Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale

Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, Minlan Yu

California Institute of Technology, *Microsoft Research*, *University of Southern California*

Paper # 253

Abstract

As data analytics clusters grow in size and complexity, providing *scalable and predictable* performance is an important challenge. This requires scheduling hundreds of thousands of tasks per second while still making jobs immune to unforeseen interactions that cause the emergence of straggler tasks. For these reasons, clusters are (i) moving to *decentralized* designs where multiple schedulers schedule tasks autonomously, and (ii) deploying straggler mitigation schemes that schedule *speculative* copies for stragglers, picking the earliest completion.

In this paper, we design Hopper, the first *decentralized speculation-aware job scheduler* that is provably optimal. We implement Hopper and show that it speeds up low latency (sub-second) Spark jobs on a 200 machine cluster by up to 66% over state-of-the-art decentralized schedulers. Further, since speculation-aware job scheduling is an open problem even in centralized schedulers, we also evaluate a centralized implementation of Hopper inside the Hadoop and Spark schedulers. This yields job speedups by 50% over all existing centralized solutions.Thus, Hopper provides a *unified speculation-aware job scheduler for centralized and decentralized* schedulers that is *provably optimal*.

1 Introduction

Data analytics frameworks have been successful in realizing 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 *predictable and scalable performance* for interactive analytics [1, 2] presents an important ongoing challenge. Indeed, providing predictable and scalable performance is acknowledged as a prominent goal in production clusters at Google and elsewhere [3].

Scalability: With clusters scaling to tens of thousands of machines, where each machine is equipped with tens of *compute slots* for tasks, schedulers have to make hundreds of thousands of scheduling decisions per second. This requirement is about two orders of magnitude beyond the (already highly-optimized) centralized schedulers adopted by current frameworks [4, 5, 6]. Therefore, frameworks are beginning to adopt *decentralized* designs [3, 7, 8] where multiple schedulers operate autonomously, with each of them scheduling only a subset of the jobs. Such designs are highly scalable since there is no requirement to maintain central state.

Predictability: As the scale and complexity of clusters increase, hard-to-model systemic interactions that degrade performance of tasks become common [3, 9]. Consequently, many tasks become "stragglers", running slower than expected – nearly one-fifth of all tasks can be categorized as stragglers in Facebook's Hadoop cluster and can run up to $8\times$ slower than expected [9]. Stragglers lead to significant performance degradation, both in terms of delay and variability. As a result, mitigating the impact of stragglers has received widespread attention, e.g., [9, 10, 11, 12, 13], and currently every major cluster implements some form of straggler mitigation. The most widely deployed solution is *speculation*, i.e., speculatively running extra copies of tasks that either have already, or are likely to become, stragglers, and then picking the earliest of the copies to finish. Speculation significantly improves the predictability of task, and hence, job completion times.

Providing scalability and predictability: In this paper, we seek to design a decentralized job scheduler that allocates resources to jobs while *also* accounting for the speculative copies that will be spawned for straggler mitigation. In fact, *even current centralized schedulers do not effectively coordinate job scheduling and speculation.* All existing schedulers take the requirements (number of slots/tasks) of jobs when they arrive and allocate slots to them to minimize job completion time [14, 15, 16, 17] and/or ensure fairness [18, 19]. Requirements of jobs, however, change due to dynamic speculation of tasks. If the scheduler performs best-effort speculation by treating speculative tasks as normal ones, it often lacks the urgency required for speculative copies to be effective. On the other hand, if the scheduler budgets a fixed number of slots for speculation, it risks reserving too little so as to not speculate enough tasks, or too much causing wastage of resources.

Our contribution is the design of a *decentralized speculation-aware job scheduler* that provides both predictable and scalable performance. The scheduler we propose, named Hopper, dynamically allocates slots online to jobs, keeping in mind the speculation requirements necessary for predictable performance while maintaining scalability by avoiding the need to maintain the state of all jobs centrally.

Hopper is based on structural (and simple) design guidelines that provably guarantee optimal performance, for both centralized and decentralized clusters. At the core of its design is the idea of a *virtual size* for jobs, which characterizes the point of diminishing returns for the performance gains that come from allocating more slots for speculation to a job versus speculating on other jobs. Building on virtual job sizes, we also identify different resource allocation strategies depending on the cluster capacity constraints. When there are limited slots, we ensure the smaller jobs are allotted their virtual job sizes. When slots are not limited, we allocate them in proportion to the virtual sizes.

Importantly, the core components of Hopper are *easy to decentralize*, and so the decentralized version of Hopper nearly matches what is possible given perfect global information. A key factor in this is that Hopper adopts a "power of many choices" viewpoint to approximate the global state, which is more appropriate than the traditional "power of two choices" viewpoint due to the fact that stragglers create heavy-tailed task sizes. Additionally, Hopper provides lightweight mechanisms to obtain the cluster capacity constraints and virtual job sizes.

To demonstrate the potential of Hopper, we have built both centralized and decentralized prototypes. The decentralized prototype of Hopper is built by augmenting the recently proposed Sparrow scheduler [7] that is decentralized across many independent schedulers. Hopper incorporates many practical features of jobs into its scheduling. It estimates the amount of *intermediate* data produced by the job and accounts for their pipelining between phases. It also carefully balances *data locality* requirements of tasks while staying faithful to the guidelines. The centralized prototype of Hopper is built inside current centralized scheduling frameworks Hadoop [20] (for batch jobs) and Spark [5] (for interactive jobs).

We have evaluated our decentralized and centralized prototypes on a 200 node private cluster using workloads derived from Facebook's and Microsoft Bing's production analytics clusters. The decentralized implementation of Hopper reduces the average job completion time by up to 66% compared to the Sparrow [7] decentralized scheduler. The gains are consistent across speculation algorithms, DAGs of jobs, and locality constraints, while providing fine-grained control on the amount of unfairness introduced. The centralized prototype of Hopper also reduces average job completion time by 50% compared to state-of-the-art centralized schedulers.

2 Challenges and Opportunities

In this section, we first briefly present details of existing schedulers, both how they allocate resources across jobs and how they handle straggling tasks (§2.1). Then, using simple illustrations, we illustrate the inefficiencies that come from a *lack of coordination* between scheduling and speculation. This highlights the challenges and opportunities associated with speculation-aware scheduling (§2.2). We then explain the additional challenges for such a coordinated design in decentralized schedulers (§2.3). Note that, to this point, even centralized systems do not coordinate job scheduling and straggler mitigation, let alone decentralized designs.

2.1 Background on Cluster Schedulers

Job scheduling – allotting compute slots to jobs for their tasks – is a well-studied topic, so here we describe only two widely-deployed approaches. For simplicity, we omit important practical factors like data locality [21], placement constrains [22], and multi-phase (e.g., map and reduce phases) jobs, though all existing solutions and our design do incorporate them.

A policy of particular interest 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). For example, in Figure 1a, there are two jobs, job A with 4 tasks, and job B with 6 tasks (we deal with the assignment of speculative slots shortly). Under SRPT, four slots are assigned to job A and the remaining two slots to job B. SRPT, in the case of a single server, provably minimizes job completion time in a very strong sense [23]. In multiple server setting, although SRPT is no longer optimal, it achieves the best competitive ratio among online algorithms [14].

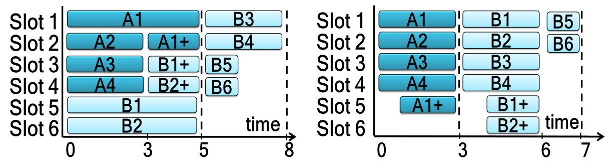
Of course, unfairness is a concern for SRPT given its prioritization of small jobs at the expense of large jobs. Fairness is a crucial issue in many clusters, and as a result, popular job schedulers [18, 19, 21, 22, 24] incorporate notions of fairness in dividing resources among the jobs. Without loss of generality, we focus on the so-called *Fair Scheduler* [18] that allocates the available compute slots evenly among all the active jobs. Such strong fairness naturally comes with performance inefficiencies compared to SRPT.

Importantly, all existing job schedulers (including the above two) do not consider upfront the possibility of tasks in the job straggling when allocating resources to them. In cluster workloads today, nearly one-fifth of tasks can be categorized as stragglers, and can run up to $8\times$ slower than the median task of the job [9]. The dominant technique for straggler mitigation is to spawn *speculative* copies for straggler tasks, and then pick the results from the earliest among them (and kill the unfinished copies) [10, 11, 13].[1]

2.2 Combining Scheduling and Speculation

We next show that if job schedulers and speculation mechanisms operated *jointly*, there can be significant

[1]Speculation strategies mainly differ in when to spawn speculative copies and how many speculative copies to spawn.



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

Figure 1: **Combining SRPT scheduling and speculation for two jobs A (4 tasks) and B (6 tasks) on a 6-slot cluster. The + suffix indicates speculation. Table 1 has task durations for both the initial tasks and the speculative copies.**

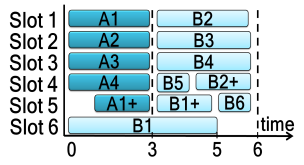


Figure 2: Hopper**: completion time for jobs A and B are** 3 **and** 6**, respectively.**

A	A1	A2	A3	A4
t_{init}	8	3	3	3
t_{new}	2	2	2	2

B	B1	B2	B3	B4	B5	B6
t_{init}	6	6	3	3	1	1
t_{new}	2	2	2	2	1	1

Table 1: t_{init} **and** t_{new} **are task durations of the original and speculative copies of each task.** t_{rem} = t_{init} – **elapsed time.**

gains. We first explore two intuitive baseline approaches for combining decisions on job scheduling and speculation using illustrative examples. In these examples we assume that stragglers can be detected after a task is run for 1 time unit and that, at this point, speculation is performed if the remaining running time (t_{rem}) is longer than the time to run a new task (t_{new}). For simplicity, in these examples we assume that $t_{new} < t_{init}$ for all tasks, though this will not be true in practice. Further, we assume that schedulers do not preempt running tasks due to the overheads and complexity.

Best-effort Speculation: A simple approach, which is also the most common in practice, is to treat speculative tasks the same as normal tasks. So, 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 (6 tasks) that are scheduled using the SRPT policy. The scheduler has to wait until time 3 to find an open slot for the speculative copy of A1, despite detecting it was straggling at time 1.[2] Clearly, this approach is problematic. In Figure 1a, if the job scheduler had allocated a slot to A's speculative task at time 1 (instead of letting B use it), then job A's completion time would have been reduced, without slowing down job B (see Table 1 for the task durations). In the interest of space, we omit a similar illustration for fair scheduling when both jobs are allotted 3 slots each.

Budgeted Speculation: The main problem in the above example is a lack of slots for speculation when needed. Thus, an alternative approach is to have the job scheduler reserve a fixed "budget" of slots for speculative tasks. In this way, speculative tasks for stragglers do not have to wait and can be run much earlier. Budgeting the right size of the resource pool for speculation, however, is challenging 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 two slots (slot 5 and slot 6) being reserved for speculative tasks. This, unfortunately, leads to slot 6 lying fallow from time 0 to 3. If the job scheduler could have used the wasted slot 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 occur even with a fair scheduler. Note that reserving one instead of two slots will not solve the problem, since two speculative copies are required to run simultaneously at a later time.

Speculation-aware Job Scheduling: Figure 2 shows how joint decision making helps in the above example. At time $0-3$, we allocate 1 extra slot to job A (for a total of 5 slots), thus allowing it to speculate task A1 promptly. After time 3, we can dynamically reallocate the slots to job B to optimize its speculation solution and reduce its completion time. As a consequence, the average completion time drops compared to both the budgeted as well as best-effort strategies. The joint design allowed idling of slot 5 until time 1 (budgeting) but after task A1 finished, it allocated all the slots. Further, in this example it can easily be seen that the schedule in Figure 2 is *optimal* (for completion time).

Thus, the goal of speculation-aware job scheduling distills to *dynamically allocating slots for speculation based on the distribution of stragglers and cluster utilization*. In doing so, it should take care to *not deviate too far from fair allocations*. Although straggler mitigation and job scheduling are heavily-studied problems in both research and practice, devising a joint optimal solution is still an open problem even for centralized scheduling. Our solution achieves 50% improvement even in the centralized context compared to these prior proposals.

2.3 Decentralized Scheduling

Thus far we have discussed the challenges in joint speculation-aware job scheduling in centralized schedulers. To cope with growing sizes and jobs having larger parallelism [25], clusters are adopting decentralized schedulers (e.g., at Google [3], Cosmos [8] at Mi-

[2]At time 3, when A2 finishes, the job scheduler allocates the slot to job A. This is because based on SRPT, job A's remaining processing time is smaller than job B's. Job A speculates task A1 because A1's $t_{rem} = t_{init} - 3 = 5$ is larger than $t_{new} = 2$ (see Table 1).

crosoft, Sparrow [7]). Briefly, decentralized schedulers have many autonomous schedulers that assign tasks to machines. When assignments on a machine collide, they get queued and *workers* on machines process the queue to schedule tasks (explained in detail in §4).

In addition to the challenges mentioned in §2.2, *decentralized* speculation-aware scheduling has further 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, decentralized schedulers work by *probing* machines for their tasks. Consequently, every scheduler has information about only a subset of the cluster (to avoid overheads). Therefore, the algorithm for dynamically allocating slots for speculation has to function with incomplete information about both the machines as well as jobs in the cluster. Finally, decentralized schedulers are mainly critical for interactive analytics (jobs running sub-second or in a few seconds), thus precluding any time-consuming gossiping between schedulers.

In §3 and §4, we describe the details of our speculation-aware job scheduler whose design is not only optimal in a centralized setting but also can be decentralized easily and efficiently.

3 Central Speculation-Aware Scheduling

In this section, we introduce the design of the *optimal* speculation-aware job scheduler in a centralized setting before extending the design to a decentralized setting in §4. The design is motivated by guidelines that provably lead to throughput-optimal performance. For ease of exposition, we do not include the details of the analyses (they can be found in [26]), but instead provide a discussion of guidelines that emerge from the analysis.

The key questions raised in the design of speculation-aware job scheduling are the following. How many speculative copies to spawn for a task? How to prioritize between speculative and original copies across jobs, especially when slots in the cluster are limited? Finally, how are the above decisions made within DAGs of dependent tasks? We address all the above questions in this section.

At a high level, the design includes two key components. The first is the notion of a "virtual job size", which we use to quantify the impact that job-specific factors like straggler likelihood, the DAG of tasks, etc., have on the speculation for a given job (§3.1). The second is the dynamic capacity allocation to jobs using very different scheduling rules depending on cluster utilization (§3.2). We incorporate DAGs of tasks and fairness in §3.3.

3.1 Virtual Job Sizes

A crucial aspect of speculation-aware job scheduling is understanding how much speculation is necessary to optimize a given job's completion time. Our analysis leads to the notion of "virtual job size" that is an estimate of the original size of the job plus the speculative copies that will be spawned. It is this "virtual job size" that is crucial for determining how to divide capacity across jobs, as we illustrate later in §3.2.[3]

The number of speculative copies spawned for tasks in a job is naturally a function of the magnitude of the stragglers (i.e., the distribution of task durations). Task durations in production traces from Facebook and Microsoft Bing [13] follow a heavy-tailed Pareto distribution, and so the Pareto tail parameter β represents the likelihood of stragglers. Roughly, smaller β means that, if a task has already run for some time, there is higher likelihood of the task running longer. In our traces $1 < \beta < 2$.

Given these assumptions, we derive the "desired (minimum) level of speculation" for a job as $2/\beta$. This level of speculation is important because it is the point of diminishing return, i.e., speculating on a job above this level results in a lower throughput improvement than the improvement that would come from speculating on a job whose allocation is below this level. So, for a scheduler to be throughput optimal, no job should speculate above this level until all jobs speculate at this level [26]. Note that, overall, this ensures that more speculation is used when stragglers are more frequent (smaller β).

Motivated by the above, we define the *virtual size of a job* ($V_i(t)$, at any time t) as its number of remaining tasks ($T_i(t)$) multiplied by the $2/\beta$, i.e., $V_i(t) = \frac{2}{\beta}T_i(t)$.

An important consequence of defining virtual size of a job in this way is that the allocation of slots to jobs can consider speculation decisions through a simple translation (size $\rightarrow$ virtual size); see line 2 in Pseudocode 1.

Note that this discussion has assumed homogeneous β, but in our implementation β is learned and can be both time-dependent and job-dependent.

3.2 Dynamic Resource Allocation

Given the virtual job sizes defined above, the next question is how to allocate resources across jobs. There are two cases that matter: whether the system is capacity constrained or not.

(i) Slots in the cluster are constrained: If the sum of the virtual sizes of jobs in the cluster is more than the number of slots, then the key design challenge is to decide how much capacity to trim from the allocations of each job. Options vary from giving the limited slots to a few jobs and allowing them to maintain the desired level of speculation, to giving all jobs some sub-optimal number of slots to avoid starving any of the jobs. Our analysis indicates the following guideline.

[3]Of course, straggler mitigation strategies typically spawn speculative copies for a task only after observing its performance for a short duration. We ignore this observation duration in analyses as it is relatively negligible compared to the task's duration in practice.

```
1: procedure HOPPER(⟨Job⟩ $J$, int $S$, float $\beta$)
       totalVirtualSizes ← 0
2:     for each Job $j$ in $J$ do
           $j.\mathrm{V_{rem}} = (2/\beta)\, j.\mathrm{T_{rem}}$
                         ▷ $j.\mathrm{T_{rem}}$: remaining number of tasks
                                     ▷ $j.\mathrm{V_{rem}}$: virtual size
           totalVirtualSizes += $j.\mathrm{V_{rem}}$
3:     SortAscending($J$, $\mathrm{V_{rem}}$)
4:     if $S <$ totalVirtualSizes then
5:         for each Job $j$ in $J$ do
               $j.\mathrm{slots} \leftarrow \lfloor \min(S, j.\mathrm{V_{rem}}) \rfloor$
               $S \leftarrow \max(S - j.\mathrm{slots}, 0)$
6:     else
7:         for each Job $j$ in $J$ do
               $j.\mathrm{slots} \leftarrow \lfloor (j.\mathrm{V_{rem}}/\mathrm{totalVirtualSizes}) \times S \rfloor$
```

Pseudocode 1: Hopper **(centralized) for jobs in set J with S slots in the cluster and task distribution parameter β.**

Guideline 1 *If there are not enough slots for every job to maintain its desired level of speculation, then, to achieve throughput-optimality, slots should be dedicated to the smallest jobs and each job should be given a number of slots equal to its virtual size.*

Intuitively, this results from the fact that task durations are heavy-tailed, and thus the benefit from additional speculation (up to the desired level) is larger than the benefit from scheduling additional tasks. The scheduler should process jobs in ascending order of virtual sizes $V_i(t)$ giving each job its desired speculation level until capacity is exhausted (see lines $3-5$ in Pseudocode 1).

This guideline is similar to the spirit of SRPT; however (unlike SRPT) it crucially pays attention to the desired speculation level of jobs when allocating capacity. Note that prioritizing small jobs may lead to unfairness for larger jobs. We discuss this issue in §3.3.

(ii) Slots in the cluster are plenty: If the sum of the virtual sizes of jobs in the cluster is less than the number of slots, then we have enough slots to allocate every job its virtual size, while still having slots left over. Thus, the key design challenge becomes how to divide the extra capacity among the jobs to make best use of it. The scheduler could give all the extra slots to a few jobs in order to complete them very quickly, or split the slots evenly across jobs, or pick other options in between. Our analysis leads to the following guideline.

Guideline 2 *If there are enough slots to permit every job to maintain its desired level of speculation, then, to achieve throughput-optimality, the slots should be shared "proportionally" to the virtual sizes of the jobs.*

Specifically, jobs should be allocated slots proportional to their virtual job sizes, i.e., every job i receives

$$\left(\frac{V_i(t)}{\sum_j V_j(t)}\right) S = \left(\frac{T_i(t)}{\sum_j T_j(t)}\right) S \text{ slots}, \qquad (1)$$

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 guideline is different in spirit from SRPT – large jobs get allocated more slots than small jobs. The reason for this is that every job is guaranteed the desired level of speculation already. Extra slots are more valuable for large jobs due to the fact that they are likely to incur more stragglers. In the context of the examples in §2, the above guidelines prescribe *how many* slots to budget at every point in time based on straggler probabilities in the currently running jobs and cluster utilization.

3.3 Additional practical complexities

The design guidelines we have discussed so far are based on single-phased jobs. In this section, we extend the guidelines to handle DAGs of tasks and fair allocations.

DAG of Tasks: The characteristic of importance in jobs with multi-phased DAGs is that phases that are not often separated by strict barriers but are rather *pipelined*. Downstream tasks do not wait for *all* the upstream tasks to finish but read the upstream outputs as the tasks finish [27]. Pipelining the reads is beneficial because the upstream tasks are typically bottlenecked on other *non-overlapping* resources (CPU, memory), while reading takes network resources.

The scheduler's goal in this setting is to balance the gains due to overlapping network utilization while still favoring upstream phases with smaller remaining number of tasks. We capture this using a simple 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. We approximate the remaining works using the amount of data remaining to be read and written, respectively; the exact details of estimating α are deferred to §5.3. It suffices to understand that α favors jobs with higher remaining communication and lower remaining tasks in the current phase.

Given the weighting factor α, there are two key adjustments that need to be made to Hopper. First, in Guideline 1, the prioritization of jobs based on the virtual size $V_i(t)$ should be 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.[4] Second, the virtual size itself needs to be redefined as $V_i(t) = \frac{2}{\beta} T_i(t)\sqrt{\alpha_i}$ to ensure throughput-optimality [26]. This result is similar in spirit to the optimality of square-root proportionality in load balancing across heterogeneous servers [28].

Incorporating Fairness: While fairness is an important constraint in clusters to ensure basic predictability, conversations with datacenter operators reveal that it is not

[4]Results in [15] show that picking the $\max\{T_i(t), T_i'(t)\}$ is 2-speed optimal for completion times.

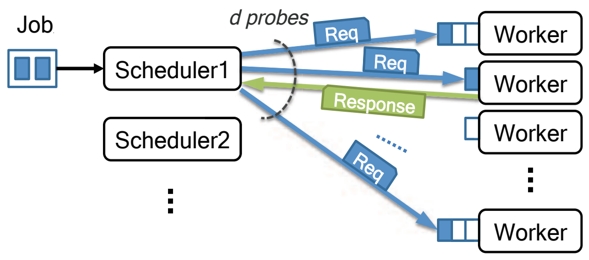


Figure 3: **Decentralized scheduling architecture.**

an absolute requirement. Thus, we relax the notion of fairness currently employed by cluster schedulers, e.g., [18]: if there are $N(t)$ active jobs at time t, then each job is assigned $S/N(t)$ slots. 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 $S/N(t) - \epsilon$ slots at *all* times t. The fairness knob ϵ can be set as a fraction of $S/N(t)$; $\epsilon \to 0$ indicates absolute fairness while $\epsilon \to 1$ indicates focus on performance.

Hopper can be adjusted to guarantee ϵ fairness in a very straightforward manner. In particular, if a job receives less than the its fair share, i.e., fewer than $S/N(t) - \epsilon$ slots, the job's capacity assignment is bumped up to $S/N(t) - \epsilon$. Next, the remaining slots are allocated to the remaining jobs according to Guidelines 1 and 2. Note that this is a form of projection from the original (unfair) allocation into the feasible set of allocations defined by the fairness constraints. Further, note that, while we have described the mechanism in terms of "job fairness" the same approach can be used to ensure approximate "user fairness" if desired.

Our experimental results (§6.3) highlight that even at moderate values of ϵ, *nearly all jobs finish faster then they would have under fair scheduling.* This fact, though initially surprising, is similar to the conclusions about SRPT-like policies in other contexts. For example, in single server scheduling SRPT is intuitively unfair to large job sizes but in reality improves the average response time of every job size (when job sizes are heavy-tailed) compared to fair schedulers [29, 30, 31].

4 Hopper: Decentralized Scheduling

In this section, we adapt the guidelines described in §3 to decentralized schedulers. Decentralized schedulers are growing in importance as cluster sizes grow. As we will explain in this section, a key benefit of our guidelines in §3 is that they are easy to decentralize.

Decentralized schedulers, like the recently proposed Sparrow [7] and others [3, 8], broadly adopt the following design (see Figure 3). 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.

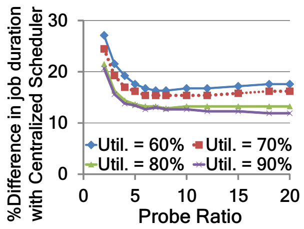
A scheduler first pushes *reservation requests* for its tasks to workers; each request contains the identifier of the scheduler placing the request along with the remaining number of unscheduled tasks in the job. When a worker is vacant, it pulls a task from the corresponding scheduler based on the reservation requests in its waiting queue. In this framework, workers decide which job's task to run and scheduler for the corresponding job decides which task to run within the chosen job. This decoupling naturally facilitates the design of Hopper, given the interface between job scheduling and straggler mitigation in the design. Though we adopt the overall design structure of 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 §3.

Decentralizing Hopper involves the following steps—approximating worker-wide information at each scheduler (§4.1), deciding if the number of slots are constrained (§4.2), and calculating virtual sizes (§4.3).

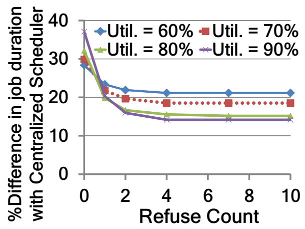
4.1 Power of Many Choices

Decentralized schedulers have to *approximate* the global state of the cluster—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" [32]. This celebrated and widely used result highlights that, in many cases, one nearly matches the performance of an optimal 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 (see §3.1). Recent theoretical results have indeed proven that when task sizes are heavy-tailed probing $d > 2$ choices can provide orders-of-magnitude improvements [33]. 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



(a) Number of probes, d



(b) Number of refusals

Figure 4: **The impact of number of probes and number of refusals on Hopper's performance.**

queues. Two choices may not be enough to avoid such backed-up queues, given the higher frequency of large job sizes. 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 4a to highlight the benefit of using $d > 2$ probing choices. Our simulation considers a cluster of 50 schedulers and 10,000 workers with job sizes and task durations as per Pareto distribution (§3.1). Job performance with decentralized Hopper is within just 15% of the optimal centralized scheduler; the difference plateauing beyond $d = 4$. Sparrow, that uses $d = 2$, is 70% off from the optimal schedule. In practice, however, there are overheads due to increased message processing, which we will capture in §6.

4.2 Is the system capacity constrained or not?

In the decentralized setting, workers implement our scheduling guidelines. Recall that Guideline 1 or Guideline 2 is applied depending on whether the system is constrained for slots or not. This 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. 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 1, i.e., enforces an SRPT priority on its queue. Specifically, when a worker is idle, it sends a *type*-0 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 type-0 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 (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

```
procedure ResponseProcessing(Response response )
    Job $j$ ← response.job
    if response.type = 1 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 ≥ 2 then          ▷ 2 is refuse threshold
        $j$ ← $J$.PickAtRandom()
        SendResponse($j$, 1)            ▷ type-1 response
    else
        $j$ ← $J$. min(virtual_size)
        SendResponse($j$, 0)            ▷ type-0 response
```

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

an "unsatisfied" job exists).

Subsequently, the worker sends a type-0 response to the scheduler corresponding 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 2. Once it is following Guideline 2, 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 *type-1* 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 4b show that performance with two or three refusals is within $10\% - 15\%$ of the optimal central scheduler.

4.3 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 sending 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.

5 Implementation Overview

We now discuss the decentralized and centralized implementations of Hopper.

5.1 Decentralized Implementation

Our decentralized implementation uses the Sparrow [7] framework, which consists of many schedulers and workers (one each on every machine) [34]. Arbitrarily many schedulers can operate concurrently; though we use 10 in our experiments. Schedulers allow submissions of jobs using Thrift RPCs [35].

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 [36, 5, 37]. 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 etc. Figure 6 in [7] explains the architecture.

Our implementation modifies the scheduler as well as the worker. The workers implement the core of the guidelines in §3—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 as per Pseudocode 3. The scheduler deals with a response from the worker as per Pseudocode 2. Note that the scheduler does not "cancel" its pending requests even after the job's tasks have been scheduled (including virtual size), because if the system is not slot-constrained, it would be able to use more slots (as per Guideline 2).

5.2 Centralized Implementation

We also implement Hopper inside two centralized frameworks: Hadoop YARN (version 2.3) and Spark (version 0.7.3). Hadoop jobs read data from HDFS [38] 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.

Data Locality: Unlike the Sparrow model where locality constraints are implicitly met by queuing the reservation requests, centralized schedulers have more leeway since they have a global view. This raises the following trade-off between locality and our guidelines.

As per our guidelines, however, tasks of the next best job to schedule may not have memory local slots available [39]. Our analysis shows that memory locality drops from 98% of tasks with currently deployed techniques to as low as 54% if scheduled purely based on our guidelines without regard to memory locality. Not only are the tasks not achieving memory locality slowed down, the ensuing increase in network traffic also slows down the data transfers of intermediate phases.

To counter this, we devise a simple relaxation approach to balance adherence to our guidelines and locality. In the ordering of jobs, instead of allotting slots to the job with the smallest virtual size, we allow for picking any of the smallest $k\%$ of jobs whose tasks can run with memory locality on the available slots. Among these smallest $k\%$ jobs, we pick the one which can achieve memory locality for the maximum number of tasks. Further, once we pick a job, we schedule all its tasks (so that a few unscheduled tasks do not delay it) before resuming to the scheduling order as per our guidelines. In practice, a small value of k suffices ($\leq 5\%$) due to high churn in task completions and slot availabilities (evaluated in §6).

5.3 Estimating Intermediate Data Sizes

Recall from §3.3 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.

For multi-waved jobs [37, 25], Hopper can do better. It uses the ratio of intermediate to input data of the completed tasks as a predictor for the future (incomplete) tasks. Data from Facebook's and Microsoft Bing's clusters (described in §6.1) shows that while the ratio of input to output data size of tasks vary from 0.05 all the way to 18, the ratios *within* tasks of a phase have a coefficient-of-variation of only 0.07 and 0.24 at median and 90th percentile, thus lending themselves to effective learning. Hopper calculates α as the ratio of the data remaining to be read (by downstream tasks) over the data remaining to be produced (by upstream tasks).

Hopper's approach for pipelining phases easily composes to DAGs of arbitrary depths since it deals with only two phases at time, i.e., the currently running phase and the downstream phase that is reading its output.

6 Evaluation

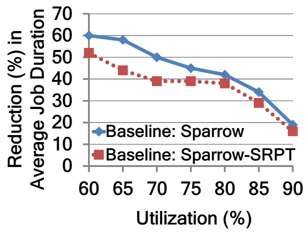
We evaluate our prototypes of Hopper – with both decentralized and centralized scheduling – on a 200 machine cluster. We evaluate Hopper's overall gains in §6.2 and design choices in §6.3. In §6.4 we evaluate the gains with Hopper in a centralized scheduler. Key highlights:

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. (§6.2)
2. Hopper's balancing of fairness and performance ensures that only 4% of jobs slow down, and jobs which slow down do so by $\leq 5\%$. (§6.3)
3. Hopper's centralized prototype improves jobs by 50% compared to centralized SRPT. (§6.4)

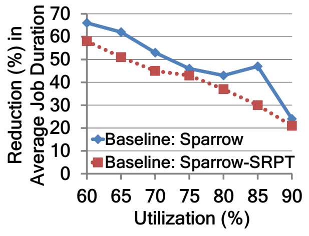
6.1 Setup

Workload: Our evaluation is based on traces from Facebook's production Hadoop [20] cluster $(3,500$ machines) and Microsoft Bing's Dryad cluster ($\mathcal{O}(1000)$ machines) from Oct-Dec 2012. The traces capture over a million jobs (experimental & production). The tasks have diverse resource demands of CPU, memory and IO, varying by a factor of $24\times$. To create our workload, we retain the inter-arrival times of jobs, their input sizes and number of tasks, resource demands and job DAGs of tasks whose lengths vary considerably.

To evaluate our prototype of decentralized Hopper, we use in-memory Spark [5] 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 by appropriate factors, and evaluate on utilizations between 60% and 90%, consistent with Sparrow [7]. For the centralized Hopper's evaluation, we replay the original trace unchanged.

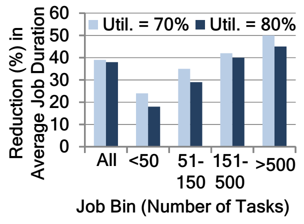


(a) Facebook

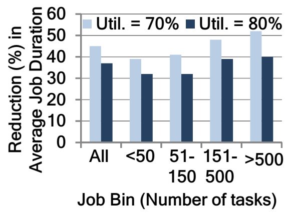


(b) Bing

Figure 5: Hopper's **gains with cluster utilization.**



(a) Facebook



(b) Bing

Figure 6: Hopper's **gains by job bins over Sparrow-SRPT.**

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 over-subscription. Each experiment is a 6 hour run and it is repeated five times; we report the median.

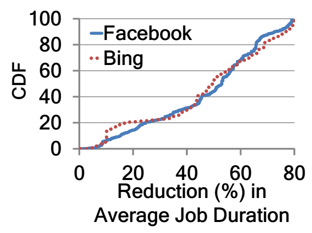
Baseline: We compare Hopper to the state-of-the-art decentralized scheduler: Sparrow [7], which performs decentralized scheduling using a "batched" power-of-two choices. We also augment Sparrow to include an SRPT heuristic to improve its performance. In short, when a worker has a slot free, it picks the task of the job that has the least unfinished tasks (instead of the standard FIFO ordering in Sparrow). Finally, we augment Sparrow by combining it with LATE [10] using "best effort" speculation (§2).[5] 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 the coordination between scheduling and speculation. Note that comparisons between decentralized schedulers (including Hopper) and a centralized baseline is not the focus of our work.

6.2 Hopper's Improvements

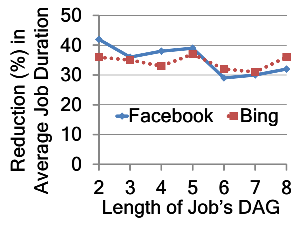
In our experiments, unless otherwise stated, we set the fairness allowance ϵ as 10%, probe ratio as 4 and speculation algorithm in every job to be LATE [10].[6] The

[5]We do not consider "budgeted" speculation due to the difficulty of picking a fixed budget.

[6]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



(a) Distribution



(b) DAG

Figure 7: **(a) CDF of Hopper's gains, and (b) gains as the length of the job's DAG varies; both at 60% utilization.**

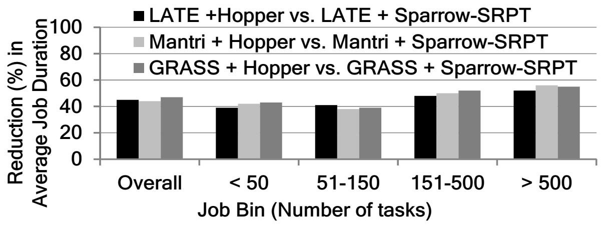


Figure 8: **Hopper's results are independent of the straggler mitigation strategy.**

values of the α (§5.3) and β (§3.1) are learned appropriately. While the accuracy of these learnings is high, we do not present the details due to space constraints.

Overall Gains: Figure 5 plots Hopper's gains for varying utilizations, compared to stock Sparrow and Sparrow-SRPT. Jobs, overall, speedup by over 60% at lower utilizations. The gains compared to Sparrow are marginally better than Sparrow-SRPT at low cluster utilizations. At high utilizations, Hopper's gains compared to both are similar. An interesting point is that Hopper's gains with the Bing workload in Figure 5b 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, naturally because there is not much room for any optimization at that occupancy.

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

Job Bins: Figure 6 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 [37, 9]) 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 [25], Hopper's allocation becomes important.

Distribution of Gains: Figure 7a 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.

requests at randomly chosen machines. Consequently, these tasks *may* run without data locality, and our results include such loss in locality.

DAG of Tasks: Hopper's gains hold steady for jobs with varying DAG lengths. The scripts in our Facebook (Hive scripts [40]) and Bing (Scope [41]) workloads produce DAGs of tasks which often pipeline data transfers of downstream phases with upstream tasks [27]. 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 7b shows, Hopper's gains hold with across DAG lengths.

Speculation Algorithm: We now experimentally evaluate Hopper's performance with different speculation mechanisms. LATE [10] is deployed in Facebook's clusters, Mantri [11] is in operation in Microsoft Bing, and GRASS [13] 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 8 plots the results.

While the earlier results were achieved in conjunction with LATE, a remarkable point about Figure 8 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 allocate as per the optimal speculation level. Overall, it emphasizes the aspect that resource allocation *across* jobs (with speculation) has a higher performance value than straggler mitigation *within* jobs.

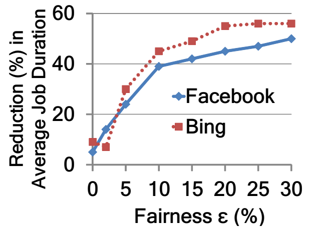
6.3 Evaluating Hopper's Design Decisions

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

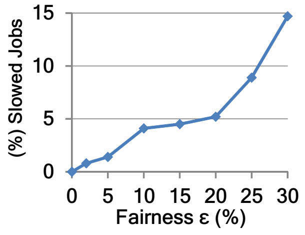
Fairness: As we had described in §3.3, the fairness knob of ϵ decides the leeway for Hopper to trade-off fairness for performance. Thus far, we had set ϵ to be 10% of the perfectly fair share of a job (ratio of total slots to jobs), now we analyze its sensitivity to Hopper's gains.

Figure 9a 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%.

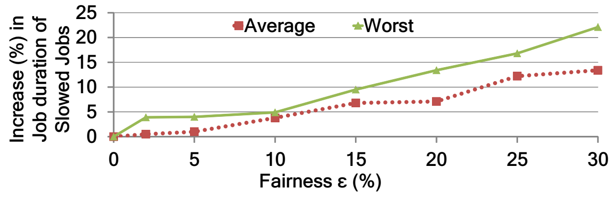
An important concern, nonetheless, is the amount of *slowdown* of jobs compared to a perfectly fair allocation



(a) Sensitivity



(b) (%) of Jobs Slowed



(c) Magnitude (%) of Slowdown

Figure 9: 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).**

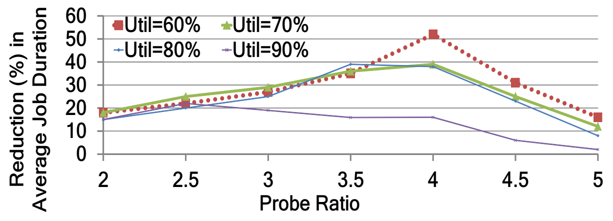
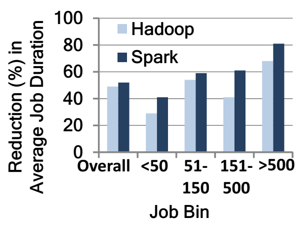


Figure 10: **Power of d choices: Impact of the number of probes on job completion.**

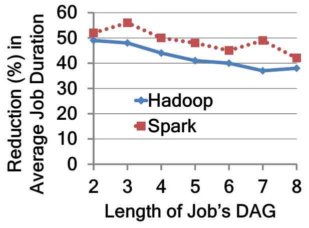
($\epsilon = 0$), i.e., all the jobs are guaranteed their fair share at all times. Figure 9b measures the number of jobs that slowed down, and for the slowed jobs, Figure 9c plots their average and worst case slowdowns. Note that fewer than 4% of jobs slow down with Hopper compared to a fair allocation at $\epsilon = 10\%$. The corresponding numbers for the Bing workload are 3.8% of jobs slowing down. In fact, both the average and worst case 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 requests 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 [32] and Sparrow [7] recommend a probe ratio of 2, we adopt a probe ratio of 4 based on our analysis in §4.

Figure 10 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.



(a) Gains



(b) DAG

Figure 11: **Centralized Hopper's gains over SRPT, overall and broken by DAG length (Facebook workloads).**

6.4 Evaluating Centralized Hopper

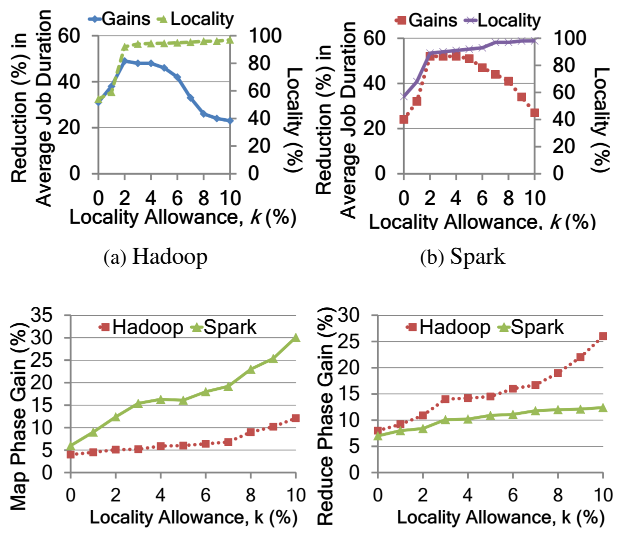
To highlight the fact that Hopper is a unified design, appropriate for both decentralized and centralized systems, we also evaluate Hopper in a centralized setting using Hadoop and Spark prototypes. Analogous to earlier evaluations, we use a centralized SRPT scheduler with LATE speculation as our baseline. Again, this an aggressive baseline since it sacrifices fairness for performance. Thus, improvements can be interpreted as coming solely from better coordination of scheduling and speculation.

Figure 11 plots the gains for the two prototypes with Facebook and Bing workloads. We achieve gains of $\sim$ 50% with the two workloads, with individual job bins improving by up to 80%. The equally encouraging performance of Hopper in the centralized and decentralized settings highlight that it is a *unified solution for speculation-aware scheduling*.

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.

Two new issues crop up in the centralized design: how DAGs of tasks are handled and how locality constraints are handled. Thus, we discuss each below.

DAG of Tasks: Like in the decentralized implementation, Hopper's gains hold consistently over varying DAG lengths, see Figure 11. 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



(a) Hadoop (b) Spark

(c) Map Phase (d) Intermediate Phases

Figure 12: **Centralized Hopper: Impact of Locality Allowance (k) (see §5.2) with Facebook workload.**

of their time in the map phase [37]. This difference is captured via α, which is learned as described in §5.3.

Data Locality: Recall from §5.2 that, while locality was implicitly handled in Hopper's decentralized scheduling, the centralized version requires a relaxation heuristic, allowing any k subsequent jobs (as a $\%$ of total jobs).

As Figures 12a and 12b show, a small relaxation of $k = 3\%$ achieves appreciable increase in locality. Gains 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, increase with k (Figures 12a and 12b, right axis).

An interesting aspect is that not all the gains with k are attributed to increases in locality. To see this, we slice the gains of individual phases—map phase, which is directly affected by locality, and other intermediate phases (e.g., reduce, join). Figure 12c, shows that the map phases in Spark speed up significantly as locality increases; Hadoop jobs' map phases much less so. This is because data locality is more significant in Spark's in-memory system as opposed to Hadoop's disk-based storage; fast networks make *disk* locality less useful [42]. Hadoop jobs gain by improvement in their other intermediate phases due to lesser network contention during intermediate data transfers (Figure 12d).

7 Related Work

The problem of stragglers was first identified in the original MapReduce paper [12], and since then there have been many works that propose to mitigate the stragglers by utilizing speculative copies, e.g., [9, 10, 11, 13]. These solutions, however, aim at mitigating the stragglers within each job, and lack coordination of resource allocation among all concurrently running jobs.

Job scheduling, on the other hand, is often done via algorithms that do not integrate straggler mitigation. Specifically, FIFO [14], the default scheduler for Hadoop and Spark, suffers from well known head-of-line blocking in multi-user cloud sharing settings. The inefficiency of FIFO inspired two different approaches: (i) introducing fairness among jobs; (ii) prioritizing small jobs.

Based on the first approach, widely used solutions include the Fair Scheduler [18], Capacity Scheduler [24], DRF [19], FLEX [43], and Quincy [21]. While these schedulers guarantee fair sharing among jobs, fairness comes with its performance inefficiencies, e.g., [44, 45].

On the second approach, the optimality of SRPT scheduling in both single [23] and multi-server [14] settings in queueing theory motivates a focus on prioritizing small jobs. Variations of SRPT that accommodate a variety of workload properties have been proposed. For example, incorporation of the dependency of two adjacent phases is highlighted in [15, 17]. Various queuing models (such as two-stage flexible flow-shop model [46], overlapping tandem queue model [15]) have also inspired new algorithmic designs. Importantly, no system yet provides coordination of scheduling and speculation.

Another crucial challenge is that of scale. As clusters scale to tens of thousands of machines, it is increasingly desirable to have decentralized job scheduling. A variety of decentralized designs have emerged to this point, e.g., [3, 7, 8]. The closest of these to the current work is Sparrow [7], which combines "late binding" with the "power of two choices" but does not consider stragglers. In fact, no decentralized schedulers have considered stragglers to this point. Thus, Hopper, represents the first decentralized speculation-aware job scheduler.

8 Conclusions

This paper proposes a decentralized speculation-aware cluster scheduler, Hopper. 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 at first blush, we find that the problem is not only unsolved thus far, but also has significant performance implications. From first principles, we derive simple structural scheduling guidelines that are provably optimal for both decentralized as well as centralized schedulers. We deploy our prototype Hopper (built in Sparrow [7], Spark [5] and Hadoop [20]) on a 200 machine cluster, and see jobs speed up by over 50% in both decentralized and centralized settings compared to current state-of-the-art schedulers.

References

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

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

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

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

[5] 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.

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

[7] Ousterhout K, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[8] 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.

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

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

[11] 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.

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

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

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

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

[16] W. Wang, K. Zhu, L. Ying, J. Tan, L. Zhang . A Throughput Optimal Algorithm for Map Task Scheduling in Mapreduce with Data Locality. In *ACM SIGMETRICS*, 2013.

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

[18] M. Zaharia, D. Borthakur, J. Sen 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.

[19] 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.

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

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

[22] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of EuroSys*, pages 365–378, 2013.

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

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

[25] 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.

[26] X. Ren. Speculation-aware resource allocation for cluster schedulers. Master's thesis, Caltech.

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

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

[29] 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.

[30] 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.

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

[32] A. Richa, M. Mitzenmacher, and R. Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.

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

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

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

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

[37] 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.

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

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

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

[41] 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.

[42] G. Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica. Disk Locality Considered Irrelevant. In *USENIX HotOS*, 2011.

[43] 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.

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

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

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