Batch Sampling: Low Overhead Scheduling for Sub-Second Parallel Jobs

Kay Ousterhout, Patrick Wendell, Matei Zaharia, Ion Stoica
University of California, Berkeley

Abstract

Large-scale data analytics frameworks are shifting towards shorter task durations and larger degrees of parallelism to provide low latency. However, scheduling highly parallel jobs that complete in hundreds of milliseconds poses a major scaling challenge for cluster schedulers, which will need to place millions of tasks per second on appropriate nodes while offering millisecond-level latency and high availability. This paper presents a decentralized load balancing approach called batch sampling, based on a generalization of the power of two random choices, that performs within a few percent of an optimal centralized scheduler. We evaluate our approach through both analytical results and an implementation called Sparrow. Sparrow schedules tasks with less than 8ms of overhead and features a design that is inherently fault-tolerant and scalable.

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 [14], 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) and to be accessed by user-facing services that run sophisticated parallel computations on a per-query basis, such as language translation and highly personalized search.

Providing low response times for parallel jobs that execute on thousands of machines poses a significant scheduling challenge. Parallel jobs are composed of many (often hundreds or thousands) of concurrent tasks that each run on a single machine. Careful placement of each task is fundamental to achieving good performance: data locality and good load balancing can significantly reduce a task's completion time. In a parallel job, the response time is determined by the last task to complete, so *every* task needs to be scheduled carefully.

Sub-second parallel jobs amplify the scheduling challenge. When tasks run in hundreds of milliseconds, scheduling decisions must be made 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 must also be performed with low latency: for 100ms tasks, scheduling delays above tens of milliseconds represent intolerable overhead. Finally, as processing frameworks approach "interactive" time-scales, allowing them to power customer-facing systems, high system availability becomes a requirement. These design requirements differ substantially from those of batch workloads.

One approach to address this problem would be to design a central scheduler for sub-second parallel tasks. Such a scheduler would need to handle two orders of magnitude higher throughput than the fastest existing schedulers (e.g., Mesos [10], YARN [17], SLURM [13]) by making scheduling decisions as quickly as one per μs on average. Additionally, achieving high availability would require the replication or recovery of large amounts of state in sub-second time. Designing such a scheduler would be a difficult engineering challenge.

In this paper, we explore a radically different approach for scheduling low-latency jobs. We propose *batch sampling*, a scheduling technique that maintains *no* shared state about cluster load and instead places tasks based on instantaneous load information from slaves. Because batch sampling does not rely on shared state, scheduling can be performed by a large number of distributed schedulers that place tasks in parallel; this distributed architecture dramatically improves throughput and availability compared to a centralized scheduler.

Batch sampling takes inspiration from web architectures where multiple frontends balance incoming requests across service nodes. Web load balancing techniques and batch sampling both provide excellent scaling and availability, but the key challenge is avoiding hotspots that can be caused by several frontends assigning requests to the same server. For load balancing single-task web requests, the well-known "power of two random choices" load balancing technique [15] alleviates hotspots by having each frontend probe two random servers and send the request to the least loaded one. Unfortunately, as we show, the power of two random choices is far less effective for *parallel* jobs, because the

tail of its queueing delay distribution is still high. This problem worsens as the level of parallelism increases, since a single task placed in a long queue delays the response time of the entire job.

Batch sampling generalizes the power of two random choices to handle parallel jobs. To schedule a job containing m parallel tasks, a frontend probes a batch of $d \cdot m$ nodes (where $d > 1$), and places the job's tasks on the m least loaded nodes. Batch sampling provides lower tail queueing delay than using two choices per task, because it can aggregate information across the set of all sampled nodes. Furthermore, tail queueing delay *improves* with higher degrees of parallelism because the frontend gets access to a larger sample, unlike with the power of two random choices.

In spite of lacking central coordination, batch sampling performs near-optimally. Using mathematical analysis, we demonstrate that at up to 70% load, 99% of jobs perform identically with batch sampling to if they were scheduled using an omniscient centralized scheduler (§4). Simulation results verify the analysis, and demonstrate that batch sampling provides response times within 4% of those provided by an omniscient, centralized scheduler at up to 60% load (§5). Batch sampling performs increasingly well with the number of cores per machine, making it well suited for today's clusters.

We have implemented batch sampling in a scheduler called Sparrow that we run on a 100-node cluster. Sparrow achieves near optimal scheduling performance, scheduling with fewer than 8 milliseconds of overhead. To evaluate batch sampling under a realistic workload, we use Sparrow to schedule short TPC-H queries and find that it performs within 12% of optimal, and reduces 95% response times by 6x at high utilization compared to a simple, randomized load balancing approach. Unlike other schedulers, where complexity grows with the number of users and the size of the cluster, Sparrow can easily distribute the workload across multiple schedulers in order to consistently provide a median scheduling latency of 8 milliseconds. These experimental results are shown in §7.

Batch sampling forgoes some features of current cluster schedulers to support low-latency parallel jobs; we discuss these limitations in §8. Finally, we summarize related work in §9.

2 Target Workload

We design a scheduler specifically for frameworks that achieve low latency job execution times using parallelism. A scheduler for low latency, highly parallel jobs must meet different requirements than those of batch workloads:

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. We expect that low-latency frameworks will be used to power user-facing services, making *high availability an operating requirement*.

To meet these requirements, we forgo some features of today's cluster schedulers; for instance, we do not design for arbitrarily long tasks that take hours or days to complete. Instead, we target runtimes that span no more than 1-2 orders of magnitude (task durations in the 100ms-5s range). To co-exist with long-running, batch jobs, we envision that our scheduler will run tasks in a statically or dynamically allocated portion of the cluster that has been allocated by a more general resource manager such as YARN [17], Mesos [10], or Omega [20].

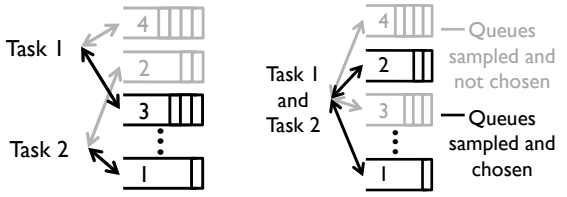
In designing for low-latency, parallel jobs, we do not abandon all features of current centralized schedulers. We target an environment that may be running low-latency queries from multiple frameworks and from multiple users, so the scheduler must enforce resource allocation policies when aggregate demand exceeds capacity. While our distributed, latency-oriented design cannot perfectly enforce resource allocation policies as a central scheduler could, we demonstrate that a distributed architecture can closely approximate global fairness policies. We also support common types of 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).

3 Probe-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 schedulers that place tasks in parallel; distribution improves both throughput and availability.

3.1 Batch Sampling

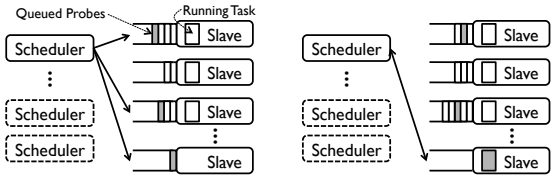
We propose using distributed schedulers that maintain *no* shared state; instead, schedulers place tasks using instantaneous load information from slaves. Our approach, which we call *batch sampling*, generalizes Mitzenmacher's "power of two random choices" load balancing



(a) Per-task sampling selects queues of length 1 and 3.

(b) Batch sampling selects queues of length 1 and 2.

Figure 1: Placing a parallel, two-task job. Batch sampling outperforms per-task sampling because tasks are placed in the least loaded of the entire *batch* of sampled queues.



(a) When a job arrives at a scheduler with m tasks, the scheduler probes dm machines. Slaves queue the probes until they have available resources to run the task.

(b) When a slave has available resources to run a task, it requests the task from the scheduler, and then runs the task.

Figure 2: Scheduling an m-task job using batch sampling.

technique [15].

The power of two random choices technique proposes assigning an incoming task to the least-loaded of two randomly chosen slaves. Each slave is assumed to run a single task at a time and queue remaining tasks; the "least-loaded" slave is chosen based on the length of the queue. In his seminal work [15], Mitzenmacher demonstrated that assigning tasks in this manner performs dramatically better than assigning each task to a random slave, and surprisingly, closely approximates placing tasks on the globally least-loaded slave.

A direct application of the power of two random choices to parallel jobs would simply place each of the job's tasks on the least loaded of two randomly selected slaves. Using mathematical analysis (§4) and simulation (§5.2), we demonstrate that this approach leads to poor *job* response times. The response time of a parallel job is dictated by the slowest response time of all of the job's constituent tasks, which can be dramatically worse than the median task response time.

Batch sampling generalizes the power of two random choices by sampling for all tasks in the job in a single batch. Rather than placing each task individually, as in the power of two random choices ("per-task sampling"), batch sampling places the m tasks in a job on the least loaded of dm slaves (for $d \geq 1$). As shown in Figure 1, with $d = 2$, batch sampling and per-task sampling sample the same number of queues. However, because batch sampling can select the m least loaded of all $2m$ queues, it performs at least as well as per-task sampling and often much better. As shown in §4, as the number of tasks in a job increases, batch sampling provides a significant improvement over sampling individually for each task.

Batch sampling relies on no shared state between jobs that are being scheduled, so can easily be distributed over multiple schedulers. When a scheduler receives a job with m tasks, the scheduler randomly selects a set of dm candidate slaves ($d \geq 1$), where d is a tunable parameter. The scheduler sends a *probe* to each of the dm slaves, requesting instantaneous load information. Based on the load information, the scheduler places the job's tasks on the set of machines that collectively offer the lowest response time. State need not be shared between jobs, so multiple schedulers can operate independently.

The remainder of this paper uses *wait time* to describe the time from when a task is launched to when it begins executing, *service time* for the time the task spends executing and *response time* as the sum of the two.

3.2 On-Demand Task Assignment

One challenge with batch sampling is the need to predict queueing time on slaves. In the simplest realization of batch sampling, slaves would immediately reply to probes with the number of queued tasks. The scheduler would wait for replies from all slaves, and then place the job's tasks on the slaves with the shortest queues. While appealing in its simplicity, this approach assumes queuing time can accurately be predicted based on the number of tasks in the queue. Predicting task wait times based solely on queue length is typically ineffective [8], particularly when task service times are known to be variable. The difficulty of predicting wait times renders this strategy impractical for any real deployment.

In order to avoid needing to predict task wait time, we propose implementing batch sampling using *on-demand task assignment*. As shown in Figure 2, slaves queue probes rather than replying immediately. When the slave has enough resources to run a new task, it responds to the probe with a `getTask()` RPC, requesting a task specification from the scheduler. The scheduler returns a specification of the task; as described in §6, this specification is opaque to the scheduling system and is passed on to the application executor running on the slave. The scheduler does not determine which tasks will be assigned to which slaves until receiving `getTask()` requests, so that it can assign the m tasks to the slaves where the tasks will execute soonest without needing to predict the runtimes of other tasks. Some slaves will reply to the scheduler af-

ter it has launched all tasks for the job, in which case the scheduler simply replies with a no-op message.

On-demand task assignment avoids needing to predict task runtimes at the expense of idle time while slaves are requesting tasks. As we show in §4.3, this tradeoff is worthwhile in our target environment, where network RTT is short relative to task length.

3.3 Handling Placement Constraints

Batch sampling easily handles jobs with job-level constraints (e.g., all tasks should be run on a slave with a GPU) by simply selecting the dm candidate slaves from the subset of slaves that satisfy the constraint.

Probe-based scheduling also handles task-level constraints. Many jobs may have task-level placement constraints; for example, if each task operates on some input data the task should ideally be co-located with the memory or disk storing the input data. To handle placement constraints, the scheduler randomly probes d slaves for each task, so the total number of slaves probed remains dm. However, instead of choosing the dm slaves randomly from the entire cluster, the d slaves for each task are selected randomly from the set of preferred nodes.

3.4 Straggler Mitigation

Batch sampling does not explicitly consider stragglers; instead, straggler mitigation can be implemented on top of batch sampling. Straggler mitigation techniques such as task speculation [3, 24, 4, 8], deal with the nondeterministic variation in task execution time due to unpredictable causes (e.g., resource contention and failures). Such variation is best detected by the application generating the jobs, which is likely to have more information about expected task durations. When an application determines that some tasks in a job are taking unexpectedly long to complete, the application can simply re-submit these tasks to be scheduled as a new, separate job using batch sampling. Thus, batch sampling is orthogonal and complementary to straggler mitigation techniques.

3.5 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 [11] opt for *weighted fair sharing* or proportional sharing, which allocates each user a share of the cluster proportional to the user's weight. When a user is not using its share, it is distributed evenly amongst other users. Proportional sharing has global semantics: it applies 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 probes from each user. As a result, two users competing on the same slave will get shares 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 be allocated their fair share of the aggregate resources they use *in expectation*. Fair sharing provides isolation, as each user will get its share irrespective of the demand from others at each node.

Some cluster operators may prefer strict priorities to fair shares; e.g., jobs serving user-facing requests should receive strict priority over explorative queries. Our design can provide a strict priority service by maintaining a separate queue for each priority at each slave and always servicing tasks from the highest priority active queue.

4 Analysis

This section presents an analytical description of the performance of batch sampling in comparison with other techniques. Recall that scheduling parallel jobs is difficult because the response time of a job is determined by the worst response time of its constituent tasks. Even if median *task* response time is low, job response time may be high, because a job's response time is dictated by the response time of the last task to complete, so may be substantially worse than the median task response time. This makes techniques which attempt to minimize mean or median task response time generally ineffective, particularly for jobs with a large number of tasks. This section provides analytical descriptions of job response time under (i) purely random approaches, (ii) direct application of the power of two random choices and (iii) batch sampling. The section concludes by quantifying the performance of alternative implementions of 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. The mean job arrival rate is $\frac{\rho n}{mt}$, where n represents the number of servers in the cluster, t denotes the mean service time of a task, and we refer to $\rho < 1$ as the load (Table 1 summarizes our notation). Servers run a single task at a time, and queue remaining tasks. We assume zero network delay and consider a system with an infinitely large number n of servers.

4.2 Probability of Zero Wait Time

To provide intuition for the performance of different load balancing approaches, we begin by considering the prob-

n	Number of servers in the cluster
ρ	Load
m	Tasks per job
d	Probes per task
t	Mean task service time

Table 1: Summary of notation.

Random Placement	$(1-\rho)^m$
Per-Task Sampling	$(1-\rho^d)^m$
Batch Sampling	$\sum_{i=m}^{d\cdot m}(1-\rho)^i\rho^{d\cdot m-i}\binom{d\cdot m}{i}$

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

ability that a job experiences zero wait time, which happens when all tasks in the job are placed on idle slaves. Since an ideal, omniscient scheduler can always place tasks on idle slaves when the cluster is under 100% utilized, quantifying how often our approach places jobs on idle slaves closely approximates 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.

One insight we exploit in this analysis is that the fraction of non-idle slaves in the cluster will be equal to the load ρ on the cluster. For $\rho < 1$, 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, in expectation, ρn slaves will be running one task (with potentially additional tasks queued), and the remaining $(1-\rho)n$ slaves will be idle.

When tasks are randomly assigned to slaves, the probability that all m tasks in a job are assigned to idle machines is $(1-\rho)^m$. Figure 3 (top) plots this probability for a two different job sizes and demonstrates that random placement rarely places all tasks in a job on idle machines, even for modest job sizes and low cluster loads.

Previous work has demonstrated that for scheduling individual tasks, choosing the less loaded of two randomly sampled queues yields drastically improved performance over random placement [15]. We directly apply this technique to 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. Using per-task sampling rather than random placement improves results significantly, as shown in Figure 3.

We use per-task sampling to place jobs with task-level constraints, as described in §3.3. When a job's tasks are

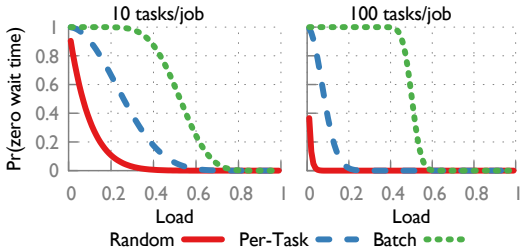


Figure 3: Probability that a job will experience zero wait time in a single-core environment, using random placement, sampling 2 servers/task, and sampling $2m$ machines to place an m-task job.

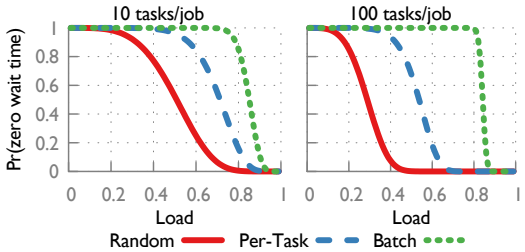


Figure 4: Probability that a job will experience zero wait time in a system of 4-core servers.

constrained to run on a small number of nodes, per-task sampling closely approximates optimal performance, because even an ideal placement is constrained to only a few choices for each task.

While per-task sampling is an improvement over random sampling, it 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, 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 describes the probability of placing all tasks on idle machines using this approach. As shown in Figure 3, batch sampling outperforms per-task sampling, and the improvement from using batch sampling over per-task sampling increases with the number of tasks per job.

4.2.1 Multicore

Operating in environments where multiple tasks can run concurrently on a single machine, such as when machines have several cores, 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. Because each server can run multiple concurrent tasks, each probe describes load on c processing units rather than just one, which increases the likelihood of finding an idle or lightly loaded processing unit.

To analyze the performance in multicore environments, we make two simplifying assumptions: first, we assume that the distribution of idle cores is independent of whether cores reside on the same machine; and second, we assume that the scheduler places at most 1 task per machine, even if multiple cores are available. Based on these assumptions, the probability that a given machine does not contain at least one empty core is ρ^c. Replacing ρ in Table 2 with ρ^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 achieves near perfect performance for even higher loads.

4.3 On-Demand Task Assignment Performance

As described in §3.2, batch sampling can be implemented in a variety of ways, each of which makes a different tradeoff. We propose that slaves wait to request tasks until they have available resources to run the task, which avoids needing to predict queueing times and ensures that a job's m tasks will be run on the m slaves in the probed set that can run the tasks soonest. However, this improvement comes at the expense of wasting idle resources on the slaves that are requesting new tasks to run.

In our target scenario, requesting tasks only when resources are idle leads to only 2% efficiency loss. The fraction of time that node monitors spend idle while requesting tasks is $d \cdot \mathrm{RTT}/t$ (where d denotes the number of probes per task, RTT denotes the mean network round trip time, and t denotes mean task service time). In our un-optimized deployment on EC2, median network round trip time was 1 millisecond. We expect that the shortest tasks will complete in about 100ms and that schedulers will use a probe ratio of no more than 2. Thus, we waste no more than 2% of cluster resources and in turn ensure that each always runs on the first available queue. The small efficiency loss relies on the assumption that network latencies are orders of magnitude shorter than task runtimes; in other environments where network latencies and task runtimes are the same order of magnitude, on-demand task placement will not present a worthwhile tradeoff.

5 Simulation Results

To understand system behavior under a wide variety of parameterizations and workloads, we built an event-based simulator. Simulated response times match those observed in our deployment, as shown in §7.1.

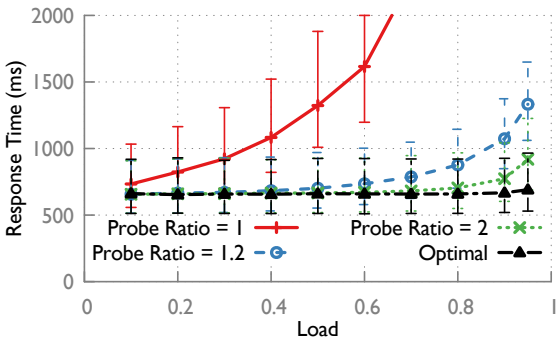


Figure 5: Median, 5th, and 95th percentile response times for 500-task jobs in a simulated cluster of 10,000. Tasks are exponentially distributed with mean 100ms.

Simulated results demonstrate that batch sampling provides near-optimal response times: response times using batch sampling with probe ratio of 2 are within 4% of response times with an optimal scheduler at up to 60% load. Performance degrades as load increases: at 95% utilization, batch sampling performs within 33% of optimal. We compare batch sampling with on-demand task assignment to alternatives, including per-task sampling directly based on the power of two random choices, and simple batch sampling (without on-demand task assignment) and demonstrate that batch sampling provides a 2x improvement over both alternatives at high utilizations. Batch sampling performs increasingly well as the number of cores per machine increases, which bodes well given current trends towards many-core machines.

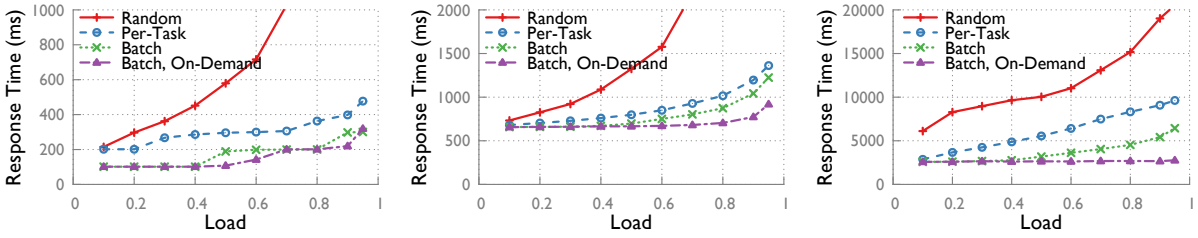
5.1 Impact of Probing Ratio (d)

One key question is what degree of oversampling is necessary to achieve good performance. Figure 5 shows the impact of the probing ratio d on job response time, for a 10,000 node cluster running 500-task jobs that arrive as a Poisson process. Task runtime is exponentially distributed with mean 100ms. Even though mean task runtime is 100ms, job response time is much longer because it is dictated by the longest response time of 500 tasks.

Simulation results demonstrate that modest probing achieves significant gains: even $d = 1.2$ 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 4% of the median response time with an omniscient scheduler at up to 60% load. This gap widens to 33% at 95% load.

5.2 Alternate Sampling Techniques

The scheduling technique presented in this paper, batch sampling with on-demand task assignment, improves



(a) Constant task duration of 100ms (b) Exponentially distributed task durations with mean 100ms (c) Pareto-distributed (shape parameter 1.5) task durations

Figure 6: Comparison of random placement, per-task sampling, standard batch sampling (where slaves reply immediately to probes and schedulers place tasks in the shortest queues) and batch sampling with on-demand task assignment. Graphs depict median, 5th, and 95th percentile response times for 500-task jobs in a 10,000-node cluster.

upon the Power of Two Random Choices load balancing technique in two ways. First, batch sampling generalizes the power of two random choices to parallel jobs by sampling for all tasks in the job in a single batch. Second, on-demand task assignment removes the need to predict the completion times of currently running tasks.

Figure 6 quantifies the benefit of these two improvements in a simulated 10,000-node cluster running 500-task jobs. Each plot compares random placement (the simplest possible load balancing technique), a direct application of the power of two random choices to parallel jobs ("per-task sampling"), batch sampling, and batch sampling with on-demand task placement. We compare the 4 techniques using 3 different task-duration distributions that have increasingly heavy tails: constant, exponentially-distributed, and pareto-distributed task lengths. For all three distributions, batch sampling with on-demand task assignment presents a significant improvement over the other techniques. The improvement from doing batch sampling and on-demand task assignment widens as the distribution gets more heavy tailed; for pareto-distributed task durations at 95% utilization, batch sampling with on-demand task placement provides a 3x reduction in response time compared to per-task sampling.

5.3 Benefit of Operating on Multiple Cores

Simulation results confirm the analytical results in §4.2.1 demonstrating that batch sampling performs better in environments were multiple tasks can run concurrently on each machine. When servers can run multiple concurrent tasks, each probe gains more implicit information, because it describes load on multiple processing units. The simulation models c-core machines by running c tasks concurrently on each slave and queueing remaining tasks. As shown in figure 7, running on quad-core machines improves response times by over 20% (at 95% load) compared to running on single-core machines, keeping task arrival rate and total cluster processing power fixed.

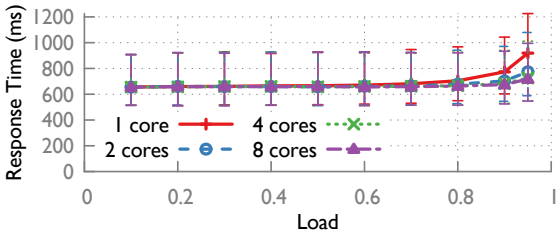


Figure 7: Response time in a 10,000-core cluster using batch sampling improves with the number of cores per slave. Each job executes 500 tasks, each of which has exponentially distributed runtime with mean 100ms.

6 Implementation

Sparrow is a distributed scheduler implementation that uses batch sampling to perform low latency task placement.[1] As shown in Figure 8, Sparrow schedules from a distributed set of schedulers that are each responsible for assigning tasks to slaves. Because batch sampling does not require *any* communication between schedulers, arbitrarily many schedulers may operate concurrently, and users or applications may use any available scheduler to place jobs. Schedulers expose a Thrift service [1] to allow frameworks to submit scheduling requests. Thrift is cross-platform, allowing clients in a variety of languages. Each scheduling request includes a list of task specifications; the specification for a task includes a task description, opaque to Sparrow, and an optional list of nodes on which the task should be run.

A Sparrow node monitor runs on each slave, and federates resource usage on the slave by replying to probes and requesting task specifications from schedulers when resources become available. Node monitors run tasks in a

[1]The Sparrow code is publicly available at `https://github.com/radlab/sparrow`.

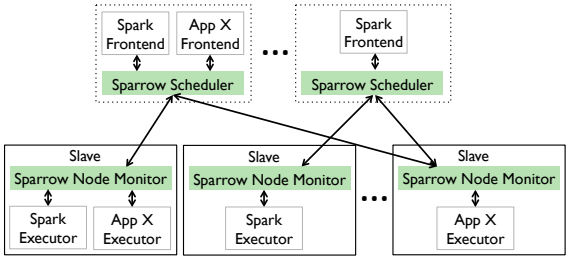


Figure 8: Frameworks that use Sparrow are decomposed into frontends, which generate tasks, and executors, which run tasks. Frameworks schedule jobs by communicating with any one of a set of distributed Sparrow schedulers. Sparrow node monitors run on each slave machine and federate resource usage on that slave.

fixed number of *slots*; in our implementation, the number of slots on each machine is set to the number of cores.

Sparrow performs task scheduling for one or more concurrently operating frameworks. As shown in Figure 8, frameworks are composed of long-lived *frontend* and *executor* processes, a model employed by many systems (e.g., Mesos [10]). Frontends accept high level queries or job specifications (e.g., a SQL query) from exogenous sources (e.g., a data analyst, web service, business application, etc.) and compile them into parallel tasks for execution on workers. Frontends are typically distributed over multiple machines to provide high performance and availability. Executor processes are responsible for executing tasks, and are long-lived to avoid startup overhead such as shipping binaries or bringing large datasets into cache. Executor processes for multiple frameworks may run co-resident on a single machine; the node monitor federates resource usage between co-located frameworks.

6.1 Spark on Sparrow

In order to test batch sampling using a realistic workload, we have ported Spark [23] to Sparrow by writing a Spark scheduling plugin that uses the Java Sparrow client. This plugin is 132 lines of Scala code.

The execution of a Spark job begins at a Spark frontend, which compiles a functional query definition into a multi-phase parallel job. The first phase of the job is typically constrained to execute on machines that contain partitions of the cached input data set, while the remaining phases (which read data shuffled over the network) can execute anywhere. The Spark frontend passes a list of task descriptions (and any associated placement constraints) for the first phase of the job to a Sparrow scheduler, which assigns the tasks to slaves. Because Sparrow schedulers are lightweight, we run a scheduler alongside each Spark frontend to ensure minimum scheduling latency. When the Sparrow scheduler assigns a task to a slave, it passes the task description provided by the Spark frontend to the Spark executor running on the slave, which uses the task description to launch the task. When one phase completes, Spark requests scheduling of the tasks in the subsequent phase.

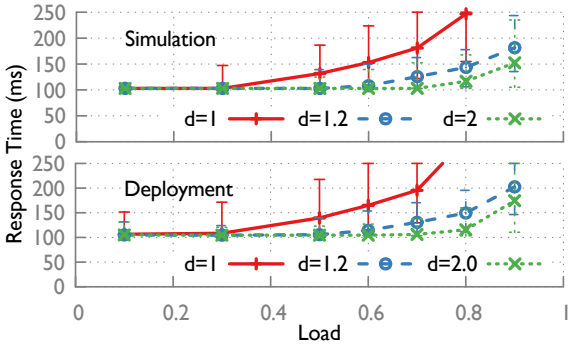


Figure 9: Effect of probe ratio, d, and utilization on median response time in a cluster composed of 100 slaves and 10 schedulers. Each job consists of 10 tasks that run consecutive floating-point multiplications for 100 milliseconds. Error bars show 5th and 95th percentiles.

7 Experimental Evaluation

We evaluate batch sampling by deploying Sparrow on a cluster of 110 EC2 high memory double extra large instances, each of which features 34.2GB of memory and 4 cores. Using constant-length tasks, we examine the impact of the probing ratio and find that with a probing ratio of 2, Sparrow provides response times within 3% of optimal at up to 70% utilization. We also use Sparrow to schedule tasks for a TPC-H [7] workload, which features heterogeneous queries, and demonstrate that Sparrow provides response times within 12% of optimal and with median queueing delay of 8ms.

7.1 Impact of Probing Ratio (d)

Evaluating the impact of the probing ratio, d, in the deployment demonstrates that deployed performance matches simulated performance. Figure 9 (bottom) shows the impact of the probing ratio d on job response time for a 110-node cluster for both the real and simulated systems. 10 nodes serve as schedulers, and the remaining 100 nodes (400 total cores) are slaves. Schedulers launch 10-task jobs; deployed tasks continuously run floating-point multiplications to fully utilize a single CPU for 100ms and simulated tasks run for exactly 100ms. The simulation uses a fixed network round trip time of 1 millisecond, consistent with the deployment.

Phase	Response Time		
	Median	5th %ile	95th %ile
Table-Scan	157ms	121ms	467ms
Group-By	150ms	88ms	501ms
Order-By	74ms	53ms	340ms

Table 3: Median, 5th, and 95th percentile response times of the 3 phases of TPC-H query 3. Response times are measured in a cluster running at approximately 65% utilization.

Deployed and simulated results are nearly identical; the deviation at high utilization stems from minor variations in task service times in the deployment when the slaves are heavily utilized.

Figure 9 also demonstrates that, consistent with simulated and analytical results, a modest amount of probing leads to near-optimal response time. Analytical results in §4.2 predicted that with $d = 2$, nearly all 10-task jobs would be placed on idle cores when the cluster was under 70% utilization; consistent with this prediction, in the deployment with $d = 2$, median response time is within 2.5ms of optimal at up to 70% utilization.

7.2 Performance on a Realistic Workload

To evaluate Sparrow against a realistic workload, we measure Sparrow's performance scheduling queries for Shark [9], a large scale data analytics platform. Shark lets users load working sets into memory on several parallel machines and analyze them using a SQL-based query language. Our experiment features several users concurrently using Shark to run ad-hoc analytical queries in a 100-node cluster. The data set and queries are drawn from the TPC-H OLAP database benchmark.

This experiment differs from the synthetic workload in several important ways. First, Shark queries are compiled into multiple Spark [23] stages that each trigger a submission request to Sparrow. The response time of a Shark query is dictated by the accumulated response time of each sub-phase. Second, the duration of each phase is not uniform, instead varying from a few tens of milliseconds to several hundred. Third, the phases in this workload have heterogeneous numbers of tasks, corresponding to the amount of data processed in that phase.

The query corpus features four TPC-H queries, with 2, 3, 3, and 5 phases respectively. The queries operate on a denormalized in-memory copy of the TPC-H input dataset (scale factor 1). Each frontend queries a distinct data-set that is striped over 90 of the 100 machines: this consists of 30 three-way replicated partitions. Figure 3 breaks down a single Shark query in terms of its constituent phases. The first phase scans and filters records,

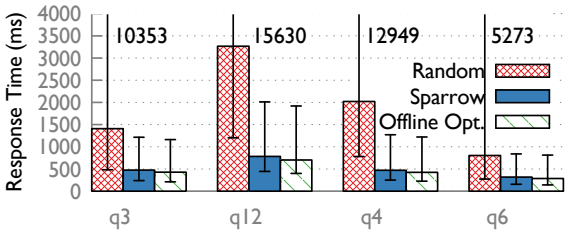


Figure 10: Median, 5th, and 95th percentile response times for TPC-H queries run using both random placement and Sparrow with probe ratio 2. Both are from a cluster running at average utilization of 65%.

the second aggregates data into groups (via a shuffle) and the third sorts by a group-related value. Each phase is listed with its median response time during the experiment. These phases are each scheduled through Sparrow's job launch API.

In the experiment, each of 10 users launches random permutations of the TPC-H queries continuously for several minutes. The queries are launched at an aggregate rate of approximately 30 per second in order to keep the cluster 65% utilized, on average. Queries are launched through 10 Sparrow schedulers, which sample (ratio $d = 2$), enqueue, and launch tasks on any of 100 slaves.

Figure 10 plots the median, 5th, and 95th percentile response times for each TPC-H query across all frontends during a 2 minute sampled period in the middle of the experiment. Approximately 4000 queries are completed during this window, resulting in around 12TB of in-memory table scans (the first phase of each query scans the entire table). The figure plots response time with Sparrow, along with 2 comparisons. The first is an analogous run using entirely randomized task placement. The second is an offline optimal calculation based on the service times witnessed during the experiment. That is, it takes each query run and calculates its zero-queue response time as a function of the service time of its constituent phases. This offline optimal calculation is conservative – it does not take into account task launch delay or queuing which is unavoidable due to small utilization bursts – both of which are inevitable even with an omniscient scheduler. Even so, Sparrow performs within 12% of the offline optimal.

Sparrow outperforms a randomized approach and approaches the offline optimal value, even though the ten Sparrow frontends act autonomously. Sparrow improves over randomized placement because it balances tasks more effectively amongst servers, thereby preventing unnecessary queueing. Figure 11 shows a CDF of task queue time during this same period, for both Sparrow

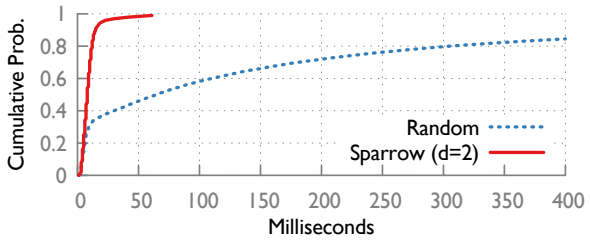


Figure 11: CDF of task wait time during a TPC-H benchmark experiment using both Sparrow (with $d = 2$) and random placement. Tasks experience much more queuing with random placement than with Sparrow.

and random. When randomization alone is used, significant queuing occurs for many tasks. Since query response time relies on every phase completing, and each phase completing relies on every task completing, even small variations amplify to poor response times. Sparrow improves 95th percentile response time over random by over 6x.

7.3 Resource Allocation Policies

As described in §3.5, batch sampling supports resource allocation policies such as proportional fairness or strict priorities. Sparrow implements weighted fair sharing: each slave maintains a separate queue for each user and determine which probe to reply to based on weighted fair queueing.

To evaluate Sparrow's ability to enforce fairness policies, we shared a cluster of 100 quad-core slaves among 3 users that were each given a 1/3 share. Two users submit at an average rate[2] of 75% of their share (100 running tasks), while the the third user, user 2, submits enough tasks to use 50% of the cluster, much more than its share. Sparrow schedules jobs using a probe ratio of 2, and each user submits jobs composed of 10 100ms tasks.

Weighted fair sharing dictates that the fair share for each user at any point in time is `min(user demand, user share)`; as shown in Figure 12, users 0 and 1 receive their fair share, even though user 2 is attempting to use resources beyond its share. User 2 is allowed to use additional slots beyond its share of 133 concurrently running tasks, to avoid wasted utilization, but is *not* allowed to infringe on the fair share of the other users.

Users often care most about isolation: even if users are receiving their fair share, a spike in one user's usage should not affect the *response time* of jobs submitted by other users. Table 4 depicts job response time in the above scenario. Users 0 and 1 receive somewhat longer

[2]The tasks for all users arrive as a Poisson process, so the instantaneous submission rate fluctuates around the average.

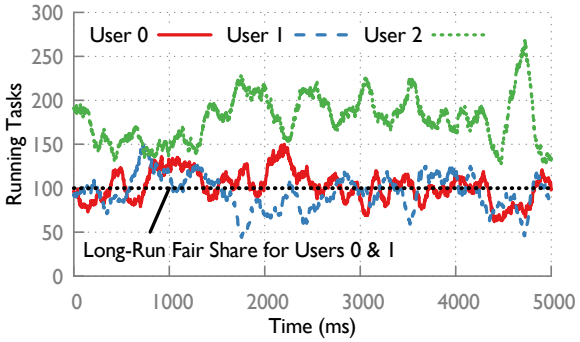


Figure 12: Number of running tasks in a 400-core cluster running 10-task jobs for 3 users. All users are given equal shares of the cluster. Users 0 and 1 submit tasks at below their fair share, while user 2 submits tasks at a higher rather than its fair share.

User	Response Time		
	Median	5th %ile	95th %ile
User 0	166.5ms	129.5ms	318.5ms
User 1	167.5ms	129.5ms	334.5ms
User 2	36.55s	48.13s	78.03s

Table 4: Response time for each of 3 users submitting 10-task jobs composed of 100-ms tasks. Users 0 and 1 are submitting at 75% of their share whereas user 2 is submitting at far over its share. User 2 experiences infinite queueing, which only minimally affects performance for users 0 and 1.

response times than if user 2 were also using 75% of its share, in which case each user would experience median response times of 106ms. This increase in response time is unavoidable given that Sparrow does not implement preemption, so tasks for users 0 and 1 may have to wait for a task from user 2 to complete. However, given that user 2 experiences infinite queueing, users 0 and 1 are minimally affected. Sparrow's approach to fairness is only a first step; exploring improved fairness policies is a subject of future work.

7.4 Relative Performance of Central Schedulers

Current cluster schedulers are not optimized for sub-second tasks. To demonstrate, we run a simple experiment scheduling tasks with both Sparrow and Mesos [10] in a 100 node cluster. We measure scheduler performance under different task durations by varying task duration and adjusting job submission rate accordingly to keep load fixed. Jobs, each composed of 20 fixed-duration tasks, are launched at regular intervals in a 100-node cluster. The cluster is kept at a constant 25% utilization (this modest rate is the maximum utilization that Mesos can sustain given the submission frequency).

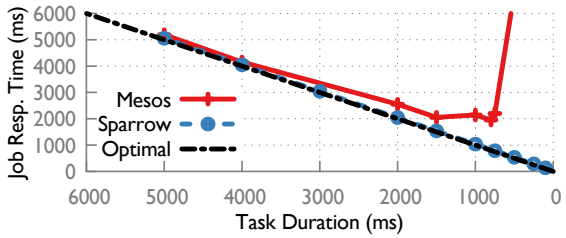


Figure 13: Response time when scheduling 10-task jobs in a 100 node cluster using both Sparrow and Mesos. Utilization is fixed at 25%, while task duration varies.

Figure 13 depicts job response time as task duration decreases. When task durations are 5 seconds or above, Sparrow and Mesos both maintain scheduling overhead of only a small fraction of response time. As task duration decreases, however, Mesos becomes heavily loaded and scheduling overhead increases. At this stage Mesos becomes overwhelmed with messaging overhead, due to complex serialized data types used in its interface (predicate-based constraints, generic resource descriptions, etc), and experiences congestion collapse. We operates Mesos in standalone mode rather than high availability mode; high availability mode writes to a shared quorum-based edit log to provide availability at the cost of additional latency.

We do not claim that Mesos could not be altered to better fit this setting, nor do we claim that Sparrow suffices as a general-purpose cluster scheduler. We merely observe that Sparrow presents a simple, demonstrably effective alternative to scaling current approaches.

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 are the focus of future work.

Scheduling Policies When a cluster becomes oversubscribed, batch sampling supports aggregate fair-sharing or priority-based scheduling. Sparrow's distributed setting lends itself to *approximated* policy enforcement in order to minimize system complexity. Bounding this inaccuracy under arbitrary workloads is another focus of future work.

With regards to job-level policies, 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 easy to address for the jobs scheduled by the same frontend, as each frontend knows the constraints of all its jobs, and the slaves where jobs' tasks run. Supporting inter-job constraints across frontends is difficult to do without significantly altering the design.

Gang Scheduling Applications that need to run m inter-communicating tasks require gang scheduling. Gang scheduling is typically implemented using bin-packing algorithms which search for and reserve time slots in which an entire job can run. Because Sparrow queues tasks on several machines, there is no central point from which to perform bin-packing. While it is often the case that Sparrow places all jobs on entirely idle machines, this is guaranteed. Applications that use Sparrow could implement inter-task communication by using a map-reduce model, where data is shuffled and stored (in memory or on disk) before being joined. This trade-off is also made by several other cluster schedulers (e.g., [10], [17], [12]).

Delay Scheduling Sparrow's design assumes that tasks either have firm placement constraints (such as needing to run where memory and disk data are stored) or no constraints. State-of-the-art MapReduce schedulers relax this assumption and employ *delay scheduling* to increase throughput [22]. Delay scheduling interprets placement constraints as "soft" and may place tasks on other machines after a certain amount of time has lapsed. Delay scheduling has a direct analog within the batch sampling framework: the scheduler would first restrict the set of probed slaves to the set of slaves that have tasks' input data, but if these probes take longer than some threshold to reply, the scheduler would send additional probes to slaves that didn't store the data. Exploring the throughput gain resulting from this technique is the focus of future work.

Cancellation Schedulers could reduce the amount of time slaves spend idle due to on-demand task assignment by canceling outstanding probes once all tasks in a job have been scheduled. Cancellation would decrease idle time on slaves with little cost.

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. HPC jobs tend to be monolithic, rather than composed of fine-grained tasks, obviating the need for high throughput schedulers. HPC schedulers thus optimize for large jobs with complex constraints, using constrained bin-packing algorithms. High throughput HPC schedulers, such as SLURM [13], target maximum throughput in the tens to hundreds of scheduling decisions per second.

Condor The Condor scheduler [19] targets high throughput computing environments using a combination of centralization and distribution. Condor's scheduling throughput is again in the regime of 10 to 100 jobs per second [5]. This is the result of several complex features, including a rich constraint language, a job checkpointing feature, and support for gang scheduling, all of which result in a heavy-weight matchmaking process. Unlike Sparrow, Condor is a general purpose cluster resource manager.

Modern Cluster Schedulers Quincy [12], Mesos [10], and YARN [17] are modern task-level cluster schedulers built for frameworks such as Hadoop. All three schedulers feature a centralized architecture. They all accept basic constraints over task placement, such as data-locality constraints.

Quincy [12] provides fairness and locality by mapping the scheduling problem onto a graph and using a solver to compute the optimal online schedule. Because the size of the graph is proportional to the number of slaves, scheduling latency grows with cluster size. In a 2500 node cluster, the graph solver takes over a second to compute a scheduling assignment [12]; while multiple jobs can be batched in a single scheduling assignment, waiting seconds to schedule a job that can complete in hundreds of milliseconds is unacceptable overhead.

YARN [17] introduces distributed per-job application masters, making it more scalable than the original Hadoop MapReduce scheduler. However, it relies on periodic (1s, by default) heartbeats from slaves to determine when resources have become available. To avoid wasted resources, heartbeats need to occur more frequently than the expected task turnover rate on a machine; to handle a multi-core machines running subsecond tasks, each slave would need to send hundreds of heartbeats per second, which would easily overwhelm the resource manager. Furthermore, high availability has not yet been implemented for YARN.

Mesos [10] imposes minimal scheduling overhead by delegating all aspects of scheduling other than fairness to framework schedulers and employs batching to handle high throughput. Still, Mesos performs poorly for subsecond tasks, as demonstrated in §7.4.

A variety of other schedulers, e.g., Omega [20], target course-grained scheduling, scheduling dedicated resources for services that handle their own request-level scheduling. Batch sampling instead targets 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.

Dremel [14] achieves response times of seconds with extremely high fanout. Dremel uses a hierarchical scheduler design whereby each query is decomposed into a serving tree. This approach exploits the internal structure of Dremel query's and its storage layout – it is closely tied to the underlying architecture of the system.

Load Balancing Concurrent work from Google [8] uses task speculation to achieve very low response times for high fanout online web services. Google's approach targets very short tasks (typically 1ms) with poor worst-case performance (99th percentile task runtime is 1 second), resulting in an approach that focuses on mitigating unpredictable variation in service times. Batch sampling instead focuses on minimizing queue imbalance; as described in §3.4, task speculation is orthogonal.

A variety of projects ([18, 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. As a result, the majority of effort is spend determining when to *re*schedule processes to balance load.

Load balancing has also been explored extensively in the theory community, as summarized by Mitzenmacher [16]. We revisit current approaches in a *parallel* setting and presents a system implementation based on those findings.

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 probing a random set of machines and placing tasks on the machines where tasks will be launched soonest. Analytical results show that for 99% of jobs in cluster of 4-core servers, batch sampling places all of the job's tasks on idle machines at up to 70% cluster load. Simulation confirms this result, and demonstrates that at up to 60% utilization, batch sampling performs with 4% of an optimal scheduler. Using Sparrow, a distributed scheduler that implements batch sampling, we demonstrated that batch sampling performs within 12% of optimal in a 100-node cluster running short TPC-H queries. Thus, as task durations continue to decrease, distributed scheduling using batch sampling presents a viable alternative to centralized schedulers.

References

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

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

[3] ANANTHANARAYANAN, G., GHODSI, A., SHENKER, S., AND STOICA, I. Why Let Resources Idle? Aggressive Cloning of Jobs with Dolly. In *HotCloud* (2012).

[4] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proc. OSDI* (2010).

[5] BRADLEY, D., CLAIR, T. S., FARRELLEE, M., GUO, Z., LIVNY, M., SFILIGOI, I., AND TANNENBAUM, T. An Update on the Scalability Limits of the Condor Batch System. *Journal of Physics: Conference Series 331*, 6 (2011).

[6] CASAVANT, T. L., AND KUHL, J. G. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Trans. Software Eng. 14* (February 1988), 141–154.

[7] COUNCIL, T. P. P. TPC Benchmarks. http://www.tpc.org/information/benchmarks.asp.

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

[9] ENGLE, C., LUPHER, A., XIN, R., ZAHARIA, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Shark: Fast Data Analysis Using Coarse-Grained Distributed Memory. In *Proc. SIGMOD* (2012).

[10] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A Platform For Fine-Grained Resource Sharing in the Data Center. In *NSDI* (2011).

[11] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed Data-Parallel Programs From Sequential Building Blocks. In *Proc. EuroSys* (2007).

[12] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proc. SOSP* (2009).

[13] JETTE, M. A., YOO, A. B., AND GRONDONA, M. Slurm: Simple linux utility for resource management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003* (2002), Springer-Verlag, pp. 44–60.

[14] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* (2010).

[15] MITZENMACHER, M. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.* (2001).

[16] MITZENMACHER, M. The Power of Two Random Choices: A Survey of Techniques and Results. In *Handbook of Randomized Computing*, S. Rajasekaran, P. Pardalos, J. Reif, and J. Rolim, Eds., vol. 1. Springer, 2001, pp. 255–312.

[17] MURTHY, A. C. The Next Generation of Apache MapReduce. http://tinyurl.com/7deh64l, February 2012.

[18] RUDOLPH, L., SLIVKIN-ALLALOUF, M., AND UPFAL, E. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In *Proc. ACM SPAA* (1991).

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

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

[21] WILLEBEEK-LEMAIR, M., AND REEVES, A. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems 4* (1993), 979–993.

[22] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Delay Scheduling: A Simple Technique For Achieving Locality and Fairness in Cluster Scheduling. In *Proc. EuroSys* (2010).

[23] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. NSDI* (2012).

[24] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. OSDI* (2008).