that picks the 100 nodes with the smallest queues in the *entire* system to run each job.

As predicted by the analytical results in §4.2, response time with two probes/task increases from near perfect (100ms) to 200ms at approximately 80% load, which corresponds to the point at which the probability of experiencing zero wait time decreases from 1 to 0.

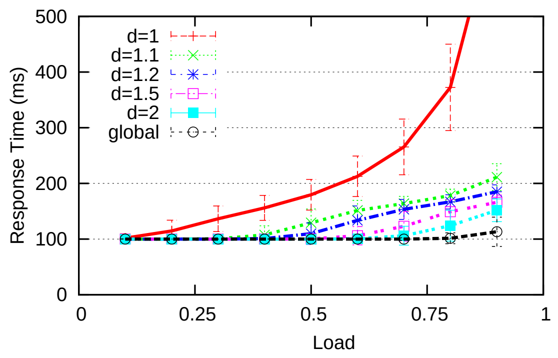
Simulation results demonstrate that modest probing achieves significant gains: even $d = 1.1$ leads to dramatically improved response time. Response time improves with increased probing, with diminishing marginal returns. For $d = 2$, response time using batch sampling is with 1% of the response time provided by the central scheduler at up to 60% load, and remains within 55% of optimal at up to 90% load.

Figure 6(b) plots the response times for 100-task jobs, where the duration of the tasks is exponentially distributed with the mean of 100ms. Baseline response times are significantly longer, because the expected completion time for 100 tasks with exponentially distributed task lengths (running on an empty cluster) is approximately $100 \log(100) = 460$ms. However, in spite of the dramatically different distribution of task lengths, the overall trend is the same: modest sampling leads to significant reduction in expected response time. Batch sampling leads to similar improvements for task lengths that follow a Pareto distribution.

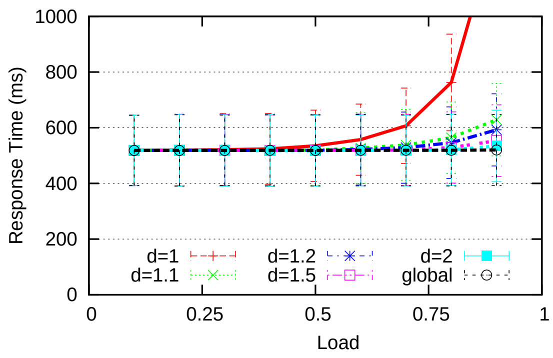
Given that results are similar for the different task length distributions and the fact that we expect task durations to be relatively homogenous for realistic workloads, the remainder of this section presents results for fixed-duration tasks.

5.2 Benefit of Operating on Multiple Cores

As demonstrated in 4, batch sampling performs dramatically better in environments where multiple tasks can run concurrently on each machine. The natural hierarchy enabled by on-chip parallelism has two benefits: first, each probe gains more implicit information, because it describes load on multiple processing units; and second, unpredictability in task run times is amortized over the set



(a)



(b)

Figure 6: Mean response times and standard deviations for 100-task jobs running on a 10,000 node cluster for different probing ratios (*d*): (a) 100ms fixed duration tasks, (b) tasks with exponentially distributed durations with mean of 100ms.

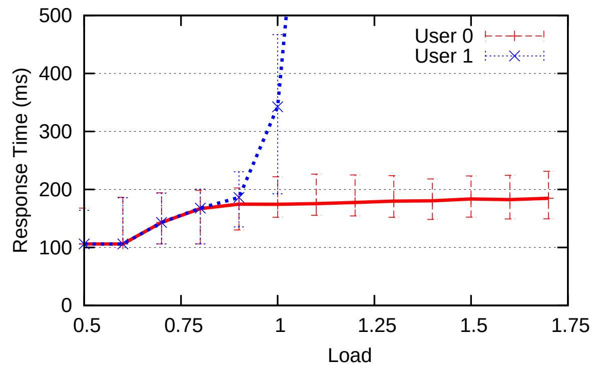


Figure 8: User 1 increases his task arrival rate while User 0 maintains a constant arrival rate of half of her fair share. User 0 continues to receive good response time, even as User 1's experiences infinite queueing.

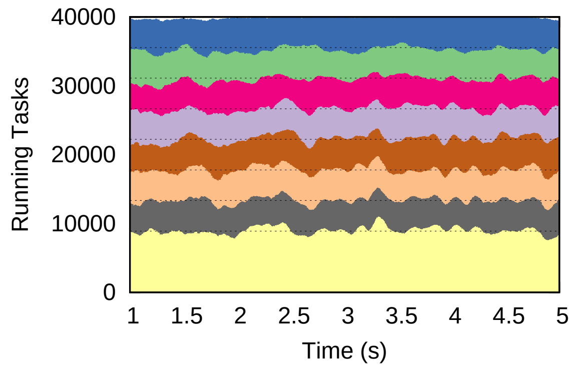


Figure 9: Tasks running for each of eight users over a 4-second period in a 40,000-core cluster (10,000 4-core servers) at 100% load. Dashed lines reflect the assigned shares to each user (user a, shown on the bottom, is assigned twice the weight of all other users).

of concurrently running tasks. The simulation employs a simplified model where each machine has c cores, and can run at most c concurrent tasks; remaining tasks are queued.

Simulation results confirm that response time improves dramatically with the number of cores per machine (keeping the task arrival rate and total cluster processing power fixed), as shown in Figure 7. The remainder of the simulation results use 4 cores per machine; these results will be even better for future clusters that use increasingly large numbers of cores per machine.

5.3 Isolation And Weighted Fair Sharing

Batch sampling performs well until the cluster nears 100% load, prompting the question of whether one user operating at below her fair share will experience poor performance when aggregate cluster load is high. Figure 8 depicts response times in this situation, using weighted fair sharing to achieve isolation on 4-core servers in a 1250-server cluster. User 0 maintains a constant job arrival rate at half of her fair share, while User 1 increases his load to well beyond his fair share. User 0's response time remains within a constant factor of optimal, even as User 1 experiences infinite queuing. Isolation improves as the number of cores per machine increases, since User 0 may need to wait for a task to complete before it can run (assuming no preemption).

Even though weighted fair sharing is performed independently on distributed worker nodes, it still achieves the correct aggregate fair shares, as shown in Figure 9. Figure 9 depicts the number of running tasks over time for each user in a 40,000 node cluster at 100% load, which hover at the fair shares (shares are denoted by dashed lines).

5.4 Constraints

This section illustrates the impact of job-level constraints on response time by modeling a scenario in which some jobs can run only on certain machines, e.g., machines with GPUs, public IP addresses, high network capacity, large

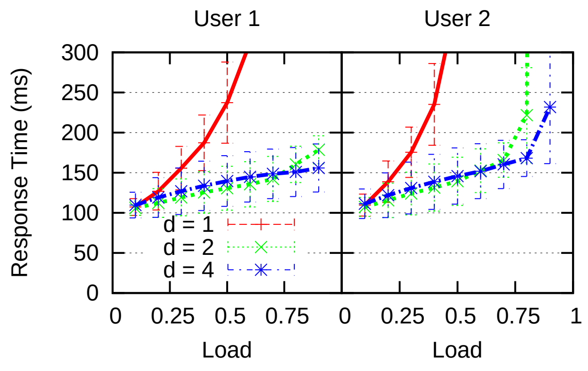


Figure 10: Mean response times for two users for different values of probing ratio, d. Jobs of user 1 can use all machines in the cluster, while the jobs of user 2 can use only half.

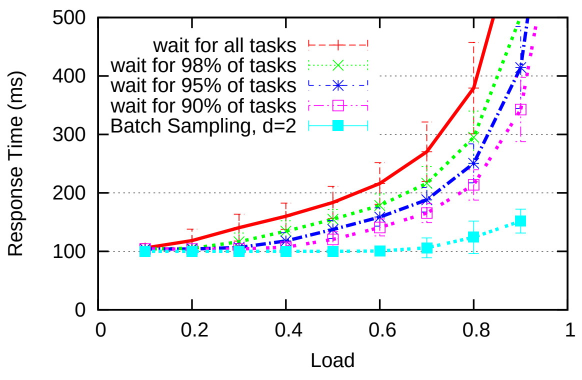


Figure 11: Improving the response time after dropping a fraction of the slowest tasks ($d = 1$). Note that this is an upperbound of speculating the same fraction of slow tasks.

memory, or machines that run a certain version of the OS. According to recent results, these constraints are quite prevalent in practice, with over 50% of the jobs at Google exhibiting such constraints [18].

We consider two users, where all jobs of user 1 can use all machines in the cluster, while all jobs of user 2 can use only *half* of the machines in the cluster. Each user generates the same workload, i.e., same number of jobs, and all jobs have 100 tasks. Figure 9 shows the response times of both users for various values of d. There are two points worth noting. First, the performance of user 1 is worse than that of user 2. This is simply because, user 1 may end up running some of its tasks on the machines that only user 2 can use. This leads to overloading these machines, which will significantly hurt the response times of user 2. In fact, for high loads, the load on user 2's machines exceeds their capacity. Second, as d increases, user 2's performance increases dramatically, as user 1 has more and more choices to avoid the machines used by user 2. In the process, user 1's performance also increases. Note that asymptotically, as the probing ratio increases user 1 will be able to avoid user 2's machines, and the performance of the two users converge (e.g., see response times of both users for $d = 1$ and load lower than 0.8).

5.5 Ignoring Slowest Tasks and Task Speculation

A variety of reactive techniques have been proposed for mitigating the problem of high tail response times in parallel jobs. These techniques include ignoring the slowest tasks in a job, and task speculation which preemptively launches copies of slow tasks [24, 4, 8]. These two techniques are related in that ignoring the slowest $p\%$ tasks of a job provides an upper bound on the job response time when speculating at most $p\%$ tasks. Thus, to bound the performance of speculation, we measure job response time after dropping the slowest 2%, 5% and 10% of tasks. Again, speculating on the slowest respective percentage of tasks would result in response times at least this high. As demonstrated in Figure 11, while these techniques can improve upon random performance, still significantly underperform our sampling approach.

6 Implementation

Sparrow is a distributed scheduler implementation that exploits batch sampling to perform low latency task placement. Sparrow operates as a scheduling service in a computing cluster, and performs task scheduling for one or more concurrently operating frameworks. Frameworks are composed of long-lived *frontend* and *worker* processes, a model employed by many systems. Front ends accept high level queries or job specifications from exogenous sources (e.g. a data analyst, web service, business application, etc.) and compile them into parallel tasks for execution on workers. Front ends are typically distributed over multiple machines to provide high performance and availability. Worker processes are responsible for executing tasks, and are long-lived to avoid startup overhead for tasks such as shipping binaries or bringing large datasets into cache. Front ends submit sets of tasks to the Sparrow, which assigns the tasks to worker nodes. Note that worker processes for multiple frameworks may run co-resident on a single machine; the scheduler assigns resources between the frameworks.

Sparrow is deployed through a set of *Sparrow daemons*, lightweight processes which run on every cluster machine and expose a Thrift service [1] to allow frameworks to launch tasks. Thrift is cross-platform, facilitating a variety of client libraries. Sparrow daemons are responsible for accepting scheduling requests from frameworks, calculating the optimal task placement, and launching those

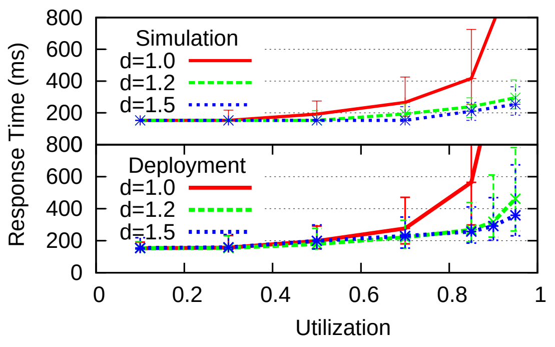


Figure 12: Simulation and deployment, for a 100-node cluster running jobs composed of ten 150ms tasks.

requests via Sparrow daemons on other machines. Sparrow daemons are collectively symmetric: each Sparrow daemon can function as a master, slave, or both, and tasks can be launched from any daemon.

We have ported Spark [23] to Sparrow by writing a Spark scheduling model which calls on the Java Sparrow client. This interface is 106 lines of Scala code.

Sparrow's deployment spurred a few practical changes to the base sampling techniques described in § 3. As in many other high fanout systems, response latencies to probes are long-tailed. While 95% of probes complete within two milliseconds, the worst 1% take 7 milliseconds or more. Waiting for the last probe to return has major consequences: with each passing millisecond the data encoded in existing probes becomes less useful. We currently handle this problem with an absolute timeout: after five milliseconds, we then place the job's tasks based on only the probes that have responded. In the rare case when not enough probes have returned after five milliseconds, we randomly place tasks; however, this situation never arose in our experiments.

To avoid a case where two front ends probe in short succession, only to both believe a machine is free and launch there, we use a temporary (5ms) reservation on the worker node. Since this is much less than the typical arrival rate of tasks on a machine, so false positives are rare.

7 Evaluation

We evaluate Sparrow running on a deployed cluster of 100 EC2 extra-large (quad-core) nodes. We examine the impact of the probing ratio, d, in a real cluster, and find that performance matches results predicted by the simulation (§7.1). We demonstrate that Sparrow provides low-latency response times for short TPC-H benchmark queries, performing significantly better than random (§7.2). Finally, we compare Sparrow to Mesos, and demonstrate that Sparrow provides dramatically better response time for sub-second queries (§7.3).

7.1 Impact of Probing Ratio (d)

We measure the impact of the probing ratio, d, in the deployment, and find that deployed performance matches simulated performance. Figure 12 depicts results for both the real and simulated systems, in a cluster of 100 quad core machines. 10 servers serve as front ends and launch tasks; the remaining 90 servers are slaves. Each job consists of 10 CPU-intensive tasks that run for approximately 150ms in the deployment and exactly 150ms in the simulation. The simulation uses a fixed network round trip time of two milliseconds, which exceeds the mean RTT seen in the deployment. As shown In Figure 12, the deployment sees improved performance with increasingly large d; even small values of d vastly outperform random at high utilization. The variance is somewhat larger for the real deployment, due to variability in the actual service times of the benchmark tasks.

7.2 Performance on a Realistic Workload

To evaluate Sparrow on a realistic workload, we measure Sparrow's performance scheduling tasks for Spark, a distributed in-memory analytics framework. Our workload consists of queries borrowed from the TPC-H benchmark, a popular benchmark for OLAP database workloads, rewritten using Spark's replicated data storage and transformation abstractions. While Sparrow's approach is not limited to scheduling for analytics platforms, large-scale data analytics are one class of applications clearly desiring low response time.

The experiment features ten Spark front ends, which each independently run queries using a unique TPC-H dataset (each data set has a scale factor of 1). Each TPC-H dataset is cached in memory using a replication factor of 3 and striped across 100 shared worker machines. To achieve low response time, we de-normalized the TPC-H-data, a technique used in existing applications such as Dremel [12]. Once translated to Spark, most TPC-H queries result in an distributed read, a filtering step, zero or more grouping operations, and potentially subsequent aggregation. These are each compiled into a Spark *phase*, or a set of parallel tasks for which Sparrow receives a launch request. The first phase has placement constraints corresponding to cached partitions of the database input table. Because most queries follow the same general format, we use just three queries, with 1, 2, and 3 phases, respectively.

Figure 13 depicts end-to-end response time of a TPC-H query as system load increases, using both Sparrow and random placement to place tasks. Each front end repeatedly launches the same query, varying the queries launched per second to increase load. Error bars plot the

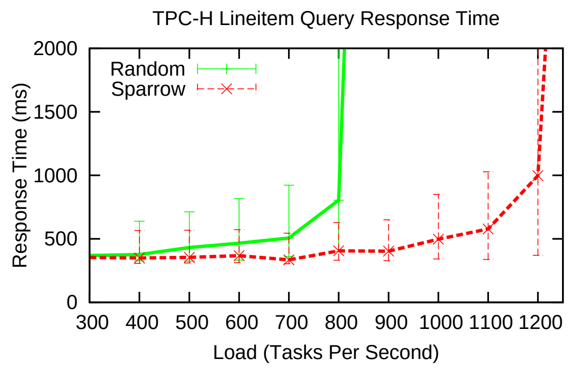


Figure 13: Response time as a function of system load, using random placement and Sparrow scheduling. At 1200 tasks per second, the system is at full utilization.

5th and 95th percentiles. Sparrow outperforms random at all load levels. At just above 60% utilization (corresponding to an arrival rate of 800 tasks per second), randomized placement becomes unstable, leading to infinite queueing. This is caused by hot-spots which naturally occur when data (in this case Spark's cache partitions) is distributed using random hashing (the standard technique for distributing data in today's data stores). If a scheduling layer does not react disproportionately to busy nodes, they quickly bottleneck the progress of the cluster.

To demonstrate the effects of poor (or good) task placement on query response time, Figure 14 depicts the number of tasks on each machine (bottom) and response time (top) over an 80-second period for the TPC-H query, using both random placement (left) and Sparrow (right) to place tasks. These measurements are taken when the system is at 55% utilization, which is near the "tipping point" for the randomized approach. Under random placement, a handful of machines become persistently over-subscribed, taking on more than 4 tasks and experiencing queueing. Sparrow's sampling avoids this uneven placement. Since input partitions are replicated, Sparrow's sampling approach allows it to avoid workload skew by shifting away from more highly contended machines. As a result, Sparrow achieves consistently low response times at this utilization. In fact, the vast majority of queries respond in under 1 second when using Sparrow, up until 1200 tasks per second, at which time the cluster is almost 100% utilized.

7.3 Comparing Sparrow to Existing Schedulers

Today, many cluster frameworks rely on centralized task schedulers. Centralization is attractive because it presents a vantage point which has complete load information about the cluster. However, this information comes at a cost: localizing all decisions at a single point introduces natural scalability limitations. To understand this cost, we compare Sparrow to Mesos [9], a widely used cluster scheduler optimized to provide high scheduling throughput, for scheduling low-latency tasks.

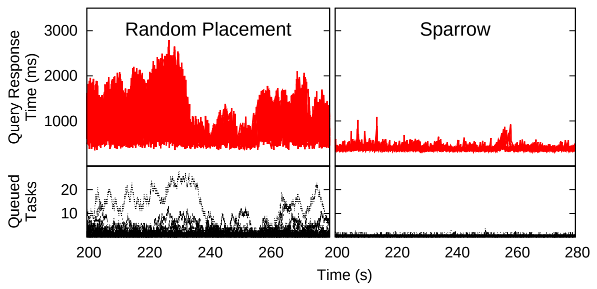


Figure 14: Queued tasks (plotted for all machines in the cluster) and query response time over an 80-second period, using random placement (left) and Sparrow (right) to place tasks in a 100-node cluster at 55% utilization. Response time exhibits a clear correlation with the number of queue tasks: high numbers of queued tasks occur concurrently with high response times. Sparrow performs dramatically better than random placement, both in terms of minimizing response time, and distributing load evenly across machines.

We evaluate these respective schedulers using a framework which launches jobs composed of 20 identical parallel tasks. The jobs are launched at regular time intervals from 5 front end nodes. They execute on 100 slaves. The task arrival rate is adjusted such that the cluster is always at 25% utilization.

With utilization fixed at 25%, we vary the task duration and record the response time of typical jobs. That is, for lower task durations, we increase the submission rate in order to keep the aggregate load constant. Results are given in Figure 15. When task durations are 5 seconds or above, Sparrow and Mesos offer roughly equal performance, both maintaining scheduling overhead of only a small fraction of response time. As task duration decreases, however, the centralized scheduling component of the Mesos master becomes heavily loaded and task launch overhead increases.

Mesos operates by offering resources to cluster frameworks and launching tasks when those offers are accepted. Once Mesos has launched a task it tracks the task status and receives messages when task start, finishes, or fails on a slave node. While the actual rate of task launches here is modest, roughly 200 tasks per second, the need to send several messages per task-lifetime (including several offers) amplifies the total rate of message passing dramatically. These messages often include complex serialized data-types, such as task and offer meta-data, which cause major serialization overhead and slow down the rate at

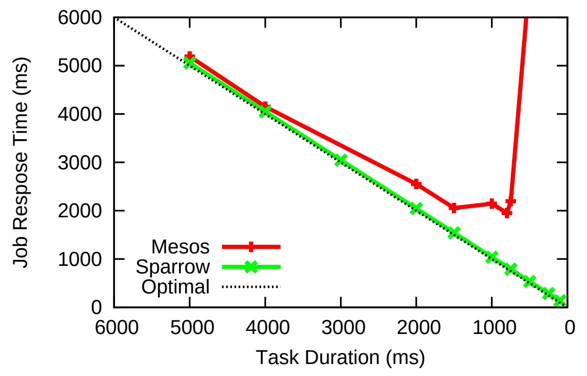


Figure 15: Response time when scheduling 10-task jobs in a 100 node cluster. Utilization is fixed at 25%, while task duration varies.

which Mesos can offer resources to frameworks.

We do not claim that Mesos could not be altered to better fit this setting, perhaps through message batching or more efficient serialization; nor do we claim that Sparrow should be used to schedule the batch workloads that Mesos was designed for. We merely observe that the alternative, simple techniques proposed in Sparrow pose a more promising path forward in the face of emerging low-latency workloads.

8 Limitations and Future Work

To handle the latency and throughput demands of low-latency frameworks, our approach sacrifices some of the features sometimes available in general cluster schedulers. Some of these limitations of our approach are fundamental, while others can be alleviated using more sophisticated techniques.

8.1 Inter-Job Constraints

Our current design does not handle inter-job constraints (e.g. "the tasks for job A must not run on racks with tasks for job B"). This limitation is relatively easy to address for the jobs scheduled by the same front end, as each front end knows the constraints of all its jobs, and the slaves where job's tasks run. In contrast, supporting inter-job constraints across front ends is much more difficult, as schedulers do not maintain global state about the system. One solution would be to have the front ends exchange job constraints as jobs are submitted, and their task placement as they are scheduled. However, this would significantly complicate the design. Studying the tradeoff between the added complexity and the benefits of supporting inter-job constraints is a topic of future work.

8.2 Gang Scheduling

Applications that need to run m inter-communicating tasks require gang scheduling. Our approach does not directly support gang scheduling. However, as shown using both the analytical and simulation results, batch sampling places almost all jobs on entirely idle machines for utilizations up to 70%, thus launching all tasks simultaneously. In addition, at the cost of increased latency, joins can be implemented using a map-reduce model, where data is shuffled and stored before being joined. This being said, investigating highly scalable mechanisms for providing gang scheduling at all utilizations is a focus of future work.

8.3 Unpredictable Task Lengths

We assume predictability in task lengths, which allows probes to return a meaningful response. While this assumption worked well for the workload we tried, workloads with heavier tails in task length may experience degraded performance.

One way to mitigate this problem at the application level is using speculation, as mentioned in §5.5, a common technique for reducing tail latencies [8].

Alternately, to remove the need for probes to predict the runtime of currently queued tasks, a worker could wait to respond to a probe until it was ready to run the task corresponding to the probe. Schedulers would assign tasks to the first schedule to reply, rather than the scheduler with the lowest reported load; since schedulers only reply when they are idle, tasks will be guaranteed to run on the probed machine that is available soonest. However, this approach is not without drawbacks: the worker machine would be idle for one round trip time per task serviced, because it needs to wait to receive the task from the scheduler. Evaluating whether the predictability benefits of this approach outweigh the inefficiency costs is one focus of future work.

8.4 Optimizing Probes

Our approach uses entirely stateless schedulers that individually probe for each job. We could improve performance by sharing information across jobs: if two jobs arrive for scheduling right after one another, the probe results from the first job may be recent enough to aid in scheduling the second job. Leveraging probe information across jobs will improve the performance of our approach without increasing the probing overhead.

9 Related Work

Scheduling in large distributed systems has been extensively studied in earlier work.

HPC Schedulers The high performance computing (HPC) community has produced a broad variety of schedulers for cluster management. However, HPC jobs tend to be monolithic, rather than composed of fine-grained tasks, obviating the need for high throughput schedulers. The highest throughput HPC scheduler we are aware of, SLURM [11], supports at most tens to hundreds of scheduling decisions per second.

Condor The Condor scheduler [19] targets high throughput computing environments, and uses a combination of centralization and distribution. Condor's performance is far below the throughput we target: Condor supports job scheduling rates of only 10 to 100 jobs per second [5]. Furthermore, Condor's heavyweight matchmaking process, which can take seconds, also make it unsuitable for sub-second workloads.

Modern Cluster Schedulers Omega, a concurrent scheduling design introduced by Google [20], uses optimistic concurrency control to schedule at massive scale. While Omega uses a centralized state store, scheduling requests are distributed over various "verticals" that schedule resources optimistically and resolve conflicts when resource usage is committed to the state store. Omega focuses on course-grained scheduling, scheduling dedicated resources for services that handle their own request-level scheduling. We instead target fine-grained scheduling, which allows high utilization by sharing resources across frameworks; we envision that batch sampling may be used to schedule a static subset of cluster resources allocated by a general scheduler like Omega. Furthermore, as demonstrated in §5.2, scheduling requests for all cores on a machine using a consolidated scheduling frameworks leads to dramatically improved performance.

Other cluster schedulers, such as Mesos [9] and Hadoop NextGen [16], target fine grained multi-framework cluster scheduling using a centralized scheduler. We sacrifice some of the functionality provided by these schedulers, including guaranteed resource reservations, to provide very high throughput and low latency scheduling decisions.

Dremel [12] achieves response times of seconds with extremely high fanout. Dremel uses a hierarchical scheduler design (which is only minimally described in published work): each query is decomposed into a serving tree, where each node in the tree is responsible for scheduling its children. Our approach does not restrict jobs to this format, and aims to schedule jobs for multiple frameworks simultaneously in order to increase cluster utilization.

Load Balancing A variety of projects ([17, 6, 21]) explore load balancing tasks in multi-processor shared-memory architectures. In such systems, processes are dynamically scheduled amongst an array of distributed processors. Scheduling is necessary each time a process is swapped out, leading to a high aggregate volume of scheduling decisions. These projects echo many of the design tradeoffs underlying our approach, such as the need to avoid centralized scheduling points. They differ from our approach because they focus on a single, parallel machine with memory access uniformity, so they dynamically *re*schedule processes to balance load. As a result, the majority of effort is spend determining when to reschedule processes.

Concurrent work from Google [8] proposes mechanisms for achieving very low response times for high fanout online services. Queries in such a service are scheduled from distributed load balancers, similar to our approach. However, Google's approaches target much lower latency queries than we focus on: a typical query is a BigTable lookup that completes in milliseconds, and response time is dominated by tail behavior (e.g. computationally intense jobs running on the same machine as one of the tasks). Because response time is dominated by tail behavior, Google proposes using speculation for the slowest tasks, leading to dramatic improvements in response time. As demonstrated in §5.5, we found speculation to have limited benefit for our target workload.

Load balancing has also been explored extensively in the theory community, as summarized by Mitzenmacher [14]. We are not aware of any work that characterizes the benefits of randomized sampling for load balancing highly parallel jobs.

10 Conclusion

This paper presents batch sampling, a simple scheduling mechanism that provides near-optimal performance for scheduling highly parallel jobs. Batch sampling assigns a job's tasks to machines by sampling a random set of machines and placing tasks on the least loaded machines in the sample. Analytical results show that for 99% of jobs in a 4-core cluster, batch sampling places all of the job's tasks on idle machines at up to 80% cluster load. Simulation confirms this result, and demonstrates that at up to 60% utilization, batch sampling performs with 1% of an optimal scheduler. Multicore architectures increase the efficacy of our approach, so clusters with more than 4 cores per machine will see even better performance. Using Sparrow, a distributed scheduler that implements this technique, we demonstrated that randomized sampling outperforms current state-of-the-art schedulers for realistic workloads.

References

[1] Apache Thrift. `http://thrift.apache.org`.

[2] The Hadapt Adaptive Analytic Platform. `http://hadapt.com`.

[3] Anonymized for Review, 2012.

[4] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proc. OSDI*, 2010.

[5] D. Bradley, T. S. Clair, M. Farrellee, Z. Guo, M. Livny, I. Sfiligoi, and T. Tannenbaum. An Update on the Scalability Limits of the Condor Batch System. *Journal of Physics: Conference Series*, 331(6), 2011.

[6] T. L. Casavant and J. G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Trans. Software Eng.*, 14:141–154, February 1988.

[7] H. Chernoff. A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the sum of Observations. In *The Annals of Mathematical Statistics*, volume 23, pages 493–507. Institute of Mathematical Statistics, 1952.

[8] J. Dean. Achieving Rapid Response Times in Large Online Services. `http://research.google.com/people/jeff/latency.html`, March 2012.

[9] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform For Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs From Sequential Building Blocks. In *Proc. EuroSys*, 2007.

[11] M. A. Jette, A. B. Yoo, and M. Grondona. Slurm: Simple linux utility for resource management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag, 2002.

[12] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.*, 2010.

[13] M. Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.*, 2001.

[14] M. Mitzenmacher. The Power of Two Random Choices: A Survey of Techniques and Results. In S. Rajasekaran, P. Pardalos, J. Reif, and J. Rolim, editors, *Handbook of Randomized Computing*, volume 1, pages 255–312. Springer, 2001.

[15] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[16] A. C. Murthy. The Next Generation of Apache MapReduce. `http://tinyurl.com/7deh64l`, Feburary 2012.

[17] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In *Proc. ACM SPAA*, 1991.

[18] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. Modeling and Synthesizing Task Placement Constraints in Google Compute Clusters. In *ACM SoCC*, 2011.

[19] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation : Practice and Experience*, 17(2-4):323–356, Feb. 2005.

[20] J. Wilkes. Omega: Cluster Management at Google. `http://research.google.com/university/relations/facultysummit2011/2011_faculty_summit_omega_wilkes.pdf`, 2011.

[21] M. Willebeek-LeMair and A. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4:979–993, 1993.

[22] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique For Achieving Locality and Fairness in Cluster Scheduling. In *Proc. EuroSys*, 2010.

[23] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. NSDI*, 2012.

[24] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. OSDI*, 2008.