Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale

Xiaoqi Ren[1], Ganesh Ananthanarayanan[2], Adam Wierman[1], Minlan Yu[3]
[1]California Institute of Technology,
[2]Microsoft,
[3]University of Southern California,
{xren,adamw}@caltech.edu, ga@microsoft.com, minlanyu@usc.edu

ABSTRACT

As clusters continue to grow in size and complexity, providing *scalable and predictable* performance is an increasingly important challenge. A crucial roadblock to achieving *predictable* performance is stragglers, i.e., tasks that take significantly longer than expected to run. At this point, speculative execution has been widely adopted to mitigate the impact of stragglers. However, speculation mechanisms are designed and operated independently of job scheduling when, in fact, scheduling a speculative copy of a task has a direct impact on the resources available for other jobs. In this work, we present Hopper, a job scheduler that is speculation-aware, i.e., that integrates the tradeoffs associated with speculation into job scheduling decisions. We implement both centralized and decentralized prototypes of the Hopper scheduler and show that 50% (66%) improvements over state-of-the-art centralized (decentralized) schedulers and speculation strategies can be achieved through the *coordination* of scheduling and speculation.

CCS Concepts

•Networks → Cloud computing; •Computer systems organization → Distributed architectures;

Keywords

speculation; decentralized scheduling; straggler; fairness

1. INTRODUCTION

Data analytics frameworks have successfully realized the promise of "scaling out" by automatically composing user-submitted scripts into *jobs* of many parallel *tasks* and executing them on large clusters. However, as clusters increase in size and complexity, providing *scalable* and *predictable* performance is an important ongoing challenge for interactive analytics frameworks [2, 32]. Indeed, production clusters at Google and Microsoft [17, 23] acknowledge this as a prominent goal.

As the scale and complexity of clusters increase, hard-to-model systemic interactions that degrade the performance of tasks become common [12, 23]. Consequently, many tasks become "stragglers", i.e., running slower than expected, leading to significant *unpredictability* (and delay) in job completion times – tasks in Facebook's Hadoop cluster can run up to 8× slower than expected [12]. The most successful and widely deployed straggler mitigation solution is *speculation*, i.e., speculatively running extra copies of tasks that have become stragglers (or likely to), and then picking the earliest copy that finishes, e.g., [12, 14, 15, 24, 50]. Speculation is commonplace in production clusters, e.g., in our analysis of Facebook's Hadoop cluster, speculative tasks account for 25% of all tasks and 21% of resource usage.

Speculation is intrinsically intertwined with job scheduling because spawning a speculative copy of a task has a direct impact on the resources available for *other* jobs. Aggressive speculation can improve the performance of the job at hand but hurt the performance of other jobs. Despite this, speculation policies deployed today are all designed and operated *independently* of job scheduling; schedulers simply allocate slots to speculative copies in a "best-effort" fashion, e.g., [14, 15, 24, 36].

Coordinating speculation and scheduling decisions is an opportunity for significant performance improvement. However, achieving such coordination is challenging, particularly as schedulers themselves *scale* out. Schedulers are increasingly becoming *decentralized* in order to scale to hundreds of thousands of machines with each machine equipped with tens of compute slots for tasks. This helps them make millions of scheduling decisions per second, a requirement about two orders of magnitude beyond the (already highly-optimized) centralized schedulers, e.g., [10, 29, 49]. In decentralized designs

SIGCOMM '15, August 17 - 21, 2015, London, United Kingdom
 ISBN 978-1-4503-3542-3/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2785956.2787481

multiple schedulers operate *autonomously*, with each of them scheduling only a subset of the jobs, e.g., [19, 23, 36]. Thus, the coordination between speculation and scheduling must be achieved without maintaining central information about all the jobs.

Contribution of this paper: In this paper we present the design of the *first speculation-aware job scheduler*, Hopper, which dynamically allocates slots to jobs keeping in mind the speculation requirements necessary for *predictable* performance. Hopper incorporates a variety of factors such as data locality, estimates of task execution times, fairness, dependencies (DAGs) between tasks, etc. Further, Hopper is compatible with all current speculation algorithms and can operate as either a centralized or decentralized scheduler; achieving *scalability* by not requiring any central state.

The key insight behind Hopper is that a scheduler must anticipate the speculation requirements of jobs and dynamically allocate capacity depending on the *marginal* value (in terms of performance) of extra slots which are likely used for speculation. A novel observation that leads to the design of Hopper is that there is a sharp "threshold" in the marginal value of extra slots – an extra slot is always more beneficial for a job below its threshold than it is for any job above its threshold. The identification of such a threshold then allows Hopper to use different resource allocation strategies depending on whether the system capacity is such that all jobs can be allocated more slots than their threshold or not. This leads to a dynamic, adaptive, online scheduler that reacts to the current system load in a manner that appropriately weighs the value of speculation.

Importantly, the core components of Hopper can be *decentralized* effectively. The key challenge to avoiding the need to maintain a central state is the fact that stragglers create heavy-tailed task durations, e.g., see [12, 14, 25]. Hopper handles this by adopting a "power of many choices" viewpoint to approximate the global state, which is fundamentally more suited than the traditional "power of two choices" viewpoint due to the durations and frequency of stragglers.

To demonstrate the potential of Hopper, we have built three demonstration prototypes by augmenting the *centralized* scheduling frameworks Hadoop [3] (for batch jobs) and Spark [49] (for interactive jobs), and the *decentralized* framework Sparrow [36]. Hopper incorporates many practical features of jobs into its scheduling. Among others, it estimates the amount of *intermediate* data produced by the job and accounts for their pipelining between phases, integrates *data locality* requirements of tasks, and provides *fairness* guarantees.

We have evaluated our three prototypes on a 200 node private cluster using workloads derived from Facebook's and Microsoft Bing's production traces. The decentralized and centralized implementations of Hopper reduce the average job completion time by up to 66% and 50% compared to state-of-the-art scheduling and straggler mitigation techniques. The gains are consistent across common speculation algorithms (LATE [50], GRASS [14], and Mantri [15]), DAGs of tasks, and locality constraints, while providing fine-grained control on fairness. Importantly, the gains *do not* result from improving the speculation mechanisms but from improved *coordination* of scheduling and speculation decisions.

2. BACKGROUND & RELATED WORK

We begin by presenting a brief overview of existing cluster schedulers: how they allocate resources across jobs, both centralized and decentralized (§2.1), and how they handle straggling tasks (§2.2). This overview highlights the lack of coordination that currently exists between scheduling and straggler mitigation strategies such as speculation.

2.1 Cluster Schedulers

Job scheduling – allotting compute slots to jobs for their tasks – is a classic topic with a large body of work.

The most widely-used scheduling approach in clusters today is based on *fairness* which, without loss of generality, can be defined as equal sharing (or weighted sharing) of the available resources among jobs (or their users) [4, 26, 30, 45, 47]. Fairness, of course, comes with performance inefficiencies, e.g., [41, 48].

In contrast, the performance-optimal approach for job scheduling is *Shortest Remaining Processing Time (SRPT)*, which assigns slots to jobs in ascending order of their remaining duration (or, for simplicity, the remaining number of tasks). SRPT's optimality in both single [39] and multi-server [37] settings motivates a focus on prioritizing small jobs and has led to many schedulers such as [31, 33, 42].

The schedulers mentioned above are all centralized; however, motivated by scalability, many clusters are beginning to adopt *decentralized* schedulers, e.g., at Google [23], Apollo [17] at Microsoft, and the recently proposed Sparrow [36] scheduler. The scalability of decentralized designs allows schedulers to cope with growing cluster sizes and increasing parallelism of jobs (due to smaller tasks [34]), allowing them to scale to millions of scheduling decisions (for tasks) per second.

Importantly, the literature on cluster scheduling (both centralized and decentralized) ignores an important aspect of clusters: straggler mitigation via speculation. No current schedulers coordinate decisions with speculation mechanisms, while our analysis shows that speculative copies account for a sizeable fraction of all tasks in production clusters, e.g., in Facebook's Hadoop cluster, speculative tasks account for 25% of all tasks and 21% of resource usage.

2.2 Straggler Mitigation via Speculation

Dealing with *straggler* tasks, i.e., tasks that take significantly longer than expected to complete, is an important challenge for cluster schedulers, one that was called out in the original MapReduce paper [24], and a topic of significant subsequent research [12, 14, 15, 50].

Clusters already blacklist *problematic* machines (e.g., faulty disks or memory errors) and avoid scheduling tasks on them. However, despite blacklisting, stragglers occur frequently, often due to intrinsically complex causes such as IO contention, interference by periodic maintenance operations, and hardware behaviors which are hard to model and circumvent [12, 22, 35]. Straggler prevention based on comprehensive root-cause analyses is an open research challenge.

The most effective, and indeed the most widely deployed, technique has been *speculative* execution. Speculation techniques, monitor the progress of running tasks, compare them to the progress of completed tasks of the job, and spawn speculative copies for those progressing much slower, i.e., straggling. It is then a race between the original and speculative copies of the task and on completion of one, the other copies are killed.[1]

There is considerable (statistical and systemic) sophistication in speculation techniques, e.g., ensuring early detection of stragglers [15], predicting duration of new (and running) tasks [16], and picking lightly loaded machines to spawn speculative copies [50]. The techniques also take care to avoid speculation when a new copy is unlikely to benefit, e.g., when the single input source's machine is the cause behind the straggling [46].

Speculation has been highly effective in mitigating stragglers, bringing the ratio of the progress rates of the median task of a job to its slowest down from 8× (and 7×) to 1.08× (and 1.1×) in Facebook's production Hadoop cluster (and Bing's Dryad cluster).

Speculation has, to this point, been done independently of job scheduling. This is despite the fact that when a speculative task is scheduled it takes resources away from other jobs; thus there is an intrinsic tradeoff between scheduling speculative copies and scheduling new jobs. In this paper, we show that integrating these two via speculation-aware job scheduling can speed up jobs by considerably, even on average. Note that these gains are *not* due to improving the speculative execution techniques, but instead come purely from the integration between speculation and job scheduling decisions.

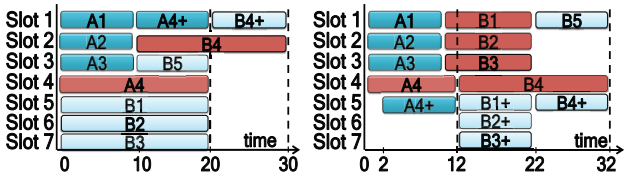
3. MOTIVATION

The previous section highlights that speculation and scheduling are currently designed and operated independently. Here, we illustrate the value of coordinated speculation and scheduling using simple examples.

3.1 Strawman Approaches

We first explore two baselines that characterize how scheduling and speculation interact today. In our examples we assume that stragglers can be detected after a task has run for 2 time units and that, at this point, a speculation is performed if the remaining running time (t_{rem}) is longer than the time to run a new copy (t_{new}). When the fastest copy of a task finishes, other running copies of the same task are killed. Note that while these examples have all jobs arrive at time 0, Hopper is designed to work in an online setting.

[1]Schedulers avoid checkpointing a straggling task's current output and spawning a new copy for just the remaining work due to the overheads and complexity of doing so. In general, even though the speculative copy is spawned on the expectation that it would be faster than the original, it is extremely hard to guarantee that in practice. Thus, both are allowed to run until the first completes.



(a) Best-effort Speculation. (b) Budgeted Speculation

Figure 1: **Combining SRPT scheduling and speculation for two jobs A (4 tasks) and B (5 tasks) on a 7-slot cluster. The + suffix indicates speculation. Copies of tasks that are killed are colored red.**

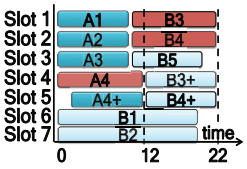


Figure 2: Hopper: **Completion time for jobs A and B are 12 and 22. The + suffix indicates speculation.**

A	A1	A2	A3	A4
t_{orig}	10	10	10	30
t_{new}	10	10	10	10

B	B1	B2	B3	B4	B5
t_{orig}	20	20	20	40	10
t_{new}	10	10	10	10	10

Table 1: **t_{orig} and t_{new} are durations of the original and speculative copies of each task.**

Best-Effort Speculation: A simple approach, which is also the most common in practice, is to treat speculative tasks the same as regular tasks. The job scheduler allocates resources for speculative tasks in a "best effort" manner, i.e., *whenever there is an open slot.*

Consider the example in Figure 1a with two jobs A (4 tasks) and B (5 tasks) that are scheduled using the SRPT policy. The scheduler has to wait until time 10 to find an open slot for the speculative copy of A4, despite detecting it was straggling at time 2.[2] Clearly, the scheduler can do better. If it had allocated a slot to A's speculative task at time 2 (instead of letting B use it), then job A's completion time would have reduced, without slowing job B (see Table 1 for task durations).

Note that similar inefficiencies occur under Fair scheduling in this example.

Budgeted Speculation: The main problem for best-effort speculation is a lack of available slots for speculation when needed. Thus, an alternative approach is to have the job scheduler *reserve* a fixed "budget" of slots for speculative tasks. Budgeting the right size of the resource pool for speculation, however, is challeng-

[2]At time 10, when A1 finishes, the job scheduler allocates the slot to job A because its remaining processing is smaller than job B's. Job A speculates task A4 because A4's $t_{rem} = t_{orig} - \text{currentTime} = 30 - 10 = 20 > t_{new} = 10$ (see Table 1).

ing because of time-varying straggler characteristics and fluctuating cluster utilizations. If the resource pool is too small, it may not be enough to immediately support all the tasks that need speculation. If the pool is too large, resource are left idle.

Figure 1b illustrates budgeted speculation with three slots (slot $5-7$) being reserved for speculation. This, unfortunately, leads to slots 6 and 7 lying fallow from time 0 to 12. If the wasted slot had been used to run a new task, say B1, then job B's completion time would have been reduced. It is easy to see that similar wastage of slots occurs with the Fair scheduler. Note that reserving one or two instead of three slots will not solve the problem, since three speculative copies are required to run simultaneously at a later time.

3.2 Challenges in Coordination

In contrast to the two baselines discussed above, Figure 2 shows the benefit of coordinated decision making.

At time $0-10$, we allocate 1 extra slot to job A (for a total of 5 slots), thus allowing it to speculate task A4 promptly. After time 10, we can *dynamically reallocate* the slots to job B. This reduces the average completion time compared to both the budgeted and best-effort strategies. The joint design budgeted slot 5 until time 2 but after task A4 finished, it used all the slots.

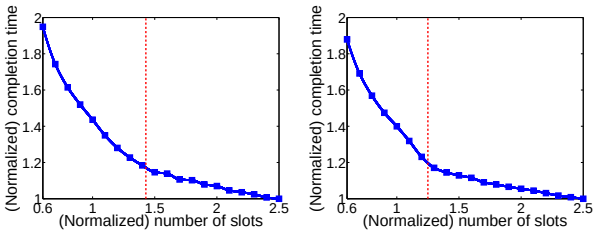
Doing such dynamic allocation is already challenging in a centralized environment, and it becomes more so in a decentralized setting. In particular, decentralized speculation-aware scheduling has additional constraints. Since the schedulers are autonomous, there is no central state and thus, no scheduler has complete information about all the jobs in the cluster. Further, every scheduler has information about only a subset of the cluster (the machines it probed). Since decentralization is mainly critical for interactive jobs (sub-second or a few seconds), time-consuming gossiping between schedulers is infeasible. Finally, running *all* the schedulers on one multi-core machine cramps that machine and caps scalability, the original drawback they aim to alleviate.

In the above example, this means making the allocation as in Figure 2 when jobs A and B autonomously schedule their tasks without complete knowledge of utilizations of the slots or even each other's existence.

Thus, the challenges for speculation-aware job scheduling are: (i) *dynamically* allocating/budgeting slots for speculation based on the distribution of stragglers and cluster utilization while being (approximately) *fair* and, in decentralized settings, (ii) using *incomplete information* about the machines and jobs in the cluster.

4. Hopper: SPECULATION-AWARE SCHEDULING

The central question in the design of a speculation-aware job scheduler is how to dynamically (online) balance the slots used by speculative and original copies of tasks across jobs. A given job will complete more quickly if it is allowed to do more speculation, but this comes at the expense of other jobs in the system.



(a) $\beta = 1.4$ (b) $\beta = 1.6$

Figure 3: **The impact of number of slots on single job performance. The number of slots is normalized by job size (number of tasks within the job). β is the Pareto shape parameter for the task size distribution. (In our traces $1 < \beta < 2$.) The red vertical line shows the threshold point.**

Hopper's design is based on the insight that the balance between speculative and original tasks must dynamically depend on cluster utilization. The design guidelines that come out of this insight are supported by theoretical analysis in a simple model [8]. We omit the analytic support due to space constraints and focus on providing an intuitive justification for the design choices. Pseudocode 1 shows the basic structure.[3]

We begin our discussion of speculation-aware job scheduling by introducing the design features of Hopper in a centralized setting. We focus on single-phased jobs in §4.1, and then generalize the design to incorporate DAGs of tasks (§4.2), data locality (§4.4), and fairness (§4.3). Finally, in §5, we discuss how to adapt the design to a decentralized setting.

4.1 Dynamic Resource Allocation

The examples in §3 illustrate the value of *dynamic* allocation of slots for speculation. Our analysis indicates that this dynamic allocation can be separated into two regimes: whether the cluster is in "high" or "low" load.

The distinction between these two regimes follows from the behavior of the marginal return (in terms of performance) that jobs receive from being allocated slots. It is perhaps natural to expect that the performance of a job will always improve when it is given additional slots (because these can be used for additional speculative copies) and that the value of additional slots has a decreasing marginal return (because an extra slot is more valuable when the job is given few slots than when the job already has many slots). However, surprisingly, a novel observation that leads to the design of Hopper is that the marginal return of an extra slot has a sharp threshold (a.k.a., knee) where, below the threshold, the marginal return is large and (nearly) constant

[3]For ease of exposition, Pseudocode 1 considers the (online) allocation of all slots to jobs present at time t. Of course, in the implementation, slots are allocated as they become available. See Pseudocode 2 and 3 for more details.

```
1: procedure HOPPER(⟨Job⟩ J(t), int S, float β)
       totalVirtualSizes ← 0
2:     for each Job j in J(t) do
           j.V(t) = (2/β) j.T_rem
                        ▷ j.T_rem: remaining number of tasks
                                    ▷ j.V(t): virtual job size
           totalVirtualSizes += j.V(t)
3:     SortAscending(J(t), V(t))
4:     if S < totalVirtualSizes then
5:         for each Job j in J(t) do
               j.slots ← ⌊min(S, j.V(t))⌋
               S ← max(S − j.slots, 0)
6:     else
7:         for each Job j in J(t) do
               j.slots ← ⌊(j.V(t)/totalVirtualSizes) × S⌋
```

Pseudocode 1: Hopper **(centralized) allocating S slots to the set of jobs present at time t, $J(t)$, with task distribution parameter β.**

and, above the threshold, the marginal return is small and decreasing.

Figure 3 illustrates this threshold using a simulation of a sample job with 200 tasks (with Pareto sizes, common in production traces) and LATE [50] speculation when assigned various numbers of slots. Crucially, there is a marked change in slope beyond the vertical dashed line, indicating the change in the marginal value of a slot. Note, that such a threshold exists for different job sizes, speculation algorithms, etc. Further, in the context of a simple model, we can prove the existence of a sharp threshold [8].

The most important consequence of the discussion above is that it is desirable to ensure every job is allocated enough slots to reach the threshold (if possible) before giving any job slots beyond this threshold. Thus, we refer to this threshold as the "desired (minimum) allocation" for a job or simply the "virtual job size".

Guideline 1. *It is better to give resources to a job that has not reached the desired (minimum) allocation than a job that has already reached the point.*

This guideline yields the key bifurcation in the Hopper design, as illustrated in line 4 of Pseudocode 1. Additionally, it highlights that there are three important design questions, which we address in the following sections: (i) How can we determine the desired allocation (virtual size) of a job? (ii) How should slots be allocated when there are not enough to give each job its desired allocation, i.e., when the cluster is highly utilized? (iii) How should slots be allocated when there are more than enough to give each job its desired allocation, i.e., when the cluster is lightly utilized?

(i) Determining the virtual size of a job

Determination of the "desired (minimum) allocation", a.k.a., the "virtual" size, of a job is crucial to determining which regime the system is in, and thus how slots should be allocated among jobs. While the virtual job size is learned empirically by Hopper through measurements of the threshold point during operation, it is important to point out that it is also possible to derive a useful static rule of thumb analytically, which can give intuition for the design structure.

In particular, the task durations in production traces (e.g., Facebook and Bing traces described in §7) typically follow a heavy-tailed Pareto distribution, where the Pareto tail parameter β (which is often $1 < \beta < 2$) represents the likelihood of stragglers [12, 13, 14, 25]. Roughly, smaller β means that stragglers are more damaging, i.e., if a task has already run for some time, there is higher likelihood of the task running longer.

Given the assumption of Pareto task durations, we can prove analytically (in a simple model) that the threshold point defining the "desired (minimum) allocation" is $\max(2/\beta, 1)$, which corresponds exactly to the vertical line in Figure 3 (see [8] for details). While we show only two examples here, the estimate this provides is robust across varying number of tasks, β, etc. [4]

Thus, we formally define the "virtual job size" $V_i(t)$ for job i at any time t, as its number of remaining tasks ($T_i(t)$) multiplied by $2/\beta$ (since $\beta < 2$ in our traces), i.e., $V_i(t) = \frac{2}{\beta} T_i(t)$. This virtual job size determines which regime the scheduler should use; see line 2 in Pseudocode 1. In practice, since β may vary over time, it is learned online by Hopper (see §7) making it adaptive to different threshold points as in Figure 3.

(ii) Allocation when the cluster is highly utilized

When there are not enough slots to assign every job its virtual size, we need to decide how to distribute this "deficiency" among the jobs. The scheduler could either spread the deficiency across all jobs, giving them all less opportunity for speculation, or satisfy as many jobs as possible with allocations equaling their virtual sizes.

Hopper does the latter. Specifically, Hopper processes jobs in ascending order of their virtual sizes $V_i(t)$, giving each job its desired (minimum) allocation until all the slots are exhausted (see lines $3-5$ in Pseudocode 1). This choice is in the spirit of SRPT, and is motivated both by the optimality of SRPT and the decreasing marginal return of additional slots, which magnifies the value of SRPT. Additionally, our theoretical analysis (in [8]) shows the optimality of this choice in the context of a simple model.

Guideline 2. *At all points in time, if there are not enough slots for every job to get its desired (minimum) allocation, i.e., a number of slots equal to its virtual size, then slots should be dedicated to the smallest jobs and each should be given a number of slots equal to its virtual size.*

[4] We make the simplifying assumption that task durations of *each job* are also Pareto distributed. A somewhat surprising aspect given the typical values of β ($1 < \beta < 2$) is that even when so many slots are allocated for redundant speculative copies, faster "clearing" of tasks is overall beneficial.

Note that prioritization of small jobs may lead to unfairness for larger jobs, an issue we address shortly in §4.3.

(iii) Allocation when the cluster is lightly utilized

At times when there are more slots in the cluster than the sum of the virtual sizes of jobs, we have slots left over even after allocating every job its virtual size. The scheduler's options for dividing the extra capacity are to either split the slots across jobs, or give all the extra slots to a few jobs in order to complete them quickly.

In contrast to the high utilization setting, in this situation Hopper allocates slots proportionally to the virtual job sizes, i.e., every job i receives $(V_i(t)/\sum_j V_j(t))S$ slots, where S is the number of slots available in the system and $V_i(t)$ is the virtual size; see line 7 in Pseudocode 1. Note that this is, in a sense, the opposite of the prioritization according to SRPT.

The motivation for this design is as follows. Given that all jobs are already receiving their (minimum) desired level of speculation, scheduling is less important than speculation. Thus, prioritization of small jobs is not crucial, and the goal should be to extract the maximum value from speculation. Since stragglers are more likely to occur in larger jobs (stragglers occur in proportion to the number of tasks in a job, on average[5]), the marginal improvement in performance due to an additional slot is proportionally higher for large jobs. Thus, they should get prioritization in proportion to their size when allocating the extra slots. Our analytic work in [8] highlights that this allocation is indeed optimal in a simple model.

Guideline 3. *At all points in time, if there are enough slots to give every job its desired (minimum) allocation, then, the slots should be shared "proportionally" to the virtual sizes of the jobs.*

Since the guidelines specify allocations at the granularity of every job, it is easy to cope with any fluctuations in cluster load (say, from lightly to highly utilized) in an online system.

4.2 Incorporating DAGs of Tasks

The discussion to this point has focused on single-phased jobs. In practice, many jobs are defined by multiple-phased DAGs, where the phases are typically *pipelined*. That is, downstream tasks do not wait for *all* the upstream tasks to finish but read the upstream outputs as the tasks finish, e.g., [6]. Pipelining is beneficial because the upstream tasks are typically bottlenecked on other *non-overlapping* resources (CPU, memory), while the reading downstream takes network resources. The additional complexity DAGs create for our guidelines is the need to balance the gains due to overlapping network utilization with the improvements that come from favoring upstream phases with fewer remaining tasks.

We integrate this tradeoff into Hopper using a weighting factor, α per job, set to be the ratio of remaining work in the downstream phase's network transfer to the remaining work in the upstream phase. Specifically, α favors jobs with higher remaining communication and lower remaining tasks in the current phase. The exact details of estimating α are deferred to §6.3.

Given the weighting factor α, there are two key adjustments that we make to the guidelines discussed so far. First, in Guideline 2, the prioritization of jobs based on the virtual size $V_i(t)$ is replaced by a prioritization based on $\max\{V_i(t), V_i'(t)\}$, where $V_i(t)$ is the virtual remaining number of tasks in the current phase and $V_i'(t)$ is the virtual remaining work in communication in the downstream phase.[6] Second, we redefine the virtual size itself as $V_i(t) = \frac{2}{\beta}T_i(t)\sqrt{\alpha_i}$. This form follows from the analysis in [8] and is similar in spirit to the optimality of square-root proportionality in load balancing across heterogeneous servers [21].

For DAGs that are not strict chains, but are wide and "bushy", we calculate α by summing over all the running and their respective downstream phases.

4.3 Incorporating Fairness

While fairness is an important constraint in clusters, conversations with data center operators reveal that it is not an absolute requirement. Thus, we relax the notion of fairness currently employed by cluster schedulers, e.g., [47], which enforce that if there are $N(t)$ active jobs and S available slots at time t, then each job is assigned $S/N(t)$ slots.

Specifically, to allow some flexibility while still tightly controlling unfairness, we define a notion of *approximate* fairness as follows. We say that a scheduler is ϵ-fair if it guarantees that every job receives at least $(1-\epsilon)S/N(t)$ slots at *all* times t. The fairness knob $\epsilon \to 0$ indicates absolute fairness while $\epsilon \to 1$ focuses on performance.

Hopper can be adjusted to guarantee ϵ-fairness in a very straightforward manner. In particular, if a job receives slots less than its fair share, i.e., fewer than $(1-\epsilon)S/N(t)$ slots, the job's capacity assignment is increased to $(1-\epsilon)S/N(t)$. Next, the remaining slots are allocated to the remaining jobs according to Guidelines 2 or 3, as appropriate. Note that this is a form of projection from the original (unfair) allocation into the feasible set of allocations defined by the fairness constraints.

Our experimental results (§7.3) highlight that even at moderate values of ϵ, *nearly all jobs finish faster than they would have under fair scheduling.* This fact, though initially surprising, is similar to the conclusions

[5]Machines in the cluster are equally likely to cause a straggler [12]; known problematic machines are already blacklisted (see §2).

[6]Results in [31] show that picking the $\max\{T_i(t), T_i'(t)\}$ is 2-speed optimal for completion times when stragglers are not considered.

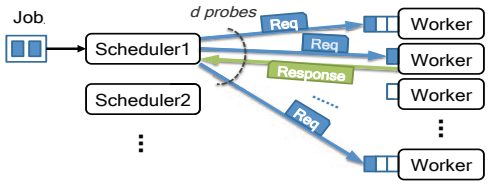


Figure 4: **Decentralized scheduling architecture.**

about SRPT-like policies. Despite being intuitively unfair to large job sizes, it in fact improves the average response time of every job size (when job sizes are heavy-tailed) compared to fair schedulers [28, 43, 44].

4.4 Incorporating Data Locality

As such, the guidelines presented does not consider data locality [11, 48] in the scheduling of tasks. Tasks reading their data from remote machines over the network run slower. In addition, such remote reads also increase contention with other intermediate tasks (like reduce tasks) that are bound to read over the network.

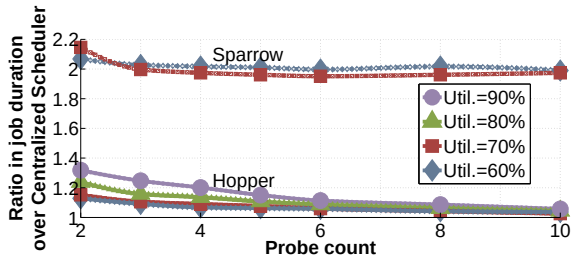
We devise a simple relaxation approach for balancing adherence to our guidelines and locality. Specifically, we adjust the ordering of jobs in Guideline 2 to include information about locality. Instead of allotting slots to the jobs with the smallest virtual sizes, we allow for picking any of the smallest k% of jobs whose tasks can run with data locality on the available slots. In practice, a small value of k ($\leq 5\%$) suffices due to high churn in task completions and slot availabilities (§7.4).

5. DECENTRALIZED Hopper

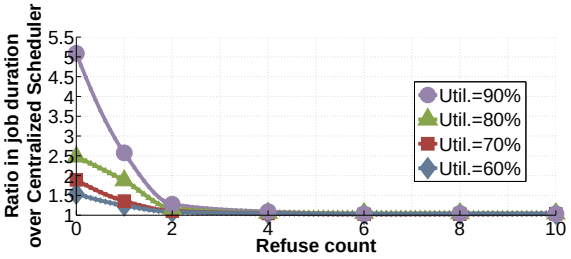
In this section, we adapt the guidelines described in §4 to design a decentralized (online) scheduler. Decentralized schedulers are increasingly prominent as cluster sizes grow. As we explain in this section, a key benefit of our guidelines in §4 is that they can be decentralized with little performance loss.

Decentralized schedulers, like the recently proposed Sparrow [36], broadly adopt the following design (see Figure 4). There are multiple independent *schedulers* each of which is responsible for scheduling one or a subset of jobs; for simplicity, a single job never spans across schedulers. Every scheduler assigns the tasks of its jobs to machines in the cluster (referred to as *workers*) that executes the tasks. The architecture allows for an incoming job to be assigned to any of the available schedulers, while also seamlessly allowing new schedulers to be dynamically spawned.

A scheduler first pushes *reservation requests* for its tasks to workers; each request contains the identifier of the scheduler placing the request along with the remaining number of unscheduled tasks in the job. When a worker is vacant, it pulls a task from the corresponding scheduler based on the reservation requests in its waiting queue. In this framework, workers decide which job's task to run and the scheduler for the corresponding job decides which task to run within the chosen job.



(a) Number of probes, d



(b) Number of refusals

Figure 5: **The impact of number of probes and number of refusals on Hopper's performance.**

This decoupling naturally facilitates the design of Hopper.

Though we adopt an overall design structure similar to Sparrow for the decentralization of Hopper, it is important to note that Hopper's design is fundamentally different because it integrates straggler mitigation based on the guidelines behind Hopper introduced in §4.

Decentralizing Hopper involves the following steps: approximating worker-wide information at each scheduler (§5.1), deciding if the number of slots are constrained (§5.2), and calculating virtual sizes (§5.3).

5.1 Power of Many Choices

Decentralized schedulers have to *approximate* the global state of the cluster – the states of all the workers – since they are unaware of other jobs in the system. A common way to accomplish this is via the "power of two choices" [38]. This celebrated and widely used result highlights that, in many cases, one nearly matches the performance of a centralized implementation by querying two workers for their queue lengths, and choosing the shorter of the queues. In fact, this intuition underlies the design of Sparrow as well, which combines the idea with a form of "late binding"; schedulers send reservation requests for every task to two workers and then let workers pull a task from the corresponding scheduler when they have a free slot. We adopt "late binding", as used in Sparrow, but replace the "power of two choices" with the "power of many choices".

The reason for this change is that the effectiveness of the "power of two choices" relies on having *light-tailed task size distributions*. The existence of stragglers means that, in practice, task durations are heavy-tailed, e.g., [12, 14, 25]. Recent theoretical results have proven

that, when task sizes are heavy-tailed, probing $d > 2$ choices can provide orders-of-magnitude improvements [18]. The value in using $d > 2$ comes from the fact that large tasks, which are more likely under heavy-tailed distributions, can cause considerable backing up of worker queues. Two choices may not be enough to avoid such backed-up queues, given the high frequency of straggling tasks. More specifically, $d > 2$ allows the schedulers to have a view of the jobs that is closer to the global view.

We use simulations in Figure 5a to highlight the benefit of using $d > 2$ probing choices in Hopper and to contrast this benefit with Sparrow, which relies on the power of two choices. Our simulation considers a cluster of 50 schedulers and 10,000 workers and jobs with Pareto distributed ($\beta = 1.5$) task sizes. Job performance with decentralized Hopper is within just 15% of the centralized scheduler; the difference plateaus beyond $d = 4$. Note that Sparrow (which does not coordinate scheduling and speculation) is $> 100\%$ off for medium utilizations and even further off for high utilizations (not shown on the figure in order to keep the scale visible). Further, workers in Sparrow pick tasks in their waiting queues in a FCFS fashion. The lack of coordination between scheduling and speculation results in a long waiting time for speculative copies in the queues which diminishes the benefits of multiple probes. Thus parrow cannot extract the same benefit Hopper has from using more than two probes. Of course, these are rough estimates since the simulations do not capture overheads due to increased message processing, which are included in the evaluations in §7.

5.2 Is the system capacity constrained?

In the decentralized setting workers implement our scheduling guidelines. Recall that Guideline 2 or Guideline 3 is applied depending on whether the system is constrained for slots or not. Thus, determining which to follow necessitates comparing the sum of virtual sizes of all the jobs and the number of slots in the cluster, which is trivial in a centralized scheduler but requires communication in an decentralized setting.

To keep overheads low, we avoid costly gossiping protocols among schedulers regarding their states. Instead, we use the following adaptive approach. Workers start with the conservative assumption that the system is capacity constrained (this avoids overloading the system with speculative copies), and thus each worker implements Guideline 2, i.e., enforces an SRPT priority on its queue. Specifically, when a worker is idle, it sends a *refusable* response to the scheduler corresponding to the reservation request of the job it chooses from its queue. However, since the scheduler queues many more reservation requests than tasks, it is possible that its tasks may have all been scheduled (with respect to virtual sizes). A refusable response allows the scheduler to refuse sending any new task for the job if the job's tasks are all already scheduled to the desired speculation level

```
procedure ResponseProcessing(Response response )
    Job j ← response.job
    if response.type = non-refusable then
        Accept()
    else
        if (j.current_occupied < j.virtual_size) Accept ()
        else Refuse()
```

Pseudocode 2: **Scheduler Methods.**

```
procedure Response(⟨Job⟩ J, int refused_count)
        ▷ J: list of jobs in queue of the worker excluding
    already refused jobs
    if refused_count ≥ refusal_threshold then
        j ← J.PickAtRandom()
        SendResponse(j, non-refusable)
    else
        j ← J. min(virtual_size)
        SendResponse(j, refusable)
```

Pseudocode 3: **Worker: choosing the next task to schedule.**

(ResponseProcessing in Pseudocode 2). In its refusal, it sends information about the job with the smallest virtual size in its list which still has unscheduled tasks (if such an "unsatisfied" job exists).

Subsequently, the worker sends a refusable response to the scheduler corresponding to second smallest job in its queue, and so forth till it gets a threshold number of refusals. Note that the worker avoids probing the same scheduler more than once. Several consecutive refusals from schedulers without information about any unsatisfied jobs suggests that the system is not capacity constrained. At that point, it switches to implementing Guideline 3. Once it is following Guideline 3, the worker randomly picks a job from the waiting queue based on the distribution of job virtual sizes. If there are still unsatisfied jobs at the end of the refusals, the worker sends a *non-refusable* response (which cannot be refused) to the scheduler whose unsatisfied job is the smallest. Pseudocode 3 explains the Response method.

The higher the threshold for refusals, the better the view of the schedulers for the worker. Our simulations (with 50 schedulers and 10,000 workers) in Figure 5b show that performance with two or three refusals is within $10\% - 15\%$ of the centralized scheduler.

5.3 Updating Virtual Job Sizes

Computing the remaining virtual job size at a scheduler is straightforward. However, since the remaining virtual size of a job changes as tasks complete, virtual sizes need to be updated dynamically. Updating virtual sizes accurately at the workers that have queued reservations for tasks of this job would require frequent message exchanges between workers and schedulers, which would create significant overhead in communication and processing of messages. So, our approach is to piggyback updates for virtual sizes on other communication messages that are anyway necessary between a scheduler and a worker (e.g., schedulers send-

ing reservation requests for new jobs, workers sending responses to probe system state and ask for new tasks). While this introduces a slight error in the virtual remaining sizes, our evaluation shows that the approximation provided by this approach is enough for the gains associated with Hopper.

Crucially, the calculation of virtual sizes is heavily impacted by the job specifics. Job specific properties of the job DAG and the likelihood of stragglers are captured through α and β, respectively, which are learned online. Note that jobs from different applications may have heterogeneous α and β.

6. IMPLEMENTATION OVERVIEW

We now give an overview of the implementation of Hopper in decentralized and centralized settings.

6.1 Decentralized Implementation

Our decentralized implementation uses the Sparrow [36] framework, which consists of many schedulers and workers (one each on every machine) [9]. Arbitrarily many schedulers can operate concurrently; though we use 10 in our experiments. Schedulers allow submissions of jobs using Thrift RPCs [1].

A job is broken into a set of tasks with their dependencies (DAG), binaries and locality preferences. The scheduler places requests at the workers for its tasks; if a task has locality constraints, its requests are only placed on the workers meeting its constraints [13, 40, 49]. The workers talk to the client executor processes (e.g., Spark executor). The executor processes are responsible for executing task binaries and are long-lived to avoid startup overheads (see [36] for a more detailed explanation).

Our implementation modifies the scheduler as well as the worker. The workers implement the core of the guidelines in §4 – determining if the system is slot-constrained and accordingly prioritizing jobs as per their virtual sizes. This required modifying the FIFO queue at the worker in Sparrow to allow for custom ordering of the queued requests. The worker, nonetheless, augments its local view by coordinating with the scheduler. This involved modifying the "late binding" mechanism both at the worker and scheduler. The worker, when it has a free slot, works with the scheduler in picking the next task (using Pseudocode 3). The scheduler deals with a response from the worker as per Pseudocode 2.

Even after *all* the job's tasks have been scheduled (including its virtual size), the job scheduler does not "cancel" its pending requests; there will be additional pending requests with any probe ratio over one. Thus, if the system is not slot-constrained, it would be able to use more slots (as per Guideline 3).

In our decentralized implementation, for tasks in the input phase (e.g., map phase), when the number of probes exceeds the number of data replicas, we queue up the additional requests at randomly chosen machines. Consequently, these tasks *may* run without data locality, and our results in §7 include such loss in locality.

6.2 Centralized Implementation

We implement Hopper inside two centralized frameworks: Hadoop YARN (version 2.3) and Spark (version 0.7.3). Hadoop jobs read data from HDFS [5] while Spark jobs read from in-memory RDDs.

Briefly, these frameworks implement two level scheduling where a central *resource manager* assigns slots to the different *job managers*. When a job is submitted to the resource manager, a job manager is started on one of the machines, that then executes the job's DAG of tasks. The job manager negotiates with the resource manager for resources for its tasks.

We built Hopper as a scheduling plug-in module to the resource manager. This makes the frameworks use our design to allocate slots to the job managers. We also piggybacked on the communication protocol between the job manager and resource manager to communicate the intermediate data produced and read by the phases of the job to vary α accordingly; locality and other preferences are already communicated between them.

6.3 Estimating Intermediate Data Sizes

Recall from §4.2 that our scheduling guidelines recommend scaling every job's allocation by $\sqrt{\alpha}$ in the case of DAGs. The purpose of the scaling is to capture pipelining of the reading of upstream tasks' outputs.

The key to calculating α is estimating the size of the *intermediate* output produced by tasks. Unlike the job's input size, intermediate data sizes are not known upfront. We predict intermediate data sizes based on similar jobs in the past. Clusters typically have many recurring jobs that execute periodically as newer data streams in, and produce intermediate data of similar sizes.

Our simple approach to estimating α works sufficiently well for our evaluations (accuracy of 92%, on average). However, we realize that workloads without many multi-waved or recurring jobs and without tasks whose duration is dictated by their input sizes, need more sophisticated models of task executions.

7. EVALUATION

We evaluate our prototypes of Hopper – with both decentralized and centralized scheduling – on a 200 machine cluster. We focus on the overall gains of the decentralized prototype of Hopper in §7.2 and evaluate the design choices that led to Hopper in §7.3. Then, in §7.4 we evaluate the gains with Hopper in a centralized scheduler in order to highlight the value of coordinating scheduling and speculation. The key highlights are:

1. Hopper's decentralized prototype improves the average job duration by up to 66% compared to an aggressive decentralized baseline that combines Sparrow with SRPT (§7.2).

2. Hopper ensures that only 4% of jobs slow down compared to Fair scheduling, and jobs which do slow down do so by $\leq 5\%$ (§7.3).

3. Centralized Hopper improves job completion times by 50% compared to centralized SRPT (§7.4).

7.1 Setup

Cluster Deployment: We deploy our prototypes on a 200-node private cluster. Each machine has 16 cores, 34GB of memory, 1Gbps network and 4 disks. The machines are connected using a network with no oversubscription.[7]

Workload: Our evaluation runs jobs in traces from Facebook's production Hadoop [3] cluster (3,500 machines) and Microsoft Bing's Dryad cluster (O(1000) machines) from Oct-Dec 2012. The traces consist of a mix of experimental and production jobs. Their tasks have diverse resource demands of CPU, memory and IO, varying by a factor of 24× (refer to [27] for detailed quantification). We retain the inter-arrival times of jobs, their input sizes and number of tasks, resource demands, and job DAGs of tasks. Job sizes follow a heavy-tailed distribution (quantified in detail in [12]). Each experiment is a replay of a representative 6 hour slice from the trace. It is repeated five times and we report the median.

To evaluate our prototype of decentralized Hopper, we use in-memory Spark [49] jobs. These jobs are typical of interactive analytics whose tasks vary from sub-second durations to a few seconds. Since the performance of any decentralized scheduler depends on the cluster utilization, we speed-up the trace appropriately, and evaluate on (average) utilizations between 60% and 90%, consistent with Sparrow [36].

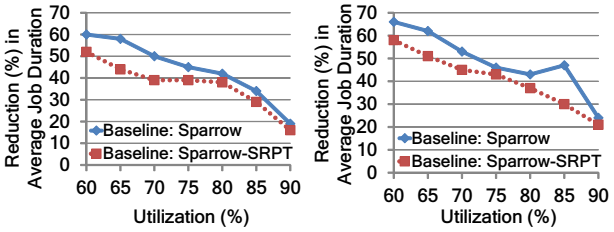
Stragglers: The stragglers in our experiments are those that occur *naturally*, i.e., not injected via any model of a probability distribution or via statistics gathered from the Facebook and Bing clusters. Importantly, the frequency and lengths of stragglers observed in our evaluations are consistent with prior studies, e.g., [14, 15, 50]. While Hopper's focus is not on improving straggler mitigation algorithms, our experiments certainly serve to emphasize the importance of such mitigation.

Baseline: We compare decentralized Hopper to Sparrot-SRPT, an augmented version of Sparrow [36]. Like Sparrow, it performs decentralized scheduling using a "batched" power-of-two choices. In addition, it also includes an SRPT heuristic. In short, when a worker has a slot free, it picks the task of the job that has the least unfinished tasks (instead of the standard FIFO ordering in Sparrow). Finally, we combine Sparrow with LATE [50] using "best effort" speculation (§3); we do not consider "budgeted" speculation due to the difficulty of picking a fixed budget.

The combination of Sparrow-SRPT and LATE performs strictly better than Sparrow, and serves as an aggressive baseline. Our improvements over this aggressive benchmark highlight the importance of coordinating scheduling and speculation.

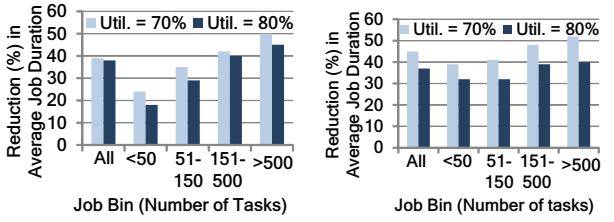
We compare centralized Hopper to a centralized SRPT scheduler with LATE speculation. Again, this is an aggressive baseline since it sacrifices fairness for performance. Thus, improvements can be interpreted as coming solely from better coordination of scheduling and speculation.

[7]Results with a 10Gbps network are qualitatively similar.



(a) Facebook (b) Bing

Figure 6: Hopper's gains with cluster utilization.



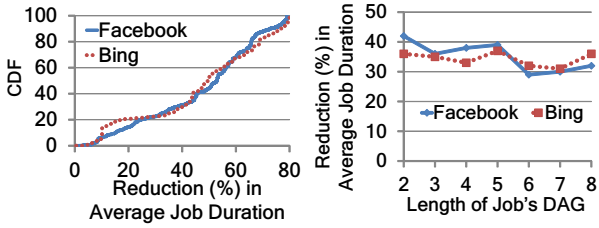
(a) Facebook (b) Bing

Figure 7: Hopper's gains by job bins over Sparrow-SRPT.

7.2 Decentralized Hopper's Improvements

In our experiments, unless otherwise stated, we set the fairness allowance ϵ as 10%, probe ratio as 4 and speculation algorithm in every job to be LATE [50]. Our estimation of α (§6.3) has an accuracy of 92% on average. As the workload executes, we also continually fit the parameter β of task durations based on the completed tasks (including stragglers); the error in β's estimate falls to $\leq 5\%$ after just 6% of the jobs have executed.

Overall Gains: Figure 6 plots Hopper's gains for varying utilizations, compared to stock Sparrow and Sparrow-SRPT. Jobs, overall, speedup by $50\% - 60\%$ at utilization of 60%. The gains compared to Sparrow are marginally better than Sparrow-SRPT. When the utilization goes over 80%, Hopper's gains compared to both are similar. An interesting point is that Hopper's gains with the Bing workload in Figure 6b are a touch higher (difference of 7%), perhaps due to the larger difference in job sizes between small and large jobs, allowing more opportunity for Hopper. Gains fall to $< 20\%$ when utilization is high ($\geq 80\%$), naturally because there is not much room for any optimization at that occupancy.



(a) Distribution (b) DAG

Figure 8: **(a) CDF of Hopper's gains, and (b) gains as the length of the job's DAG varies; both at** 60% **utilization.**

While not plotted, gains at utilizations $\leq$ 30% are no more than 14%. Expectedly, at such low utilizations, there is little requirement for smarter speculation or probing.

Note that the above utilizations are on average and there is considerable variation. At 80% average utilization, Hopper allocates 53% of jobs using Guideline 2 (high utilization) and the remaining 47% of jobs using Guideline 3 (low utilization). This indicates that 53% of jobs in the experimental run arrived such that the cluster did not have enough slots to allocate every job its virtual size.

The results so far highlight that Sparrow-SRPT is a more aggressive baseline than Sparrow, and so we compare only to it for the rest of our evaluation.

Job Bins: Figure 7 dices the gains by job size (number of tasks). Gains for small jobs are less compared to large jobs. This is expected given that our baseline of Sparrow-SRPT already favors the small jobs. Nonetheless, Hopper's smart allocation of speculative slots offers 18% – 32% improvement. Gains for large jobs, in contrast, are over 50%. This not only shows that there is sufficient room for the large jobs despite favoring small jobs (due to the heavy-tailed distribution of job sizes [12, 13]) but also that the value of deciding between speculative tasks and unscheduled tasks of other jobs increases with the number of tasks in the job. With trends of smaller tasks and hence, larger number of tasks per job [34], Hopper's allocation becomes important.

Distribution of Gains: Figure 8a plots the distribution of gains across jobs. While the median gains are just higher than the average, there are $>$ 70% gains at higher percentiles. Encouragingly, gains even at the 10^{th} percentile are 15% and 10%, which shows Hopper's ability to improve even worse case performance.

DAG of Tasks: The scripts in our Facebook (Hive scripts [7]) and Bing (Scope [20]) workloads produce DAGs of tasks which often pipeline data transfers of downstream phases with upstream tasks [6]. The communication patterns in the DAGs are varied (e.g., all-to-all, many-to-one etc.) and thus the results also serve to underscore Hopper's generality. As Figure 8b shows, Hopper's gains hold across DAG lengths.

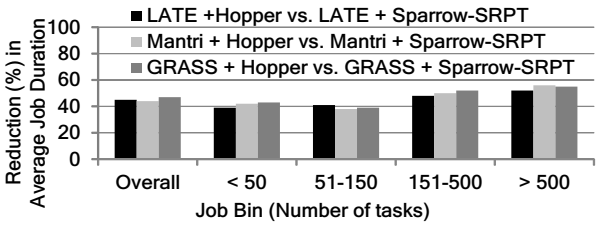


Figure 9: **Hopper's results are independent of the straggler mitigation strategy.**

Speculation Algorithm: We now experimentally evaluate Hopper's performance with different speculation mechanisms. LATE [50] is deployed in Facebook's clusters, Mantri [15] is in operation in Microsoft Bing, and GRASS citegrass is a recently reported straggler mitigation system that was demonstrated to perform near-optimal speculation. Our experiments still use Sparrow-SRPT as the baseline but pair with the different straggler mitigation algorithms. Figure 9 plots the results.

While the earlier results were achieved in conjunction with LATE, a remarkable point about Figure 9 is the similarity in gains even with Mantri and GRASS. This indicates that as long as the straggler mitigation algorithms are aggressive in asking for speculative copies, Hopper will appropriately balance speculation and scheduling. Overall, it emphasizes the aspect that resource allocation *across* jobs (with speculation) has a higher performance value than straggler mitigation *within* jobs.

7.3 Evaluating Hopper's Design Decisions

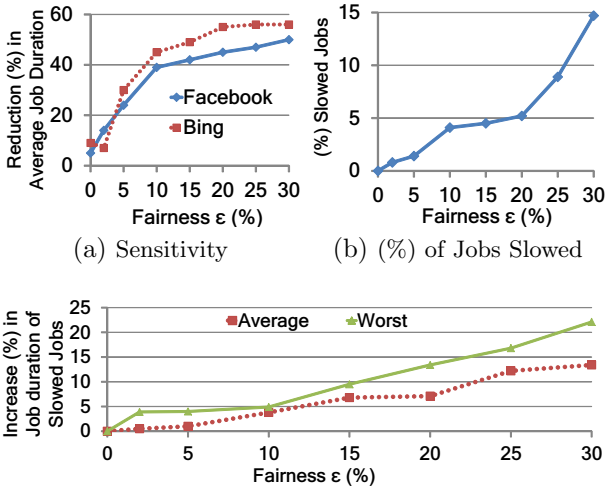
We now evaluate the sensitivity of decentralized Hopper to our key design decisions: fairness and probe ratio.

Fairness: As we had described in §4.3, the fairness knob ϵ decides the leeway for Hopper to trade-off fairness for performance. Thus far, we had set ϵ to be 10% of the perfectly fair share of a job (ratio of total slots to jobs), now we analyze its sensitivity to Hopper's gains.

Figure 10a plots the increase in gains as we increase ϵ from 0 to 30%. The gains quickly rise for small values of ϵ, and beyond $\epsilon = 15\%$ the increase in gains are flatter with both the Facebook as well as Bing workloads. Conservatively, we set ϵ to 10%.

An important concern, nonetheless, is the amount of *slowdown* of jobs compared to a perfectly fair allocation ($\epsilon = 0$), i.e., when all the jobs are guaranteed their fair share at all times. Any slowdown of jobs is because of receiving fewer slots. Figure 10b measures the number of jobs that slowed down, and for the slowed jobs, Figure 10c plots their average and worst slowdowns. Note that fewer than 4% of jobs slow down with Hopper compared to a fair allocation at $\epsilon = 10\%$. The corresponding number for the Bing workload is 3.8%. In fact, both the average and worst slowdowns are limited at $\epsilon = 10\%$, thus demonstrating that Hopper's focus on performance does *not* unduly slow down jobs.

Probe Ratio: An important component of decentralized scheduling is the probe ratio – the number of re-



(a) Sensitivity (b) (%) of Jobs Slowed

(c) Magnitude (%) of Slowdown

Figure 10: ϵ **Fairness. Figure (a) shows sensitivity of gains to ϵ. Figure (b) shows the fraction of jobs that slowed down compared to a fair allocation, and (c) shows the magnitude of their slowdowns (average and worst).**

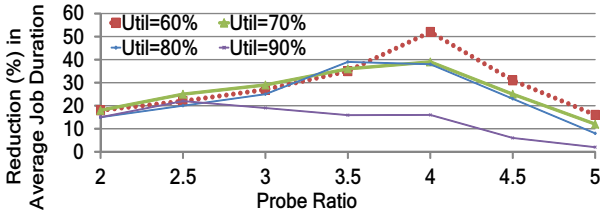


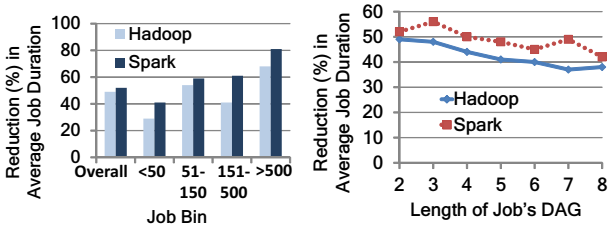
Figure 11: **Power of d choices: Impact of the number of probes on job completion.**

quests queued at workers to number of tasks in the job. A higher probe ratio reduces the chance of a task being stuck in the queue of a busy machine, but also increases messaging overheads. While the power-of-two choices [38] and Sparrow [36] recommend a probe ratio of 2, we adopt a probe ratio of 4 based on our analysis in §5.

Figure 11 confirms that higher probe ratios are indeed beneficial. As the probe ratio increase from 2 onwards, the payoff due to Hopper's scheduling and straggler mitigation results in gains increasing until 4; at utilizations of 70% and 80%, using 3.5 works well too. At 90% utilization, however, gains start slipping even at a probe ratio of 2.5. However, the benefits at such high utilizations are smaller to begin with.

7.4 Centralized Hopper's Improvements

To highlight the fact that Hopper is a unified design, appropriate for both decentralized and centralized systems, we also evaluate Hopper in a centralized setting using Hadoop and Spark prototypes. Figure 12 plots the gains for the two prototypes with Facebook and Bing workloads. We achieve gains of $\sim$ 50% with the



(a) Gains (b) DAG

Figure 12: **Centralized Hopper's gains over SRPT, overall and broken by DAG length (Facebook workloads).**

two workloads, with individual job bins improving by up to 80%.

As with the decentralized setting, gains for small jobs are lower due to the baseline of SRPT already favoring small jobs. Between the two prototypes, gains for Spark are consistently higher (albeit, modestly). Spark's small task durations makes it more sensitive to stragglers and thus it spawns many more speculative copies. This makes Hopper's scheduling more crucial.

DAG of Tasks: Like in the decentralized implementation, Hopper's gains hold consistently over varying DAG lengths, see Figure 12. Note that there is a contrast between Spark jobs and Hadoop jobs. Spark jobs have fast in-memory map phases, thus making intermediate data communication the bottleneck. Hadoop jobs are less bottlenecked on intermediate data transfer, and spend more of their time in the map phase [13]. This difference is captured via α, which is learned as described in §6.3.

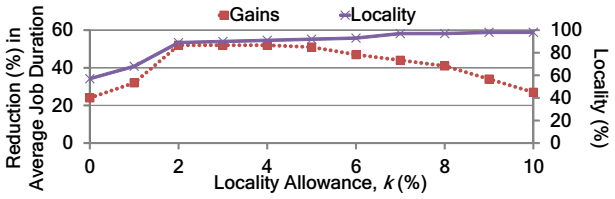
Data Locality: Recall from §4.4 that we achieve data locality using a relaxation heuristic to allow any k subsequent jobs (as a % of total jobs).

As Figure 13a shows, a small relaxation of $k = 3\%$, which is what we have used so far, achieves appreciable increase in locality in Spark. Gains are steady for a bit but then start dropping beyond a k value of 7%. This is because the deviation from the theoretical guidelines overshadows any increase in gains from locality. The fraction of data local tasks, naturally, increases with k (Figures 13a). Hadoop results are similar (13b).

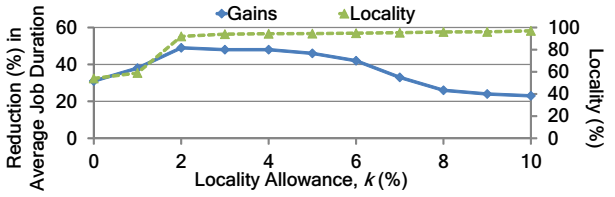
Note that even when we enhance a centralized SRPT scheduler to include the above locality heuristic, it gains no more than 20% compared to centralized SRPT (without the locality heuristic). This indicates that Hopper's gains are predominantly due to coordinated speculation and scheduling.

8. CONCLUSIONS

With launching speculative copies of tasks being a common approach for mitigating the impact of stragglers, schedulers face a decision between scheduling speculative copies of some jobs versus original copies of other jobs. While this question is seemingly simple, we find



(a) Spark



(b) Hadoop

Figure 13: **Centralized Hopper: Impact of Locality Allowance (k) (see §6.2) with Facebook workload.**

that the problem is not only unsolved thus far, but also has significant performance implications.

This paper proposes Hopper, the first speculation-aware job scheduler, and implements both decentralized and centralized prototypes. We deploy our prototypes (built in Sparrow [36], Spark [49] and Hadoop [3]) on a 200 machine cluster, and see job speed ups of 66% in decentralized settings and 50% in centralized settings compared to current state-of-the-art schedulers. In addition to providing performance improvements in both centralized and decentralized settings, Hopper is compatible with all current speculation algorithms and incorporates data locality, fairness, DAGs of tasks, etc.; thus, it represents a unified speculation-aware scheduling framework.

9. ACKNOWLEDGMENT

We would like to thank Michael Chien-Chun Hung, Shivaram Venkataraman, Masoud Moshref, Niangjun Chen, Qiuyu Peng, and Changhong Zhao for their insightful discussions. We would like to thank the anonymous reviewers and our shepherd, Lixin Gao, for their thoughtful suggestions. This work was supported in part by National Science Foundation (NSF) with Grants (CNS-1319820, CNS-1423505).

10. REFERENCES

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

[2] Cloudera Impala. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html.

[3] Hadoop. http://hadoop.apache.org.

[4] Hadoop Capacity Scheduler. http://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html.

[5] Hadoop Distributed File System. http://hadoop.apache.org/hdfs.

[6] Hadoop Slowstart. https://issues.apache.org/jira/browse/MAPREDUCE-1184/.

[7] Hive. http://wiki.apache.org/hadoop/Hive.

[8] Hopper Technical Report. https://sites.google.com/site/sigcommhoppertechreport/.

[9] Sparrow. https://github.com/radlab/sparrow.

[10] The Next Generation of Apache Hadoop MapReduce. http://developer.yahoo.com/blogs/hadoop/posts/2011/02/mapreduce-nextgen/.

[11] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Popularity Content in MapReduce Clusters. In *EuroSys*, 2011.

[12] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *USENIX NSDI*, 2013.

[13] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *USENIX NSDI*, 2012.

[14] G. Ananthanarayanan, M. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *USENIX NSDI*, 2014.

[15] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, E. Harris, and B. Saha. Reining in the Outliers in Map-Reduce Clusters Using Mantri. In *USENIX OSDI*, 2010.

[16] E. Bortnikov, A. Frank, E. Hillel, and S. Rao. Predicting Execution Bottlenecks in Map-Reduce Clusters. In *USENIX HotCloud*, 2012.

[17] E. Boutin, J. Ekanayake, W. Kin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *USENIX OSDI*, 2014.

[18] M. Bramson, Y. Lu, and B. Prabhakar. Randomized load balancing with general service time distributions. In *Proceedings of Sigmetrics*, pages 275–286, 2010.

[19] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proceedings of the VLDB Endowment*, (2), 2008.

[20] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.

[21] H. Chen, J. Marden, and A. Wierman. On the Impact of Heterogeneity and Back-end Scheduling in Load Balancing Designs. In *INFOCOM*. IEEE, 2009.

[22] J. Dean. Achieving Rapid Response Times in Large Online Services. In *Berkeley AMPLab Cloud Seminar*, 2012.

[23] J. Dean and L. Barroso. The Tail at Scale. *Communications of the ACM*, (2), 2013.

[24] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 2008.

[25] F. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized Task-aware Scheduling for Data Center Networks. In *ACM SIGCOMM*, 2014.

[26] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX NSDI*, 2011.

[27] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-Resource Packing for Cluster Schedulers. In *ACM SIGCOMM*, 2014.

[28] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems (TOCS)*, 21(2):207–233, 2003.

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

[30] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *ACM SOSP*, 2009.

[31] M. Lin, L. Zhang, A. Wierman, and J. Tan. Joint Optimization of Overlapping Phases in MapReduce. *Performance Evaluation*, 2013.

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

[33] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós. On Scheduling in Map-reduce and Flow-shops. In *ACM SPAA*, 2011.

[34] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The Case for Tiny Tasks in Compute Clusters. In *USENIX HotOS*, 2013.

[35] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun. Making Sense of Performance in Data Analytics Frameworks. In *USENIX NSDI*, 2015.

[36] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *ACM SOSP*, 2013.

[37] K. Pruhs, J. Sgall, and E. Torng. Online scheduling. *Handbook of scheduling: algorithms, models, and performance analysis*, pages 15–1, 2004.

[38] A. Richa, M. Mitzenmacher, and R. Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 2001.

[39] L. Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16(3):687–690, 1968.

[40] 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.

[41] J. Tan, X. Meng, and L. Zhang. Delay Tails in MapReduce Scheduling. *ACM SIGMETRICS Performance Evaluation Review*, 2012.

[42] Y. Wang, J. Tan, W. Yu, L. Zhang, and X. Meng. Preemptive ReduceTask Scheduling for Fast and Fair Job Completion. *USENIX ICAC*, 2013.

[43] A. Wierman. Fairness and scheduling in single server queues. *Surveys in Operations Research and Management Science*, 16(1):39–48, 2011.

[44] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to unfairness in an m/gi/1. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 238–249. ACM, 2003.

[45] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K. Wu, and A. Balmin. FLEX: a Slot Allocation Scheduling Optimizer for MapReduce Workloads. In *Middleware 2010*. Springer, 2010.

[46] N. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and Faster Jobs using Fewer Resources. In *ACM SoCC*, 2014.

[47] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. In *UC Berkeley Technical Report UCB/EECS-2009-55*, 2009.

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

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

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