Speculative Slot Reservation: Enforcing Service Isolation for Dependent Data-Parallel Computations

Chen Chen, Wei Wang, Bo Li
Hong Kong University of Science and Technology
{cchenam, weiwa, bli}@cse.ust.hk

***Abstract*—Priority scheduling is a fundamental tool to provide *service isolation* for different jobs in shared clusters. Ideally, the performance of a high-priority job should not be dragged down by another with a lower priority. However, we show in this paper that simply assigning a high priority provides *no* isolation for jobs with *dependent computations*. A job, even receiving the highest priority, may give up compute slots to another before proceeding to the downstream computation, which is because of *barrier*, i.e., that the downstream computation cannot start until *all* the upstream tasks have completed. Such an interruption of execution inevitably results in a significant delay.**

In this paper, we propose *speculative slot reservation* that judiciously reserves slots for downstream computations, so as to retain service isolation for high-priority jobs. To mitigate the utilization loss due to slot reservation, we analyze the trade-off between utilization and isolation, and expose a tunable knob to navigate the trade-off. We also propose a complementary straggler mitigation strategy that uses the reserved slots to run extra copies of slow tasks. We have implemented speculative slot reservation in Spark. Evaluations based on both cluster deployment and trace-driven simulations show that our approach enforces strict service isolation for high-priority jobs, without slowing down the other jobs with a lower priority.

I. Introduction

Today's data-parallel clusters are *shared* by multiple tenants running diverse jobs with complex workflows in the form of directed acyclic graphs (DAGs). In such shared, multi-tenant environments, providing *service isolation* between jobs of different tenants is of paramount importance.

Many production schedulers have been developed to provide performance isolation in shared clusters [1]–[7], with *priority scheduling* as a fundamental tool for policy enforcement. For example, in order to prevent the performance of business-critical production jobs from being dragged down by the experimental, non-critical jobs, the scheduler assigns the former a higher priority than the latter [3], [8]–[10]. Service isolation can also be provided via fair sharing, which is implemented by *dynamic priority scheduling*, in that jobs with the least allocation share are prioritized for resource allocation [1], [2], [5]–[7].

Enforcing service isolation by means of priority scheduling critically requires that *the performance of high-priority jobs should not be adversely impacted by others with lower priorities*. While this is a trivial requirement for *non-workflow* jobs with no dependent computations (e.g., map only and reduce only), it is no longer true for workflow jobs with complex DAGs. To illustrate this problem, we ran two Spark MLlib [11] jobs,[1]

[1]In this paper, a *job* refers to an application, not a Spark *action*.

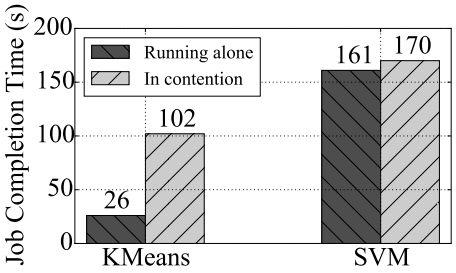


Fig. 1: [Cluster] Priority scheduling provides no service isolation. We ran two Spark MLlib jobs, KMeans and SVM, on an EC2 cluster with 4 `m4.large` instances. Each job consists of multiple dependent phases, and the degree of parallelism is 8. The KMeans job, though scheduled at a high priority, is significantly slowed down in contention with the SVM job with a lower priority.

KMeans and SVM, in a cluster of 4 Amazon EC2 `m4.large` instances [12], where KMeans is assigned a higher priority than SVM. Fig. 1 compares the completion times of the two jobs running *alone* against running *in contention*. Contrary to our expectation, KMeans, though scheduled at a higher priority, suffers from $3.9\times$ slowdown in contention with the SVM job.

The root cause of this problem results from the *work conservation* requirement of task scheduling, in that a compute slot should not be left idle if it can be offered to a backlogged task. In the previous example, both KMeans and SVM jobs consist of dependent tasks and are executed in pipelined *phases*. A phase can only start after *all* tasks in the upstream phase have completed, due to the dependency limitation known as the *barrier*. More often than not, tasks belonging to the same phase have varying execution times due to data skew, cross-rack traffic and unstable machine state [13]–[15]. Therefore, when a task of a high-priority job (e.g., KMeans) finishes, in case of a barrier, the downstream phase cannot start right away. A work conserving scheduler would then offer the newly released slot to a task of a low-priority job (e.g., SVM). Therefore, KMeans gives up most of its slots to SVM upon a phase transition, before a barrier. Worse, those slots cannot be reclaimed immediately after the barrier has been cleared, and KMeans has to wait for the completion of the occupying SVM tasks. The result is a long job completion delay, the significance of which increases with the number of phases.

That priority scheduling fails to provide desired service isolation has two consequences. First, it makes the behaviors of priority-based cluster schedulers, including fair schedulers,

hard to reason about. Second, it promotes strategic behaviors as tenants can receive more "free" compute resources by launching jobs with long-running tasks at a low priority.

In light of these problems, our objective in this paper is to enforce the strict service isolation, in that the performance of a workflow job would not be dragged down by others at lower priorities. We propose a simple "drop-in" solution to achieve this objective, which we call *speculative slot reservation*. Our key idea is to take advantage of the rich semantics of the workflow DAGs that are readily available to the scheduler upon a job submission, e.g., [16]–[19]. By tracking the execution of workflow DAGs in runtime, we can speculate if a slot released by an upstream task of a high-priority job can soon be reused to accommodate its downstream computations. This allows us to judiciously hold the soon-to-be-reused slots from being taken by a job with a lower priority, enforcing the service isolation without sacrificing much on the resource utilization.

One concern about slot reservation is the potential utilization loss. To address this problem, we develop two complementary approaches. In the first approach, we impose a slot reservation deadline beyond which the reservation expires and the slot is released to other jobs. We analyze how the reservation deadline may impact on utilization and isolation through a simple, yet accurate analytical model. Our analysis establishes the trade-off between the two metrics, based on which we implement a tunable knob to navigate the trade-off. In the second approach, instead of keeping a reserved slot idle, we turn it into *straggler mitigators* to speed up the computation. Specifically, for a job, we aggressively use its reserved slots to run extra copies of its slow tasks (i.e., stragglers), and for each task, we pick the copy, original or replica, that finishes earlier.

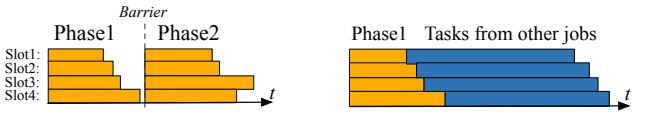
We have prototyped speculative slot reservation in Spark. Our cluster deployment on 50 `m4.large` instances in Amazon EC2 shows that our implementation provides near-perfect service isolation for priority scheduling. Compared with running alone in the cluster, high-priority jobs only experience a slight scheduling latency $< 10\%$ when contending with the background workloads with a lower priority. In addition, by tuning the knob, our implementation can flexibly navigate the trade-off space between utilization and isolation. Trace-driven simulations further demonstrate that our straggler mitigation strategy can significantly reduce the job completion time by over 70% for typical production workloads with a heavy tail of latency distribution.

II. Background and Motivation

A. Background

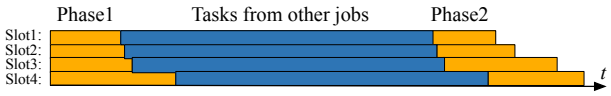
Workflow jobs. Prevalent parallel processing frameworks run data analytics jobs as workflow DAGs. For example, Dryad [20] and Tez [18] provide users with APIs to explicitly specify the workflow DAGs of data-parallel jobs. Spark [19] compiles jobs into DAGs of resilient distributed datasets (RDDs). Oozie [16] allows users to perform complex data analytics by composing multiple jobs into a logical workflow DAG.

In data-parallel clusters, workflow DAGs are executed in pipelined *phases*, each consisting of several tasks that can run in parallel on different compute slots. Between two phases there is a *barrier*, meaning that the downstream phase cannot start until *all* tasks in the upstream phases have completed. In many frameworks such as Spark [19], downstream tasks will not be submitted to the scheduler before the barrier has been cleared. As a toy example, Fig. 2a illustrates the execution of such a workflow job running alone in four slots. Note that tasks in the same phase may have *uneven* execution times due to data skew, networking, and resource competition [13].

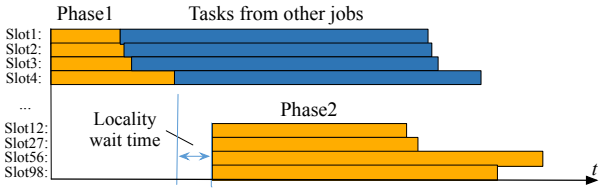


(a) Running alone without contention (b) In contention with other jobs

Fig. 2: The execution of a workflow job in pipelined phases.



(a) Case 1: Resume downstream computation on the same slots



(b) Case 2: Resume downstream computations on different slots

Fig. 3: Once the execution of a workflow job is interrupted before a barrier, its completion would be delayed.

Work conserving schedulers. In order to achieve high cluster utilization, production schedulers are designed to be *work conserving*, in that no slot is left idle in the presence of a backlogged task [1], [2], [7], [21]. This seems to be a natural requirement that should be enforced by all means. However, we show in the following subsection that naive work conservation may significantly delay job completion in a shared cluster.

B. Naive work conservation delays job completion

One issue associated with work conservation is that it results in frequent interruptions during the workflow execution, even for jobs at the highest priority. We refer back to the previous example and assume that there is another job contending at a lower priority. As illustrated in Fig. 2b, initially all slots are used to serve phase-1 of the high-priority job (shown in orange). Because of the barrier, the downstream computation cannot start right after the completion of a phase-1 task. The work conserving scheduler then offers the idle slots to the low-priority job (in dark blue), interrupting the execution of the other. We next discuss two cases and show that such a service interruption delays job completion significantly.

Case-1: Downstream computations resume on the same slots. Because downstream tasks depend on the output of upstream computations, scheduling these tasks onto the same slots used in upstream phases helps provide *data locality* [7], leading to fast task completion. However, as illustrated in

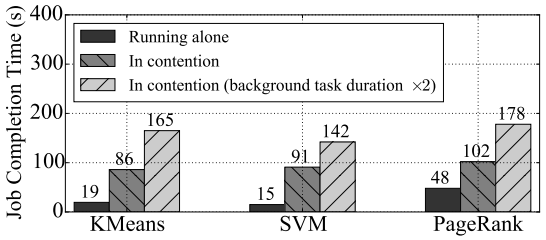
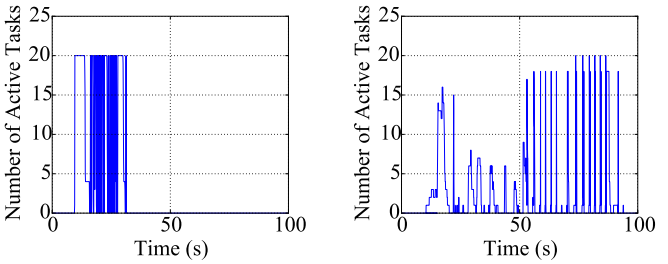


Fig. 4: [Cluster] The foreground jobs, despite having a higher priority, can be severely slowed down by the background jobs.



(a) Running alone (b) Contending with background jobs

Fig. 5: [Cluster] A detailed view of the KMeans execution over time (degree of parallelism = 20), without and with the background jobs contending at a lower priority.

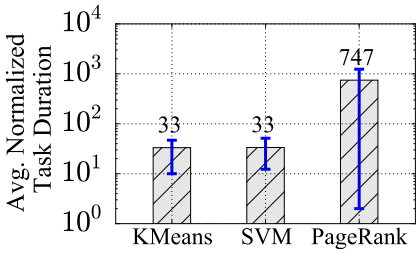


Fig. 6: [Cluster] Task slowdown without data locality.

Fig. 3a, it may take a long time for the job to reclaim the preferred slots with the dependent data.[2] But how would this impact the job completion time? We answer this question through the following experiments.

Cluster deployment: We deployed a Spark cluster in Amazon EC2 with 50 worker machines. Each machine is an `m4.large` instance with 2 cores and 6.5GB RAM, and is configured to host 2 Spark executors. We ran two types of workloads in the cluster, latency-sensitive workflow jobs and latency-tolerant batch processing jobs. We assign a higher scheduling priority to the former than to the latter, and refer to the former (latter) as the *foreground* (*background*) jobs. The foreground jobs consist of three applications, KMeans, SVM and PageRank, which are typical machine learning and graph analytics algorithms provided by SparkBench [22], a comprehensive benchmarking suites for Spark. The background workloads consist of 100 synthesized jobs randomly sampled from the Google cluster traces [10] in a one-hour window. Concerning the small cluster we have, we scaled down the task runtime in traces by $10\times$.

Methodology: To quantify how the background jobs may impact the foreground applications, we ran each foreground job in three environments at different levels of contention: running alone, in contention with background jobs, and in contention with *prolonged* background jobs (task runtime$\times 2$).

Compromise of service isolation: We compare in Fig. 4 the completion times of each foreground job running at different levels of contention with background workloads. We see that the foreground job, even granted the highest scheduling priority, is significantly delayed by the low-priority competitors, the degree of which is in proportion to the duration of background tasks. To better illustrate this problem, in Fig. 5, we micro-benchmarked the number of running tasks of KMeans over time, with and without the contention of background workloads. We see that KMeans loses slots to background jobs before a barrier, and it takes a long time to ramp up afterwards.

Case 2: Downstream computations resume on different slots. As shown in Fig. 3b, in case that the preferred slots with dependent data are not available for too long, the downstream tasks, after waiting for a certain time (specified by `spark.locality.wait` in Spark), would be scheduled onto some different slots without data locality [7]. These tasks have to fetch data from remote machines and, even worse, may run into a "cold" JVM without pre-loaded classes or compiled bytecode [23]. All these result in a non-trivial task slowdown.

To quantify the significance of task slowdown due to the loss of data locality and "cold" JVM, we randomly sampled five phases for each of the three SparkBench applications in the previous experiment and ran their tasks on different slots in different phases (at locality level `ANY`). We measured the task duration and normalize it by that of running on the same slots across phases (at locality level `PROCESS_LOCAL`). We see from Fig. 6 that the tasks, if not running at the best locality level, could be dramatically slowed down by up to two orders.

C. Summary

In summary, our experiments confirm that frequent service interruptions due to naive work conservation inevitably result in a significant job completion delay, irrespective of the scheduling priority the job receives.

We note that this problem widely exists in clusters for both inter- and intra-application scheduling. Prevalent cluster resource managers such as YARN [9] and Mesos [8] are *agnostic* to the underlying application workflows, and may frequently offer the slots released by one application to another, even though these slots would soon be needed by the downstream computations of the former. Similar problem also arises *within* an application. For example, a SparkSQL service runs different query jobs submitted by multiple client threads: any of these jobs may give up slots to others before a barrier. We expect that this problem would become even more salient in the future given the recent push of finer-grained slot multiplexing and container reuse in production clusters, e.g., Spark Job Server [24] and Tez Sessions [18].

[2]Task preemption is expensive to implement and is not supported by many cluster schedulers. Even enabled, the preemption is usually done after a grace clean period, and may incur additional slot preparation overhead.

III. Speculative Slot Reservation: The Basics

Our previous experiments reveal that the key to enforcing service isolation for a workflow job is to prevent its execution from being interrupted by another job with a lower priority. This requires us to purposely *reserve* some compute slots for downstream computations. In this section, we first show that naive reservations can be highly inefficient with low utilization. We then propose our basic design of *speculative slot reservation* to provide isolation without much utilization loss.

A. Inefficiency of naive slot reservation

Prevalent cluster resource management systems allow high-priority jobs to hold their compute resources through two simple policies: static slot reservation and timeout-based reservation. However, neither is efficient.

1. ***Static slot reservation:*** In systems such as Mesos [8] and Borg [3], cluster operators can *statically* reserve a certain number of compute slots for a group of latency-sensitive jobs. However, static slot reservation fails to capture the changing computational demands of production workloads, and may end up with either *over-provisioning* or *under-provisioning*: The former results in unnecessary resource waste with low utilization, while the latter compromises service isolation.
2. ***Timeout-based slot reservation:*** With dynamic resource allocation in Spark [25], after a task finishes, its slot (executor) can be automatically reserved for that job before a specified timeout threshold. However, this approach is inefficient in two aspects. First, the reservation is made *blindly* even if there is no downstream computation. Second, such a fixed threshold provides *no guarantee* that a slot can always be reserved till the submission of downstream computations.

B. Basic design of speculative slot reservation

Overview. Driven by the inefficiency of naive strategies, we propose *speculative slot reservation* that judiciously reserves slots to provide service isolation at a low utilization cost. Our key intuition is that *a slot should be reserved only when it could be reused shortly by downstream computations*. To this end, we make use of the DAG information of workflow jobs, which is readily available to the cluster scheduler after the job has been submitted [18]–[20]. With this information, once a task finishes on a slot, we can speculate whether the slot can be reused to accommodate a downstream task, and correspondingly choose to reserve or release that slot.

Following this intuition, we next present our basic design of speculative slot reservation in two cases, with and without *a priori* knowledge about *the degree of parallelism* of downstream phase. The degree of parallelism measures the number of parallel tasks in a phase. Algorithm 1 summarizes our basic design, which would be further refined in Sec. IV.

Case-1: No prior knowledge about the degree of parallelism in the next phase. Many frameworks typically determine the degree of parallelism at runtime [18], [19]. Without this information, we cannot know exactly how many slots are needed in the next phase. The best we can do is to assume the same degree of parallelism across different phases. Therefore, whenever a task finishes on a slot, unless the task is in the last phase, our algorithm reserves the slot for downstream computations (lines 2–8 of Algorithm 1).

While simple, this strategy turns out to be a very good approximation in production clusters. Many production workloads exhibit the stable degree of parallelism throughout the execution. For example, workload studies on traces collected from Facebook's Hadoop clusters and Microsoft Bing's Dryad clusters reveal that over 91% of jobs with ≤ 10 tasks never change the degree of parallelism in different phases [26].

Case-2: The degree of parallelism in the next phase is available beforehand. In some cases, it is possible to obtain the parallelism information *a priori*. For example, in order to fine-tune the job performance, users would like to explicitly specify the degree of parallelism in their programs. In addition, for *recurring* jobs that are run periodically in production clusters (which account for 40% of workloads in Microsoft's production clusters [27]), the degree of parallelism can be accurately learned from previous runs. Taking use of this information helps reserve slots more effectively.

In particular, let m and n respectively denote the degree of parallelism in the current and next phase. Our algorithm differentiates between the following three cases:

1. ***The degree of parallelism remains unchanged*** ***($m = n$).*** In this case, the algorithm simply reserves all slots used in the current phase (line 8 of Algorithm 1).
2. ***The degree of parallelism decreases*** ***($m > n$).*** In this case, out of the m slots used in the current phase, we let go the first $m - n$ slots that become idle, and hold the remainder (line 9-13 of Algorithm 1). This way, the utilization loss due to slot reservation is minimized.
3. ***The degree of parallelism increases*** ***($m < n$).*** In this case, simply reserving slots used in the current phase is *insufficient* to accommodate downstream computations. We therefore resort to *pre-reservation*. Specifically, we borrow the similar idea of *reduce slow start* in Hadoop [28] and define a *pre-reservation threshold* $\mathscr{R}$. When the fraction of completed tasks in the current phase exceeds $\mathscr{R}$, the scheduler starts to opportunistically reserve the additional $n - m$ slots released by other jobs (line 14-17 of Algorithm 1). This way, when the next phase starts, its tasks can acquire slots right away without further delay.

Support of priority scheduling. Our speculative slot reservation can naturally support priority scheduling. Once a slot is reserved for a job, it inherits the priority of that job. The reservation will be respected by other jobs with lower or equal priorities but can be overridden by those with higher priorities.

C. Discussions

Changing resource demands across phases. So far, we have implicitly assumed that tasks in different phases of a job have consistent resource demands and can fit in the same

Algorithm 1 Speculative Slot Reservation

```
1: procedure HANDLETASKCOMPLETION(k, s)
                                        ▷ task k completes on slot s
2:     if task k in the final phase then
3:         release s
4:     else
5:         m ← k.phase.numTasks
6:         n ← k.downstreamPhase.numTasks
7:         if n not available or m == n then
8:             reserve s and s.priority← k.job.priority
9:         else if m > n then
10:            if k in the m − n tasks finishing first in k.phase then
11:                release s
12:            else
13:                reserve s and s.priority← k.job.priority
14:        else                                              ▷ m < n
15:            reserve s and s.priority← k.job.priority
16:            if k.phase.finishedTaskFraction ≥ R then
                                  ▷ pre-reserve threshold R is reached
17:                start requesting and reserving additional n − m
   slots for k.downstreamPhase

18: procedure TRYALLOCATETASK(k, s)   ▷ task k requests slot s
19:     if s is reserved and s.priority ≥ k.job.priority then
20:         skip allocation
21:     else
22:         allocate slot s to task k
```

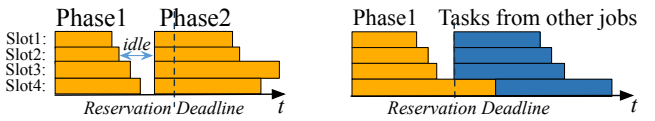
slot consisting of the same amount of resources (e.g., memory, CPU cores, etc.). While this is true in many frameworks such as Spark, it may not be the case for other frameworks such as Tez, in which tasks may demand different amounts of resources across phases. Nevertheless, speculative slot reservation can still be beneficial: if a slot used in the current phase is too small to accommodate a downstream task, we can release it immediately while pre-reserving another one of the right "size" (similar to Case-2.3, Sec. III-B).

Utilization loss due to speculative slot reservation. While our algorithm speculatively reserves slots only when needed, the utilization loss could still be a concern. However, we argue that this should be a minor issue in production clusters. Most latency-sensitive applications our algorithm targets for are *small* jobs, with short-running tasks and the low to medium degree of parallelism. According to the recent studies [26], in typical production clusters, the smallest 90% of jobs consume only 6% of the total resources. Therefore, reserving slots for these small jobs does not result in a salient utilization loss.

Despite the argument above, to further reduce the utilization loss, we refine our design with two complementary approaches in the next section.

IV. Mitigating Utilization Loss due to Reservation

In this section, we first identify stragglers as the root cause of utilization loss when slot reservation is enforced. We then analyze how long a slot should be reserved in the presence of stragglers, with an objective of striking a flexible balance between utilization loss and isolation guarantee. We further propose a straggler mitigation strategy that aggressively uses the reserved slots to execute extra copies of slow tasks.



(a) Slot reservation is *effective*, in that all tasks in the current phase have completed before the given *deadline*.

(b) Slot reservation is *ineffective*, in that not all tasks can complete by the *deadline*.

Fig. 7: Imposing a reservation *deadline* to bound the utilization loss.

A. The root cause of utilization loss

We attribute the root cause that slot reservation harms utilization to the *uneven* task runtime. Because of the barrier, a slot having its task completed earlier is kept idle until the next phase starts, a time dictated by the *slowest* task of the current phase. Therefore, the longer the latency tail of the computation, the more significant the utilization loss. Unfortunately, for a data-parallel job with a high degree of parallelism, it is common that many tasks may become *stragglers*, i.e., running much slower than expected [14], [26], [29]. Consequently, simply holding a slot till the next phase starts may, in the worst case, result in unbounded utilization loss. We address this problem in the next subsection.

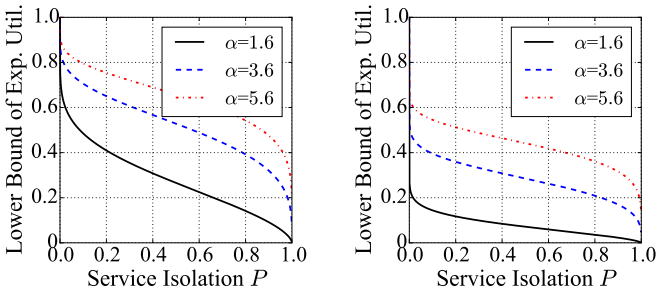
B. Hong long should a slot be reserved?

A simple, yet effective approach to mitigate the utilization loss is to impose a bounded reservation period. In particular, we associate each slot reserved in the current phase with a *deadline*, beyond which the reservation is expired, and the slot becomes free to use by other jobs. Fig. 7 shows an example. The question is: how long should we set the reservation deadline?

The trade-off between isolation and utilization. Intuitively, imposing an early deadline helps mitigate the utilization loss. The price paid is a weak isolation guarantee. The job can hold the slots for downstream computations only if *all* tasks in the current phase complete before the reservation has expired, which is less likely if the deadline is too early. Imposing a late deadline for a long reservation, on the other hand, likely retains the isolation guarantee, at the expense of low utilization.

To quantify such a trade-off between isolation and utilization, we turn to a simple analytical model, based on which we will discuss how a cluster operator can navigate the trade-off.

*1) **Notations**:* Without loss of generality, we assume that a compute phase starts at time 0. Let $\mathscr{D}$ be the slot reservation deadline, after which all the reserved slots will be released to other jobs. Let N be the number of parallel tasks in the current phase. We say slot reservation with deadline $\mathscr{D}$ is *effective* if *all* N tasks complete before $\mathscr{D}$. In other words, with effective slot reservation, the computation proceeds to the next phase without giving up slots. Given deadline $\mathscr{D}$, let P be the probability that the slot reservation is effective. We measure the level of isolation guarantee by P. Finally, let U be the utilization, measured as the fraction of *busy periods* of slots. We shall establish the tradeoff between isolation P and utilization U.



(a) Degree of parallelism = 20 (b) Degree of parallelism = 200

Fig. 8: [Numerical] Trade-off between utilization and isolation based on Eq. (4).

*2) **Workload model**:* Workload studies on Facebook and Microsoft Bing traces [29], [30] suggest that task durations in production clusters follow Pareto distribution, a heavy-tail distribution with a polynomially decaying tail. The CDF of Pareto distribution w.r.t. task duration t is given by

$$F(t) = \begin{cases} 1 - \left(\frac{t_m}{t}\right)^{\alpha} & t \geq t_m, \\ 0 & t < t_m, \end{cases} \tag{1}$$

where α is the *shape parameter* and t_m the *scale parameter*. A smaller α implies a heavier tail, i.e., more long-running tasks. For a meaningful shape parameter, we have $\alpha > 1$. Scale parameter t_m, on the other hand, measures the shortest task runtime, which can be well approximated by the duration of the task that finishes first in a phase.

*3) **Quantifying the trade-off**:* We first analyze how reservation deadline $\mathscr{D}$ may impact isolation guarantee. By definition, we measure isolation guarantee by the probability that all N tasks in the current phase complete before the deadline:

$$P = F(\mathscr{D})^N = [1 - (\frac{t_m}{\mathscr{D}})^{\alpha}]^N. \tag{2}$$

We next analyze how reservation deadline $\mathscr{D}$ may impact on the expected utilization $E[U]$. Because the exact analysis does not give a closed-form solution, we instead derive a lower bound of $E[U]$ by assuming that all slots are reserved until the deadline:

$$\begin{aligned} E[U] &\geq \frac{1}{\mathscr{D}} \int_{t_m}^{\mathscr{D}} t \cdot f(t) \mathrm{d}t \\ &= \frac{\alpha}{\alpha - 1} \left(\frac{t_m}{\mathscr{D}}\right) - \frac{1}{\alpha - 1} \left(\frac{t_m}{\mathscr{D}}\right)^{\alpha}, \end{aligned} \tag{3}$$

where $f(t)$ is the PDF of Pareto distribution.

Combining Eqs. (2) and (3), we establish the following trade-off analysis between utilization and isolation:

$$E[U] \geq \frac{\alpha}{\alpha - 1}(1 - P^{\frac{1}{N}})^{\frac{1}{\alpha}} - \frac{1}{\alpha - 1}(1 - P^{\frac{1}{N}}). \tag{4}$$

It can be shown that Eq. (4) is monotonically decreasing w.r.t. P. Meaning, a higher utilization is achieved at the expense of a lower level of isolation guarantee. As two extreme cases, enforcing strict isolation (i.e., $P = 1$) may lead to arbitrarily low utilization; providing no isolation (i.e., $P = 0$) incurs no utilization loss (i.e., $E[U] = 1$). Fig. 8 depicts the numerical results of the trade-off curve for workloads with different latency tails (i.e., α), where the degree of parallelism is set to 20 and 200, respectively. We see that the trade-off becomes increasingly sharp as the latency tail turns heavier (smaller α).

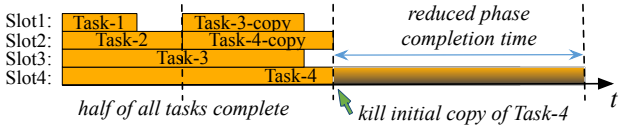


Fig. 9: An illustrative example of our straggler mitigation strategy. Once Task-2 has completed, the number of reserved slots (Slot1 and Slot2) becomes equal to the number of on-going tasks (Task-3 and Task-4). We then schedule an extra copy of each on-going task to a reserved slot. Task-4 is killed after its copy has completed, and the phase completion time is reduced.

Navigating the trade-off. Our analytical model provides a general guideline to navigate the trade-off between isolation and utilization. For example, cluster operators can allow each user to specify the desired isolation guarantee P, which is the probability that the user can retain its compute resources without being interrupted during the phase transition. To enforce the specified isolation guarantee, the operators impose the corresponding reservation deadline derived from Eq. (2):

$$\mathscr{D} = t_m(1 - P^{\frac{1}{N}})^{-\frac{1}{\alpha}}.$$

The operators can then charge each user based on the incurred utilization loss due to slot reservation (Eq. (4)). This way, the operators can fine-tune the trade-off between the demands for isolation and the incurred utilization loss. We shall evaluate this point through prototype experiments in Sec. VI.

C. Mitigating stragglers using reserved slots

As shown in Fig. 8, when workloads consist of more stragglers (i.e., heavy-tailed with small α), the trade-off between isolation and utilization can be very sharp, leaving little space for operators to land in a middle ground by tuning the slot reservation deadline. This problem can be prevalent in real systems as production workloads typically have a long tail of latency distribution with small $\alpha \in [1, 2]$ [29]. We propose a novel straggler mitigation strategy to address this problem.

Turning reserved slots into straggler mitigators. Instead of keeping the reserved slots idle, we aggressively use them to execute extra copies of slow tasks. Specifically, our algorithm keeps track of the number of on-going tasks and the number of reserved, yet idle slots in the current phase. Once the former falls below or equals to the latter, we have a sufficient number of reserved slots to help speed up *all* the on-going tasks, which we deem as "stragglers." For each of these stragglers, we launch an extra copy on a reserved slot, and pick the one—the original or the replica—that finishes earlier.

To better illustrate our idea, we refer to the example in Fig. 9, where a compute phase consists of four tasks running on four slots. After the first two tasks have completed, we have a sufficient number of reserved slots to execute extra copies

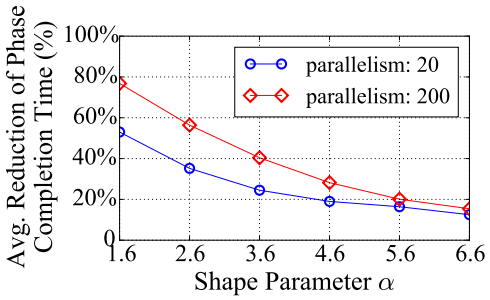


Fig. 10: [Numerical] Our straggler mitigation strategy speeds up the in-phase computation. Each data point is averaged over 1000 runs.

of the remaining two on-going tasks. In case that task-4 is a straggler, our approach effectively speeds up its execution, enabling the computation to progress to the next phase much earlier. The utilization loss is hence reduced.

We next investigate how our straggler mitigation strategy helps speed up the job execution through simple numerical studies. We postpone the simulation study to Sec. VI.

Numerical study. We start with some notations. Assume that a compute phase consists of N tasks running on N slots. Let $t_{(k)}$ denote the duration of the k^{th} shortest task. Without straggler mitigation, the phase completion time is dictated by the slowest task, i.e., $T = t_{(N)}$. We now use the reserved slots to mitigate stragglers. For the k^{th} shortest task, let $t'_{(k)}$ denote the duration of its copy. Because extra copies are launched after half of the tasks have completed, the phase completion time is derived as

$$T' = t_{(\lceil \frac{N}{2} \rceil)} + \max_{\lceil \frac{N}{2} \rceil < k \leq N} \min\{t_{(k)} - t_{(\lceil \frac{N}{2} \rceil)}, t'_{(k)}\}.$$

To quantify how straggler mitigation speeds up job completion, we compare T' and T. Given that there is no closed-form expression of T', we resort to numerical studies, where task durations are drawn i.i.d. from Pareto distribution with varying shape parameters α, from a heavy to a light latency tail. Fig. 10 shows the numerical results with different degree of parallelism. In general, the speedup derived from straggler mitigation becomes more salient when the workloads have a heavier latency tail and are of a higher degree of parallelism. For typical production workloads with $\alpha = 1.6$ [29], straggler mitigation reduces the job completion time by over 50%.

Advantages over the status quo. We compare our strategy to the state-of-the-art *speculative straggler mitigation*. Existing approaches keep track of the execution progress of each task and speculate which one is likely to become a straggler, for which an extra copy is executed on another machine [13], [26], [30]. Compared with the status quo, our approach has the following three advantages.

First, our approach does not need to maintain a complex logic for straggler speculation, and is less expensive to implement. Second, our approach requires no additional slot to mitigate stragglers for a job—all slots used are already reserved for it. In this sense, our approach is *interference-free* between jobs. Third, our approach incurs no *JVM warm-up overhead* (e.g., class loading and interpretation of bytecode) for straggler

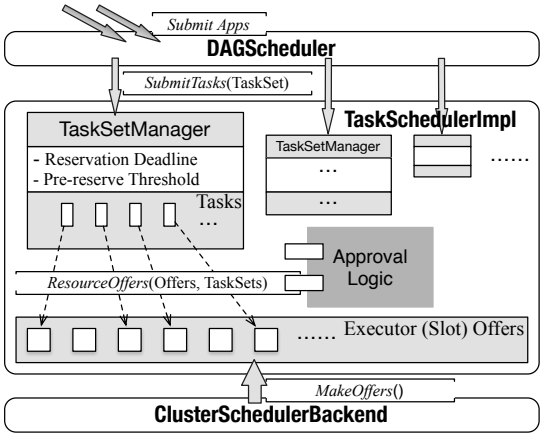


Fig. 11: Architecture overview of speculative slot reservation in Spark.

mitigation, which is frequently the bottleneck in data-parallel frameworks [23]. With our approach, a copy of a slow task is always spawned on a reserved slot with loaded classes and compiled code, given that the slot has just been used to execute tasks in the same phase. Our EC2 deployment on 20 `m4.large` instances confirms that tasks of Spark MLlib jobs can enjoy up to $8\times$ speedup if running on slots with warm data.

D. Summary

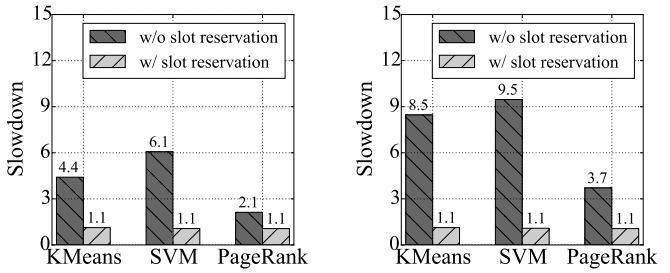
We stress that the two approaches presented in this section are *complementary* to each other in that they target on different workloads. For light-tailed workloads, judiciously setting a reservation deadline allows operators to strike a good balance between utilization loss and isolation guarantee. For heavy-tailed workloads, mitigating stragglers using reserved slots can effectively speed up job completion.

V. Prototype Implementation

In this section, we describe our prototype implementation of speculative slot reservation in Spark [19]. While we base our design in Spark, nothing will preclude implementing our proposal to other systems such as Mesos [8] and YARN [9].

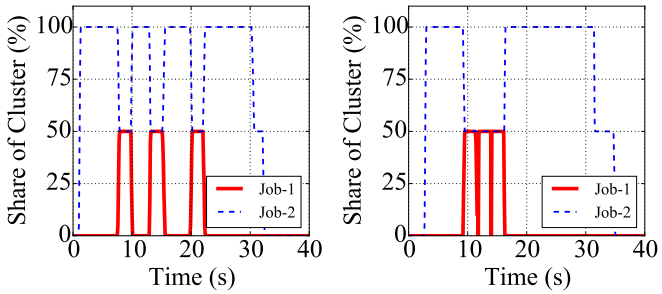
Architecture overview: We prototyped speculative slot reservation in Spark 1.6.1. Fig. 11 gives an architecture overview. In particular, we implemented the slot reservation logic in three Spark classes: (1) `DAGScheduler` which parses the workflow DAG of a job and makes an execution plan in pipelined phases, (2) `TaskSetManager` which manages the execution of all the parallel tasks within a phase, and (3) `TaskSchedulerImpl` which assigns slots obtained from `ClusterSchedulerBackend` to tasks submitted by `DAGScheduler`. We next elaborate our design in details.

Retrieving the job DAG and its execution plan: When a job is submitted to the Spark driver, its DAG information becomes readily available to `DAGScheduler`. `DAGScheduler` parses the job DAG through *backward traverse* from the final vertex using depth-first search (DFS). This process virtually constructs a DAG execution plan in a reverse order. Following this plan, `DAGScheduler` then continuously submits tasks in pipelined phases to `TaskSchedulerImpl`—the latter would then create



(a) The standard setting (b) Doubling the task durations of the background jobs

Fig. 12: [Cluster] Slowdown of each foreground job.



(a) W/o speculative slot reservation (b) W/ speculative slot reservation

Fig. 13: [Cluster] Running two synthetic jobs using Spark Fair Scheduler, where job-1 is composed of 3 pipelined phases, and job-2 consists of many parallel tasks without dependency.

a `TaskSetManager` to manage all tasks in one phase. In our implementation, `DAGScheduler` would also communicate the execution plan, along with the degree of parallelism in each phase, if specified in the user program.

Speculative slot reservation: With the workflow execution plan and the possible knowledge of the degree of parallelism, `TaskSetManager` speculatively reserves slots for the job using Algorithm 1, and notifies the reservation (or pre-reservation) to `TaskSchedulerImpl`. We also implemented the logic of deadline-based reservation (Sec. IV-B) in `TaskSetManager` to fine-tune the trade-off between utilization and isolation. The reservation deadline will be notified to `TaskSchedulerImpl`.

Enforcing slot reservation: Finally, we enforce slot reservation in `TaskSchedulerImpl`. To this end, we add the *ApprovalLogic* to `resourceOffers()`, a method of `TaskSchedulerImpl` that assigns slots to tasks. Specifically, before a slot is assigned to a task, the *ApprovalLogic* checks the validity of the assignment. The assignment is valid only if the slot has not been reserved, or it is offered to a task with a higher priority than the job that makes the reservation.

VI. Evaluation

In this section, we evaluate the performance of speculative slot reservation through prototype deployment in a 50-node EC2 cluster. For performance study at a larger scale, we resort to trace-driven simulations.

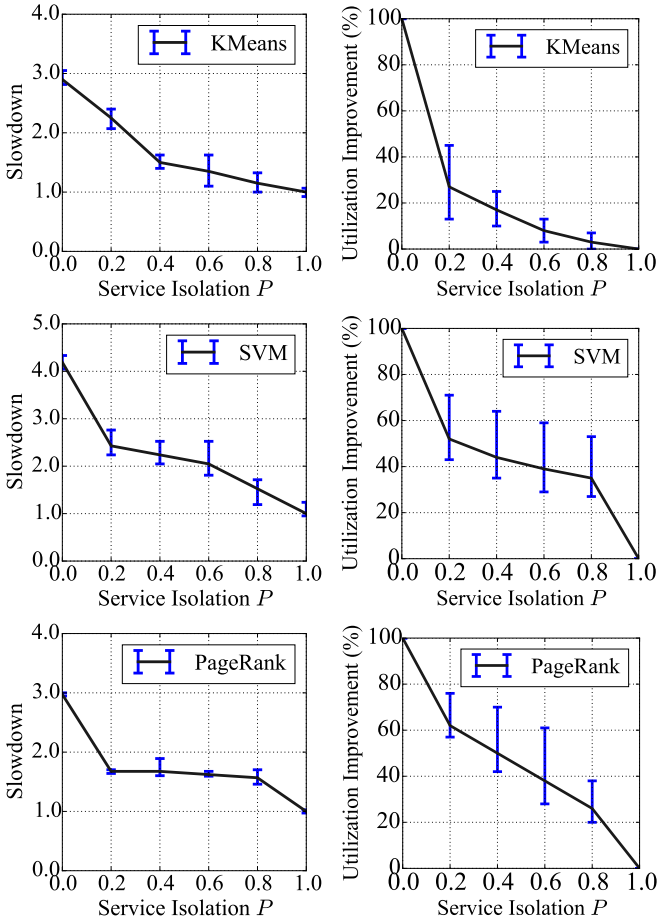


Fig. 14: [Cluster] Measured trade-off between service isolation and utilization. For all the three MLlib jobs, KMeans, SVM and PageRank, a higher level of isolation requirement leads to less slowdown, at the expense of smaller *utilization improvement*.

A. Prototype deployment

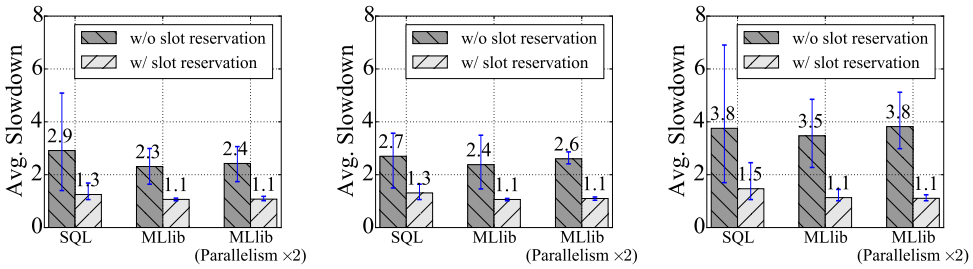
Setup. We deployed our Spark prototype (Sec. V) in a 50-node EC2 cluster with `m4.large` instances. Similar to the previous experiments in Sec. II-B, we ran two types of workloads in contention, *foreground* jobs with a high priority and *background* jobs with a low priority. The former consists of three SparkBench applications [22], KMeans, SVM and PageRank. The later consists of 100 synthesized jobs sampled from the Google traces [10] in a one-hour window.

Metric. We use *slowdown* as the main performance metric in our evaluation. Specifically, we define *slowdown* for each job as the measured job completion time (JCT) normalized by the *minimum* JCT if the job were running *alone* in the cluster:

$$\text{Slowdown} = \frac{\text{Measured JCT}}{\text{Minimum JCT when running alone}}.$$

We evaluate the performance of speculative slot reservation by answering the following questions.

Is service isolation provided? We ran two experiments. The first experiment follows the standard setup described above. In



(a) The standard setting (b) Doubling the task durations of the background jobs (c) Doubling the slowdown factor of locality

Fig. 15: [Simulation] Average slowdown of foreground jobs in different settings.

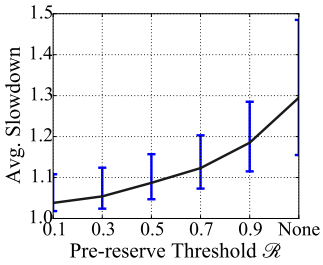


Fig. 16: [Simulation] Slowdown of the SQL jobs.

the second experiment, we emulate the contention with long-running workloads by extending the task runtime of background jobs by $2\times$. Fig. 12 compares the slowdown measured for each foreground job with and without speculative slot reservation in the two experiments. We see that our proposal enforces strict isolation for priority scheduling in both experiments. Each foreground job experiences a minor slowdown ($< 10\%$) in contention with the background jobs, mainly due to the scheduling overhead of handling more workloads.

Speculative slot reservation also helps retain service isolation for *fair schedulers*. To demonstrate this benefit, we ran two Spark jobs, using the default Spark Fair Scheduler [1]. The computation of job-1 consists of three pipelined phases, while job-2 has no dependent computations (i.e., map only). Ideally, each job should receive 50% of the cluster resources, *as if it were running alone in a half-sized cluster*. However, as shown in Fig. 13a, job-1 loses all the resources to job-2 between two phases, suffering from a severe delay due to frequent interruptions. As a comparison, Fig. 13b illustrates the allocations of the two jobs over time after speculative slot reservation has been enabled. This time, job-1 is able to withhold its fair share throughout the execution, achieving desired service isolation from job-2.

How does isolation impact on utilization? We ran each of the three foreground jobs in contention with the background jobs, at different levels of isolation requirement, specified by P (cf. Eq. (2)), the probability that the execution is not interrupted. We take $P = 1$ as the baseline, with which the utilization loss due to slot reservation reaches the *maximum*. Given an isolation requirement, we measure the *utilization improvement* over the baseline, i.e., the *percentage* of the reduction of utilization loss. We also measure the slowdown of each foreground job. Fig. 14 shows the results with varying isolation requirements in three experiments for different foreground jobs, where each data point is averaged over 10 runs. We see that a higher level of isolation requirement leads to a lower slowdown. The price paid is less utilization improvement. In general, the trade-off between isolation and utilization is smooth across the three experiments, with the sweet spot achieved at $P = 0.4$. We have only spotted a few stragglers in our workloads, and because of this, we have observed a minor utilization loss ($< 5\%$) even using the baseline algorithm with $P = 1$. Given the less significant concerns about stragglers in our EC2 cluster, we evaluate the performance of our straggler mitigation strategy (Sec. IV-C) through simulations in the next subsection.

B. Large-scale trace-driven simulations

Setup. We simulated a 1000-node cluster with 4000 slots. In such a large cluster, slot acquiring is seldom a bottleneck: a backlogged task can easily find an available slot within a short period of time. Instead, *data locality* becomes the primary concern [7], as the available slot may not have the desired input data. In our simulation, we set the *locality wait time* to 3s, the same as in Spark. That is, if a task cannot find an available slot with preferred data locality in 3s, it would accept *any* offer afterwards. Despite our observations of significant task completion delays without data locality (up to two orders in Sec. II-B), we make a *conservative* estimation in simulation by assuming $5\times$ task runtime if data locality is not provided.

Workloads. Similar to the cluster deployment, in our simulation, we ran foreground jobs at a higher priority than the background jobs. The foreground jobs are synthesized workloads from two traces, the SQL traces [31] consisting of 20 queries provided by the TPC-DS benchmark and the Spark MLlib traces we collected in our cluster deployment. To stress test the performance of speculative slot reservation, we also ran MLlib jobs with $2\times$ degree of parallelism in the foreground. The background workloads consist of a mix of 8000 jobs synthesized from the Google traces [10], SQL traces [31] and Spark MLlib traces.

Providing service isolation in large clusters. We first identify data locality as the key factor that critically impacts on the service isolation in large clusters (Case-2, Sec. II-B). We ran three experiments without speculative slot reservation. The first experiment follows the standard setup described above. In the second experiment, we injected long-running background jobs with $2\times$ task runtime. In the third experiment, we amplified slowdown factor of locality by assuming $10\times$ task runtime if running on a slot without the input data. As shown in Fig. 15b (dark bars), injecting long-running background jobs has little impact on the foreground jobs, resulting in a similar slowdown as observed in the standard setup (Fig. 15a). This verifies our previous claim that in large clusters, jobs with a high priority can quickly acquire available slots. Instead, as

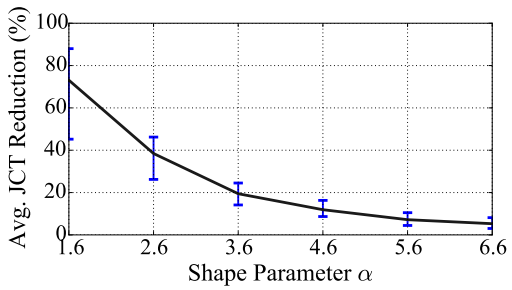


Fig. 17: [Simulation] Average JCT reduction of the foreground jobs using our straggler mitigation strategy.

shown in Fig. 15c, it is the data locality that dictates the job performance: doubling the slowdown factor of locality results in a significant increase of job slowdown.

Speculative slot reservation helps provide data locality. Slots used in previous phases hold the output data that will be fed into the downstream computations. Co-locating downstream tasks to these slots leads to a salient speedup. This benefit is clearly demonstrated in Fig. 15 (light bars). With speculative slot reservation, the two suites of MLlib jobs running in the foreground suffer from only a minor slowdown ($< 10\%$) in contention with the heavy background workloads. The SQL jobs, while experiencing a moderate level of slowdown (1.3-$1.5\times$), complete much faster than that without slot reservation.

We attribute the changing degree of parallelism as the main reason that SQL jobs are more susceptible to be dragged down by the background workloads compared with the MLlib jobs. Specifically, the job is slowed down when the downstream computation has a higher degree of parallelism that cannot be accommodated by the reserved slots in the current phase. In this case, slot pre-reservation (Case-2.3, Sec. III-B) comes as a solution, where it starts to pre-reserve extra slots as soon as the progress of the current phase has reached a specified threshold $\mathscr{R}$. As shown in Fig. 16, the earlier the pre-reservation starts (i.e., small threshold $\mathscr{R}$), the less slowdown the job suffers.

Impact on the background workload. Our study confirms that speculative slot reservation for foreground jobs has little impact on the background workload. In particular, we find that for background jobs, the average slowdown due to speculative slot reservation is less than 0.1%.

Further improvement through straggler mitigation. Finally, we investigate how our straggler mitigation strategy (Sec. IV-C) can further speed up the foreground jobs with a heavy tail of latency distribution. To this end, for each foreground job, we artificially adjusted the runtime of its task in a phase so that the latency follows Pareto distribution with a specified shape parameter α and the *same mean* as the original workload. Fig. 17 shows the reduction of job completion time (JCT) when applying our straggler mitigation strategy to workloads with different shape parameters. In general, workloads with a heavier latency tail (i.e., small α) benefit more from straggler mitigation. For typical production workloads with $\alpha = 1.6$ [29], our straggler mitigation strategy significantly reduces the JCT by 73% on average.

VII. Related Work

There have been extensive efforts on providing service isolation in shared, multi-tenant environments such as production clusters. We broadly classify these approaches into three categories, by means of queue management, resource allocation policies and OS-level isolation.

Queue management. When multiple jobs are queued up for service, service isolation can be provided by determining their scheduling orders. Production schedulers such as Mesos [8], YARN [9] and Borg [3] assign a high priority to business-critical jobs so that they can be serviced first. However, as we have explained, simply relying on the scheduling priority is unable to enforce service isolation for workflow jobs with pipelined phases.

Resource allocation policies. Service isolation can also be provided by resource allocation policies with the *sharing incentive* property [5], [6], [32]–[34]. Meaning, each of n users is guaranteed *at least* $1/n$ of the entire cluster resources. These policies are also *strategy-proofness*, in that a user cannot increase its allocation share by lying about its demands. These studies focus on the policy design, while we focus on the policy enforcement.

OS-level isolation. When tasks of different jobs are scheduled onto the same machine, cluster management systems such as Mesos [8] and Borg [3] isolate resources using OS container technologies, notably Linux containers [35] and Solaris Projects [36], that limit the CPU, memory, network bandwidth and I/O usage of a process tree. These approaches are orthogonal to our work.

VIII. Conclusion

In this paper, we showed that, contrary to the general belief, priority scheduling fails to provide service isolation for workflow jobs with dependent computations. To address this problem, we developed a simple, yet effective solution, speculative slot reservation. With speculative slot reservation, the scheduler tracks the execution of workflow DAGs in runtime, and speculatively reserves slots that have been released by upstream tasks and can soon be reused by downstream computations. To further mitigate the potential utilization loss due to slot reservation, we analyzed the trade-off between utilization and isolation, and exposed a tunable knob to navigate the trade-off. We also proposed a complementary approach that turns the reserved slots into straggler mitigators to speed up slow tasks. We have implemented speculative slot reservation in Spark and evaluated its performance in a 50-node Amazon EC2 cluster and in simulations with traces from a Google's cluster. Evaluation results suggest that speculative slot reservation enforces strict service isolation for high-priority jobs, reducing their JCTs by over 70% on average without slowing down the other jobs.

Acknowledgement

This work was supported in part by grants from RGC under the contracts 615613, 16211715 and C7036-15G (CRF), as well as a grant from NSF (China) under the contract U1301253.

REFERENCES

[1] "Spark Job Scheduling," https://spark.apache.org/docs/1.6.1/job-scheduling.html.

[2] "Hadoop Fair Scheduler," https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[3] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proc. ACM Eurosys*, 2015.

[4] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, "Altruistic scheduling in multi-resource clusters," in *Proc. USENIX OSDI*, 2016.

[5] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *Proc. USENIX NSDI*, 2011.

[6] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica, "Hierarchical scheduling for diverse datacenter workloads," in *Proc. ACM SoCC*, 2013.

[7] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proc. ACM Eurosys*, 2010.

[8] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *Proc. USENIX NSDI*, 2011.

[9] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop YARN: Yet another resource negotiator," in *Proc. ACM SoCC*, 2013.

[10] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. ACM SoCC*, 2012.

[11] "Spark MLlib," http://spark.apache.org/mllib/.

[12] "Amazon EC2," https://aws.amazon.com/ec2/.

[13] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri." in *Proc. USENIX OSDI*, 2010.

[14] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[15] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: mitigating skew in mapreduce applications," in *Proc. ACM SIGMOD*, 2012.

[16] "Apache Oozie," http://oozie.apache.org/.

[17] "Apache Pig," http://pig.apache.org/.

[18] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache Tez: A unifying framework for modeling and building data processing applications," in *Proc. ACM SIGMOD*, 2015.

[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX NSDI*, 2012.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proc. ACM Eurosys*, 2007.

[21] "Hadoop Capacity Scheduler," https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[22] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark," in *Proc. ACM CF*, 2015.

[23] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan, "Don't get caught in the cold, warm-up your jvm: Understand and eliminate jvm warm-up overhead in data-parallel systems," in *Proc. USENIX OSDI*, 2016.

[24] "Spark Job Server," https://github.com/spark-jobserver/spark-jobserver.

[25] "Spark Dynamic Resource Allocation," https://spark.apache.org/docs/1.6.1/job-scheduling.html#dynamic-resource-allocation.

[26] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. USENIX NSDI*, 2013.

[27] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: guaranteed job latency in data parallel clusters," in *Proc. ACM Eurosys*, 2012.

[28] "MapReduce Slow Start," http://www.ibm.com/support/knowledgecenter/SSGSMK_7.1.1/mapreduce_user/map_reduce_configure_slowstart.html.

[29] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized speculation-aware cluster scheduling at scale," in *Proc. ACM SIGCOMM*, 2015.

[30] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "GRASS: trimming stragglers in approximation analytics," in *Proc. USENIX NSDI*, 2014.

[31] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *Proc. USENIX NSDI*, 2015.

[32] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Choosy: Max-min fair sharing for datacenter jobs with constraints," in *Proc. ACM Eurosys*, 2013.

[33] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "Hug: Multi-resource fairness for correlated and elastic demands," in *Proc. USENIX NSDI*, 2016.

[34] W. Wang, B. Li, B. Liang, and J. Li, "Multi-resource fair sharing for datacenter jobs with placement constraints," in *Proc. IEEE/ACM SC16*, 2016.

[35] "Linux containers (LXC) overview," http://lxc.sourceforge.net/lxc.html.

[36] "Solaris Resource Management," http://docs.sun.com/app/docs/doc/817-1592.