Understanding Synchronization Costs for Distributed ML on Transient Cloud Resources

Pradeep Ambati, David Irwin, Prashant Shenoy, Lixin Gao, Ahmed Ali-Eldin, and Jeannie Albrecht‡
University of Massachusetts Amherst ‡Williams College

***Abstract*—Cloud platforms often execute parallel batch applications, such as distributed machine learning (ML), that include numerous synchronization barriers. These barriers, which prevent any task from advancing beyond a specified point until all tasks have reached that point, significantly degrade application performance by reducing it to that of the slowest "straggler" task. To address the problem, researchers have proposed numerous straggler mitigation techniques, including speculatively re-executing straggler tasks and various relaxations of strict barrier semantics. While these techniques improve parallel application performance, they incur a cost in terms of the resources wasted re-executing tasks or waiting. Importantly, these costs, which are often implicit in prior work that targets dedicated resources, become *explicit* in the cloud, which charges for resources at fine-grained intervals. In addition, the cost difference between techniques is exacerbated in cloud platforms, since they charge *substantially less* for transient resources that effectively yield a probabilistic performance across a wide range.**

While transient resources' low list price is attractive, revocations increase the frequency and severity of stragglers, which decreases parallel job performance and increases overall execution cost. To better understand the cost of synchronization, we develop simple analytical models of different straggler mitigation techniques and compare their cost and performance on on-demand and transient resources. Our analysis shows that i) transient servers offer complex tradeoffs compared to on-demand servers, and can result in higher overall costs despite their highly discounted price due to their probabilistic performance; ii) common approaches to straggler mitigation, which is a well-studied problem, are less effective using transient servers that cause frequent and severe stragglers; and iii) a recent approach to flexible synchronization offers the best cost and performance.

***Index Terms*—Synchronization, Transient Server, Revocation**

I. Introduction

Public cloud platforms provide users access to an essentially unlimited number of servers on demand without requiring a large capital investment. Thus, enterprises are increasingly leveraging public clouds to run large-scale workloads, often for distributed data processing, across hundreds-to-thousands of servers. Since distributed data processing platforms, including Hadoop [1], Spark [2], Tensorflow [3], and parameter servers (PSs) [4], simplify running jobs across many resources, they have become the dominant platforms for leveraging cloud resources. These platforms partition data processing jobs into parallel tasks that run concurrently, and, thus, must determine when to synchronize their output across tasks. Many general platforms, including Hadoop and Spark, adopt a bulk synchronous processing (BSP) model [5], which defines synchronization barriers that prevent any task from advancing beyond specified points until all tasks have reached those points.

Unfortunately, synchronization barriers significantly degrade parallel application performance by reducing it to that of the slowest "straggler" task. The original work on MapReduce identified such stragglers, which arise for many reasons, including hardware failures [6], software configuration errors [7], and resource contention due to background processes, e.g., garbage collectors or OS daemons [8], [9]. Notably, such stragglers occur even when parallel tasks run on homogeneous hardware. While prior work assumes stragglers are rare, having at least one straggler that degrades job performance is expected under high degrees of parallelism. As a result, prior work has developed many techniques to identify and mitigate the effect of stragglers on performance. For example, the original MapReduce work proposed identifying straggler tasks, speculatively executing backup tasks, and then accepting the result of whichever task finishes first (and cancelling the other task). Subsequent work further optimized the policy for when to spawn backup tasks to minimize running time [6], [10].

While Hadoop and Spark are general data processing platforms, recent frameworks, such as Tensorflow and PSs, focus specifically on distributed machine learning (ML) jobs, given their increasing importance. While these platforms must also address stragglers, distributed ML jobs enable new approaches beyond speculative execution. Specifically, distributed ML jobs in some (but not all) cases admit relaxations to strict barrier semantics without significantly affecting ML accuracy. These relaxations range from simply executing tasks asynchronously [3], [11], [12] to allowing tasks to proceed past barriers a bounded amount [4], [13], [14]. Unfortunately, in some cases, these relaxations can impact the running time and accuracy of the ML algorithm. For example, since asynchronous processing provides no guarantee of convergence, it can extend running time indefinitely if the algorithm diverges. In this case, even though tasks spend less time waiting at each barrier, their running time and resource usage becomes infinite.

Importantly, while stragglers significantly degrade application performance, as indicated above, they also significantly increase application *cost* when run in the cloud. Cloud platforms charge users for the time they use a server at fine-grained intervals, e.g., every second or minute. As a result, any time tasks spend waiting idle at barriers results in resource waste that translates into a higher cost. Of course, some of the techniques for mitigating stragglers also incur additional costs, e.g., to acquire more cloud resources to execute backup replica tasks. In general, prior work focuses on dedicated clusters in data centers and thus *does not* consider cloud platforms' fine-

grained costs when evaluating their techniques.

Yet, cost, and *not performance*, is the dominant metric when using cloud platforms, as it is nearly always possible to increase performance by acquiring more cloud resources for an increased cost, although the relationship may not be linear. However, optimizing cost tends to differ from optimizing performance because cloud platforms offer servers under multiple different contract options. In particular, cloud platforms sell their excess capacity at highly discounted prices in the form of *transient servers* [15], which they may revoke at any time to satisfy new requests for *on-demand servers*, which platforms do not revoke. Due to their unreliability, transient servers often cost up to 90% less than on-demand servers. While parallel batch jobs are ideal candidates for running on cheap transient servers, revocations degrade their usable computational capacity, since applications must either re-execute work lost at each revocation or incur some overhead to implement fault-tolerance mechanisms, such as periodic checkpointing. The higher the revocation rate the lower a transient server's capacity for doing useful work, i.e., not related to re-executing tasks or fault-tolerance overhead [16].

Currently, cloud platforms *do not* reveal any information about the revocation characteristics of transient servers [16]. Google Cloud Platform (GCP) and Microsoft Azure charge a fixed-price per unit time for transient servers [17], [18]. Amazon's Elastic Compute Cloud (EC2) originally enabled users to bid on transient servers, called spot instances, such that they were revoked whenever the *spot price* exceeded the bid price [19]. The spot price was then determined based on supply and demand by conducting a uniform price auction across the available servers and bids. As a result, the spot price revealed historical revocation rates to users, and enabled some control over the tradeoff between revocation rate and cost by adjusting the bid. However, EC2 recently modified their spot pricing algorithm to make it more stable, such that it reflects only the long-term balance of supply and demand [20]. As a result, EC2 also no longer reveals transient server revocation characteristics, or allows any control over them.

Thus, transient servers essentially define a new type of cloud server that yields a probabilistic computational capacity, dictated by its unknown revocation characteristics, but where users pay a highly discounted fixed price per unit time. Despite the discounted price, using a transient server could result in a higher overall execution cost than using an on-demand server if it yields a low capacity (due to a high revocation rate), and thus takes significantly longer to complete a job. For parallel jobs, transient servers also increase the likelihood of stragglers compared to prior work, as any revocation results in straggling, which increases resource waste and further increases overall cost compared to using on-demand servers. While straggler mitigation techniques can reduce the cost of stragglers, they each impose their own cost, making it unclear which technique is optimal and whether any provide a net cost benefit.

To better understand the design space, we develop simple analytical models to quantify and compare the expected performance and cost of executing parallel jobs using different straggler mitigation techniques on both on-demand and transient cloud resources. In analyzing these models for a representative baseline application, we draw a number of conclusions.

Transient Servers offer Complex Tradeoffs. As the degree of parallelism increases, the performance and cost of stragglers can make using transient servers more expensive than using on-demand servers despite their high discount. However, determining whether transient servers provide a net benefit is not straightforward, as their cost is a complex function of numerous parameters, including the transient server discount, degree of parallelism, revocation rate, number of barriers, network overhead, straggler mitigation technique, etc.

Common Approaches Less Effective. Some common approaches for mitigating stragglers, such as bounded staleness [4], [13], [14], are ineffective when stragglers are the expected case, as with transient servers. Other common approaches, such as spawning backup tasks [6], [7], [10] and using partial barrier semantics [21], are more effective at reducing running time and cost but offer different tradeoffs.

Flexible Synchronization Performs Best. We find that a recent approach to Flexible Synchronous Processing (FSP) [22] offers the maximum speedup per cost (relative to using a single on-demand server) across all straggler mitigation techniques, although it presents some challenges, as we discuss.

II. Model Overview

We first provide an overview of our basic model, and then discuss representative baseline values for its parameters.

A. Basic Model

Table I shows the name and description of our model's parameters, including their measurement units and range.

We assume parallel jobs must complete some workload W that requires executing some number of abstract operations, which represent a collection of CPU instructions and I/O operations. Servers execute these operations at a rate s, in operations per unit time, based on their performance capacity. We normalize s relative to the capacity of an on-demand server of a specific type, i.e., with a specific CPU, memory, and I/O capacity, that experiences no revocations. Thus, on-demand servers of the specified type complete operations at a normalized rate of $s = 1$ operation per unit time. In contrast, transient servers of the same type, which do experience revocations, complete operations at a normalized rate $0 < s < 1$. Note that even though these transient servers have the same resource capacity as the on-demand server, they must devote some resources to handling revocations, either by re-executing work lost on revocations or executing fault-tolerance mechanisms, such as checkpointing. As a result, the rate at which a transient server can perform *useful* work is strictly less than an on-demand server with the same resources.

For transient cloud servers, we model s as a random variable, since revocation rates are not revealed by cloud platforms and are thus opaque to users. Ultimately, revocation rates are a function of the variance in the high-priority foreground workload, i.e., for on-demand servers. Prior analysis of

Name	Parameter	Description	Units	Range
Workload	W	Total workload to execute for job	#Operations	$W \geq 0$
Performance	s	Normalized computing speed of a cloud server	#Operations/time	$0 < s \leq 1$
Price	p	Price of an on-demand cloud server of type i ($s = 1$) per unit time	$/time	$c > 0$
Parallelism	k	Number of parallel tasks W is evenly distributed across	#	$k > 0$
Barriers	b	Number of synchronization barriers when executing W	#	$b \geq 0$
Network Overhead	n	Network communication overhead constant per barrier	time	$n \geq 0$
Discount Factor	f	Discount factor for transient cloud server of type i	%	$0 \leq f < 1$
Backup Replicas	r	Number of backup task replicas spawned to mitigate stragglers	#	$r \geq 0$
Staleness Parameter	d	Scaling factor that determines the work per barrier interval	#	$1 \leq p \leq b$
Drop Parameter	N	Number of slow tasks dropped each barrier	#	$N \geq 0$
Total Time	T	Total time to execute workload W	time	$T \geq 0$
Total Cost	C	Total cost to execute workload W	$	$C \geq 0$

TABLE I
NAME AND DESCRIPTION OF OUR MODEL'S PARAMETERS, INCLUDING THEIR UNITS AND RANGE.

publicly-available cluster traces has shown that this variance can yield a wide range of revocations rates across different transient servers [16]. Since realistic revocation characteristics are unknown, and for simplicity, our model assumes transient servers yield a uniformly random performance s between 0 and 1 with an average performance of $s = 0.5$. Prior analysis, which estimates revocation characteristics from cluster traces and spot prices (before EC2's recent change in their pricing algorithm), suggests this average performance is akin to running jobs on transient servers and simply re-executing work lost on a revocation without using fault-tolerance [16], [23]. Note that since a transient server's performance degradation derives from unknown revocation characteristics, users cannot measure its performance capacity s, and thus cannot simply identify and replace transient servers with low values of s.

As with today's cloud platforms, our model assumes that on-demand servers with $s = 1$ incur a fixed-price p in dollars per unit time, while transient servers with $0 < s < 1$ are discounted by a factor $0 < f < 1$, which results in a fixed-price of $(1 - f) \times p$. Public cloud platforms currently discount transient servers up to 90%, or $f = 0.9$. However, note that since we assume transient servers yield an average performance of only $s = 0.5$, the discount in the expected cost C to complete a job, when accounting for the performance overhead of revocations, is actually only $0.5 \times 0.9 = 45\%$, as the jobs must run for longer compared to on-demand servers.

We adopt a simple model for parallel jobs: their total workload W is initially divided evenly across b barrier intervals and k parallel tasks running on separate (on-demand or transient) servers. This model is consistent with platforms that use bulk synchronous processing (BSP), including Hadoop, Spark, and PSs. While some platforms, such as Tensorflow, Dryad, GraphLab, Naiad, etc., allow users to specify more complex execution patterns (based on arbitrary graphs), doing so is complex and users often revert to implementing simple BSP-style synchronization. Our model also assumes every barrier incurs some network overhead to communicate its results to other tasks. We model this overhead per barrier as being linear in the number of parallel tasks, as suggested in prior work [13], resulting in a total delay across all barriers of $n \times b \times k$, where n is a constant representing network overhead. We discuss the remaining model parameters from Table I in the next section in the context of specific straggler mitigation techniques.

Name	Parameter	Baseline Value
Workload	W	8000
Parallelism	k	8
Barriers	b	500
Network Overhead	n	0.175
Discount Factor	f	0.9

TABLE II
REPRESENTATIVE MODEL PARAMETER VALUES FOR BASELINE JOB.

Given our model above, we compute a parallel job's expected running time T and cost C to execute a workload W. The running time is simply the sum of the time for computing between barriers, including the time any servers spend waiting, and for communicating at barriers. Thus, using our notation from Table I, we derive the expected running time as below.

$$T = \frac{W}{s \times k} + (n \times b \times k) \qquad (1)$$

The first term ($\frac{W}{s \times k}$) is the workload (in number of operations) divided by the product of the number of parallel tasks (k) and the expected performance of each task (s) (in operations completed per unit time). This expression yields the job's expected computation time, assuming its workload W is evenly divided across k homogeneous servers with performance s. The second term ($n \times b \times k$) represents the communication delay, as discussed above.

Similarly, we also derive the cost as below using our parameters from Table I. Here, the cost C to execute W is simply the product of i) the server discount $(1 - f)$, ii) the server price p per unit time, iii) the number of servers k used, such that there is a one-to-one mapping between servers and tasks, and iv) the expected running time T from above.

$$C = (1 - f) \times p \times k \times T \qquad (2)$$

Note that, since we model s as a random variable with a uniform distribution, T and C represent expected values and are not deterministic. In Section III, we modify the expressions for T and C above when using on-demand and transient servers under different straggler mitigation techniques.

B. Representative Baseline Parameter Values

Our model above includes many parameters that affect a parallel job's completion time and cost. Rather than explore the entire parameter space, we define representative values for these parameters to serve as a baseline for comparison. Table II shows these representative values. We extract values for

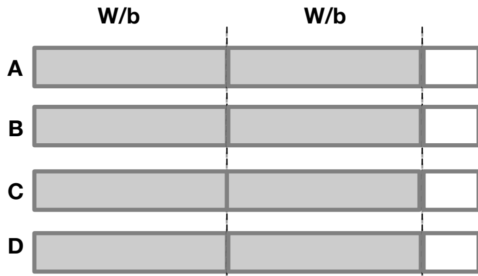

Fig. 1. Parallel job using BSP on on-demand servers.

workload (W), degree of parallelism (k), number of barriers (b), and network overhead (n) from experiments performed in recent work on stale synchronization for distributed ML [13]. These values are based on i) a distributed implementation of LDA Topic Modeling with collapsed Gibbs sampling, ii) using a dataset from the New York Times that includes 100 million tokens, and iii) executed on multi-core blade servers connected via 10Gpbs Ethernet, where each server has 8 cores, running at 2.3-2.5Ghz with 23GB RAM. This representative parallel job took ∼1700 seconds to complete using 8 parallel tasks ($k = 8$), 500 barriers ($b = 500$), and a network overhead constant of 0.175 seconds ($n = 0.175$). This results in a workload W of 8000 in our model when run on homogeneous on-demand servers using BSP, assuming no stragglers. As in [13], the job spends slightly more time computing compared to communicating at barriers. Finally, we use a baseline discount factor f of 0.9 based on the discounts for transient servers offered by Amazon, Google, and Microsoft.

C. Summary

As with any model, ours is not perfect and does not capture many job and resource characteristics that impact performance and cost. For example, unlike prior work, we only model stragglers caused by transient server revocations, and not other reasons. We assume stragglers due to frequent revocations dominate any effects from "naturally occurring" stragglers, which are rare. We also do not model the effect of different synchronization approaches on algorithmic running time and correctness. Specifically, we assume our workload W is fixed regardless of the synchronization approach, which may not be true for some problems, especially with distributed ML. Further, our baseline job above defines only a single point in a large parameter space. As a result, we intend our analysis to only highlight trends in job performance and cost for different synchronization approaches as the parameters change with a focus on scalability, i.e., an increasing degree of parallelism (k). Finally, we *do not* intend our model to be predictive, and thus, in practice, the precise performance and cost for even our baseline job may differ from our model's estimate.

III. Comparing Synchronization Models

Given the model from the previous section, we derive the expected running time T and cost C to execute a parallel job on on-demand and transient servers using different synchronization models and straggler mitigation techniques.

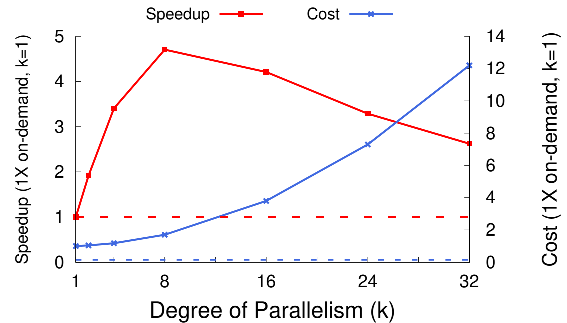

Fig. 2. Speedup and cost of executing our representative parallel job using BSP on on-demand cloud servers as the degree of parallelism increases.

A. BSP on On-demand Servers

The simplest case is to use bulk synchronous processing (BSP) on on-demand servers, as done by Hadoop, Spark, and other distributed data processing platforms. Since, in our model, on-demand servers experience no revocations, they exhibit no stragglers, and thus we do not employ any straggler mitigation technique in this case. Figure 1 depicts a parallel job using BSP on homogeneous on-demand servers, where each horizontal progress bar is a task running across time on a different server, and the vertical dotted lines represent barriers. The $\frac{W}{b}$ terms above the progress bars represent the expected work done by all tasks over each barrier interval.

In this case, each parallel task arrives at the barrier at precisely the same time, and thus there is no waiting or resource waste associated with stragglers. We simplify Equation 1 by setting $s = 1$ for on-demand servers, yielding an expected[1] running time using BSP on on-demand servers as follows.

$$T = \frac{W}{k} + (n \times b \times k) \tag{3}$$

Similarly, the discount is $f = 0$ for on-demand servers, as it only applies to transient servers. Thus, we simplify Equation 2 by removing the $(1 - f)$ term, yielding a total cost as follows.

$$C = p \times k \times T \tag{4}$$

Figure 2 then plots the speedup (left y-axis) and cost (right y-axis) of executing our representative parallel job from Section II-B as k increases. Here, the speedup and cost is normalized relative to their values when running the job on a single on-demand server with no barriers, i.e., $s = 1$, $k = 1$, and $b = 0$. The graph shows that as we increase k, the speedup, as with any parallel job, increases up to a point where the communication delay at each barrier begins to offset the benefit of using more resources. Even so, parallelization provides a clear performance advantage up to and beyond $k = 32$ with a maximum speedup near 5× at $k = 8$.

In contrast, the overall execution cost C rises dramatically as k increases, as the increasing communication delays at barriers result in wasted time where the parallel job is paying for computing capacity but not using it. Note that, under our model, increasing k when using on-demand servers can never decrease cost: even if there were no communication

[1] In this case, s is actually deterministic.

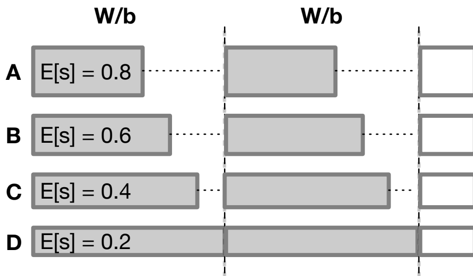

Fig. 3. Parallel job using BSP on transient servers.

delays at barriers, the cost C would remain constant as the decreased time spent computing would be exactly offset by more resources being used in parallel over that time. From the equations above, as $k \rightarrow \infty$, the cost C is $O(k^2)$. Thus, when using 32 tasks, the job gains a speedup of 2.6× but costs 12.2× more compared to using a single on-demand server.

Figure 2 illustrates how users *must* consider both performance and cost when executing parallel jobs in the cloud. Since cloud platforms provide users limitless resources, users can *always* improve their jobs' performance by purchasing more resources, but for an increasingly higher cost.

Result: *Increasing the degree of parallelism k when using on-demand servers with BSP improves performance up to a point, but at an increasingly high cost that scales super-linearly.*

B. BSP on Transient Servers

Based on the high cost above of using BSP with on-demand servers, transient servers are a potentially attractive option for executing large-scale parallel batch jobs due to their low price. However, as discussed in Section II-A, revocations decrease their usable performance capacity s, which we model as being uniformly random in the range $[0, 1]$. Importantly, recall that under BSP the slowest straggler task to reach a barrier dictates when all other (faster) tasks can proceed past the barrier. Thus, while cheaper, transient servers cause stragglers that reduce performance by increasing waiting time and resource waste. We must account for this waste and transient servers' discount when computing their expected running time and cost.

To do so, we must determine the expected speed of the slowest server: since we assume transient server performance s is uniformly distributed, this is equivalent to finding the expected minimum value when drawing k uniformly random numbers in the range $[0, 1]$. To determine this value, we note that for any value of k, given our assumption, the expected performance s across all of the k servers should be uniformly distributed. As a result, the expected performance for the fastest of k servers should be $s = \frac{k}{1+k}$ and for the slowest server should be $s = \frac{1}{1+k}$. Since the expected performance of *all* transient servers is dictated by the performance of the slowest expected server, we can reduce Equation 1 for expected overall running time for a parallel job under transient servers to the following, where we simply substitute s with $\frac{1}{1+k}$. The cost C remains the same as in Equation 2.

$$T = \frac{W(1+k)}{k} + (n \times b \times k) \quad (5)$$

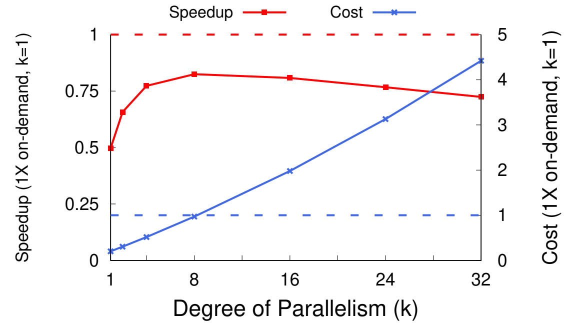

Fig. 4. Speedup and cost of executing our representative parallel job using BSP on transient cloud servers as the degree of parallelism increases.

Figure 3 illustrates the expected performance s for each transient server when $k = 4$ under our model. The figure shows that, under our model, some transient servers will yield better expected performance than others due to experiencing fewer revocations, such that the overall performance and cost is dictated by the slowest server, and where faster servers waste resources by waiting at barriers. Here, the progress bar's width represents each task's expected performance s,[2] and the area of the progress bar represents total work, such that within each barrier interval, the area of each progress bar is equal.

Similar to Figure 2, Figure 4 plots the speedup (left y-axis) and cost (right y-axis) of executing our representative parallel job on transient servers as k increases. The graph shows that a single transient server incurs an expected speedup of 0.5× (or equivalently a slowdown of 2×), since it runs at half the expected speed of an on-demand server. As k increases, similar to above, expected performance increases up to a point where communication delays offset the benefit of adding more servers. However, based on Equation 5, as $k \rightarrow \infty$, the expected speedup with transient servers can never exceed that of using a single on-demand server with $s = 1$. However, despite their low performance, due to their 90% discount, transient servers offer a lower overall cost than using a single on-demand server for up to $k = 8$.

Result: *Increasing the degree of parallelism k when using transient servers with BSP improves cost up to a point (based on their discount), but incurs an increasingly high performance penalty due to their lower and non-uniform expected performance, which diminishes their cost advantage.*

C. BSP on Transient Servers with Backup Replica Tasks

As mentioned in Section I, prior work proposes handling stragglers by identifying them, submitting a backup replica task for them, and then accepting the result of whichever task finishes first (and cancelling the other task) [6], [7], [10]. We model this approach by assuming that we can always immediately identify the slowest server(s) and submit backup replica tasks for them. While this assumption is not realistic, since we cannot assess transient server performance due to their unknown revocation rates, it serves as an upper bound on the performance and cost advantage of using backup tasks.

[2] In all figures, we denote performance using $E[s]$ to emphasize that transient server performance is an expected value, and not deterministic.

Fig. 5. Parallel job using BSP on transient servers with backup replica tasks where $k = 4$ and $r = 1$.

Figure 5 illustrates a parallel job using BSP on transient servers with backup tasks, where our degree of parallelism $k = 4$ and the number of replicas $r = 1$. The red X represents that the replica task is cancelled once one of the replicas reaches the barrier. Note that the other replica is cancelled in the second barrier interval to illustrate that performance of transient servers is probabilistic and can change over time. As shown, the replica effectively increases the expected speed of the slowest task from $s = 0.17$ to $s = 0.33$, by allowing us to discard the slowest task, but at an additional cost for the replica. Thus, the expected performance s of the slowest task is a function of both the degree of parallelism k and the number of replicas r, as shown below.

$$s = \frac{1+r}{1+k+r} \tag{6}$$

To derive the expected running time for this approach, we substitute s above into Equation 1, which yields the following.

$$T = \frac{W(1+r+k)}{k(1+r)} + (n \times b \times k) \tag{7}$$

In addition, we must also consider the number of replicas when deriving the overall cost C by multiplying by $(k + r)$ servers rather than the k servers in Equation 4, as shown below.

$$C = (1 - f) \times c \times T \times (k + r) \tag{8}$$

Figure 6 then plots the speedup (top) and cost (bottom) of executing our parallel job on transient servers with different numbers of replicas as k increases. Note that $r = 0$ represents using BSP on transient servers with no replicas and is equivalent to Figure 4's speedup and cost. The graph shows that spawning backup replica tasks improves both speedup and cost relative to not using replicas. In addition, unlike with no replicas, backup replica tasks enable the speedup to exceed $1\times$. However, replicas introduce some interesting tradeoffs. As expected, using more replicas always increases the speedup, as shown in the top figure, but the number of replicas that minimizes the cost is unclear, as $r = 4$ replicas is cheaper than both $r = 1$ replica and the extreme case of $r = k$ replicas. Overall, using replicas widens the values of k that yield both a speedup and cost advantage compared to not using replicas.

Result: *Increasing the degree of parallelism k when using transient servers with BSP and backup task replicas strictly improves the speedup and cost compared to not using replicas, and enables speedups greater than* 1.

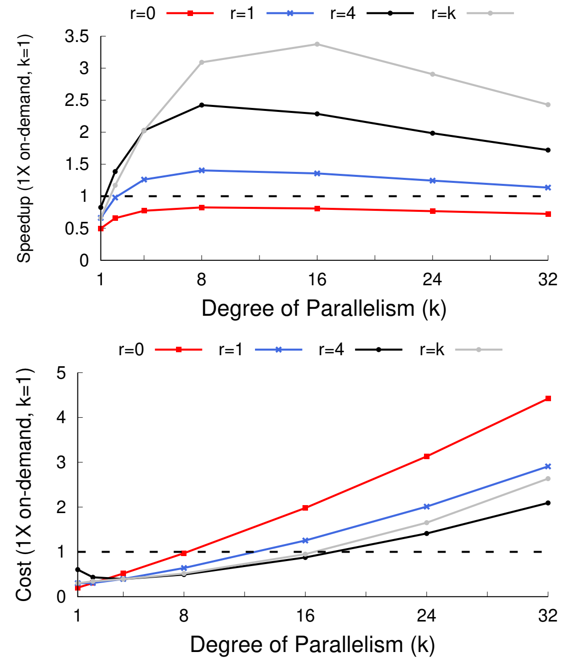

Fig. 6. The speedup (top) and cost (bottom) of executing our representative parallel job with different numbers of backup replica tasks using BSP on transient cloud servers as the degree of parallelism increases.

D. Bounded Staleness on Transient Servers

Another approach for mitigating the impact of stragglers is to enable threads to perform a bounded amount of work past each barrier [14], [13], [4]. This approach reduces the time that fast tasks wait for slow ones, and can also reduce communication costs, as it effectively reduces the number of "real" barriers. The tradeoff is that, for distributed ML in particular, some tasks may access "stale" global parameter values that do not reflect all tasks' updates, which may impact a job's algorithmic convergence time and accuracy. As stated in Section II, our model does not capture these algorithmic tradeoffs, which can affect running time and cost.

There are many variants of this general approach with slight differences in the context of distributed ML, including Arbitrarily-sized BSP (A-BSP) [14], stale synchronous processing (SSP) [13], and the bounded delay model [4]. However, as we discuss, in the context of our simplified model, these variants are equivalent. We model this approach by simply introducing a staleness parameter d, similar to the one in [13], that reduces the number of barriers by a factor d. As a result, the expected running time T is the same as in Equation 5 but substituting $\frac{b}{d}$ for b, as shown below. The cost C is the same as Equation 2, but includes the T below.

$$T = \frac{W(1+k)}{k} + (n \times \frac{b}{d} \times k) \tag{9}$$

Figure 7 illustrates bounded staleness: typically, the barriers would be defined by when the slowest task, in this case D, completes its work, indicated by the vertical line in D's progress bar. However, with bounded staleness, the other tasks may perform a bounded amount of work past the barrier, which has the effect of making the effective barrier intervals longer.

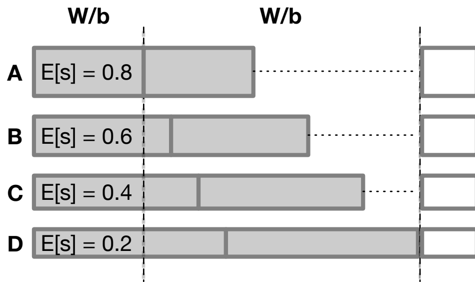

Fig. 7. Parallel job using bounded staleness on transient servers.

Figure 8 shows the speedup (top) and cost (bottom) as the degree of parallelism changes under bounded staleness with different staleness parameters d. As expected, increasing the staleness parameter increases the speedup and decreases the cost by reducing the communication delays. However, even for the maximum value of $d = 500$, resulting in a single barrier, there is never a speedup compared to using a single on-demand server, and the cost becomes higher once $k > 8$. Bounded staleness is also worse in terms of both speedup and cost when compared to using backup replica tasks.

Bounded staleness is really only effective at mitigating the impact of stragglers that are rare and temporary. In these scenarios, the expected case is similar to using BSP with on-demand servers. Bounded staleness enables tasks to reduce (or eliminate) any waiting and waste that might occur due to a few temporary stragglers. In contrast, when analyzing our model of transient servers, stragglers *are the expected case* and thus the fast tasks *always* must eventually wait for the stragglers after proceeding a bounded amount past a barrier. In this case, bounded staleness provides little benefit beyond reducing communication delays related to the number of barriers, which are not dominant at values of $k \leq 32$ in the graph.

Result: *Increasing the degree of parallelism k when using transient servers with BSP under bounded staleness is worse in terms of speedup and cost than using backup replica tasks.*

E. Partial Barriers on Transient Servers

The previous approaches extend the BSP model to mitigate stragglers. Prior work has also proposed "looser" synchronization models that relax the strict barrier semantics of BSP [21]. As one example, *partial barriers* mitigates stragglers by releasing the barrier once some number of tasks have reached it, and then cancels (or drops) the other tasks and re-distributes their work across all the servers. While prior work dynamically determines this release point based on the arrival rate of tasks to barriers, our model simply defines a drop parameter N, such that we release the barrier once $k - N$ tasks have reached it.

While such partial barriers are not applicable to all parallel jobs, they are applicable to distributed ML, as well as other examples cited in prior work [21]. Unlike with prior approaches, modeling partial barriers requires us to shift the cancelled work of slow tasks to the next barrier interval. Thus, the amount of work assigned to tasks per barrier interval increases as the job progresses. Our model redistributes this cancelled work equally across all servers in the next barrier interval. This results in work from slow tasks being shifted to faster servers.

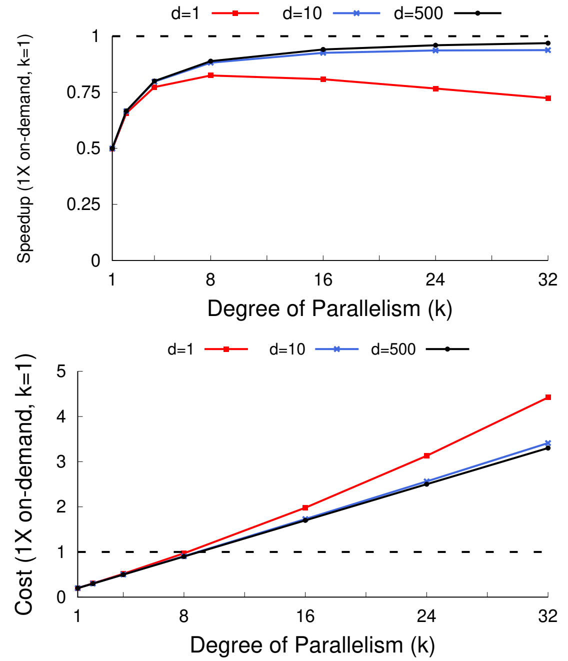

Fig. 8. The speedup (top) and cost (bottom) of executing our representative parallel job with different staleness parameters d under bounded staleness on transient cloud servers as the degree of parallelism increases.

We derive the expected running time T to complete a parallel job when using partial barriers as below.

$$T = \frac{W(1+k)}{b \times k \times (1+N)} \times [\sum_{i=2}^{b} [b-(i-1)](\frac{N}{k})^{i-2}] + \frac{W(1+k)}{b \times k} \times [\sum_{i=1}^{b} (\frac{N}{k})^{i-1}] + (n \times b \times k) \quad (10)$$

While we omit a full explanation due to space limitations, the first additive (top) term of this equation is similar to Equation 7 for the expected time when using backup replica tasks. Essentially, by dropping slow tasks, the overall speed becomes a function of the slowest non-dropped task, which is dictated by N (as opposed to r in Equation 7). However, unlike with backup replicas, we must account for the dropped work, which gets added to the work done next barrier interval and is equally distributed across all servers. The summation in the first term is the sum of the expected work that gets shifted to each barrier interval. The second additive term represents the expected running time required to finish the job after the final barrier, which is dictated by the speed of the slowest server, as we do not permit slow tasks to be dropped after the final barrier. This is why there is no N in the second additive term. The last term is the same communication delay as before.

The expected cost C is simply $(1 - f) \times p \times T \times k$ as in the basic model, as this approach, unlike with replicas, uses no additional servers. Figure 9 illustrates this approach for $k = 4$, where the slowest task is dropped at the first barrier and its work is shifted to the next interval, as indicated by the red term added to the work that interval. In this case, the additional work shifted is $W/4b$ since this was the work assigned to D in the first interval. The figure shows a different task being

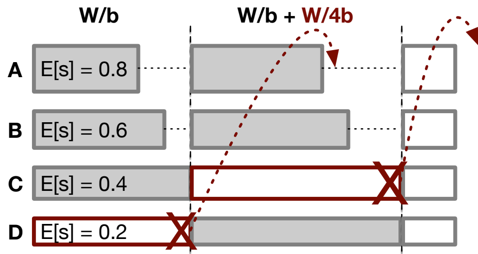

Fig. 9. Parallel job using partial barriers on transient servers.

cancelled in the second barrier interval to emphasize that these are expected speeds, and the actual speed of a transient server is probabilistic and can change (based on its revocations). Thus, the "slow" server may differ each interval.

Figure 10 shows the speedup (top) and cost (bottom) of using partial barriers on transient servers for different values of N. N defines a tradeoff such that higher values increase the speed of the "slowest" server that reaches the barrier before it is released, but it requires cancelling and re-executing more work in the next barrier interval, which increases resource waste. In this case, the extreme points ($N = 0$ and $N = k-1$) result in nearly the same speedup and cost, while setting $N = 1$ improves both the speedup and cost. As shown, $N = k/2$ results in the optimal speedup and cost, which are comparable to the speedup and cost when using the optimal number of backup replica tasks (see Figure 6). Since $N = 0$ represents using BSP on transient servers, partial barriers offers a clear advantage in terms of speedup and cost, similar to using backup replica tasks.

Compared to using backup replica tasks, partial barriers offer a slightly lower maximum speedup ($\sim$2.5$\times$ versus $\sim$3.5$\times$) for a marginally lower cost ($\sim$0.75$\times$ versus $\sim$1$\times$). As we discuss later, using backup replica tasks offers a better speedup/cost tradeoff at low values of k, while partial barriers is better at higher values of k. However, one advantage of partial barriers over using backup replica tasks is that the latter is speculative, and requires jobs to first identify slow tasks, while the former is not. Our model in Section III-C assumes an ideal case where jobs can immediately identify slow tasks and replicate them. Thus, in practice, the speedup and cost of using replicas is likely to be worse than in our idealized model. However, one disadvantage of partial barriers is that it may require algorithmic and implementation changes, since it alters the synchronization model.

Result: *Increasing the degree of parallelism k when using transient servers with partial barriers strictly improves the speedup and cost compared to BSP, and enables speedups greater than 1. The approach has a comparable speedup and cost as using backup replica tasks.*

F. Flexible Synchronization on Transient Servers

Recent work has introduced a flexible synchronous processing model [22]. FSP proposes a synchronization model that initiates synchronization barriers dynamically based on the progress of the tasks. Thus, if it identifies stragglers, FSP can dynamically initiate a synchronization barrier, and

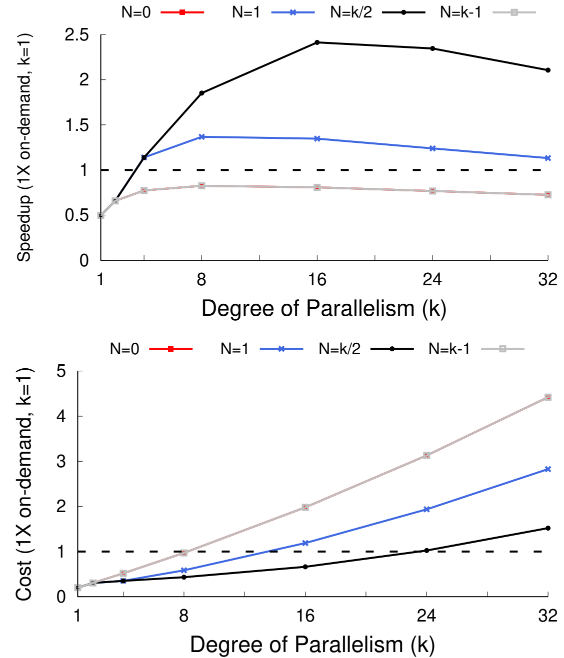

Fig. 10. The speedup (top) and cost (bottom) of executing our representative parallel job using partial barriers for different numbers of dropped slow tasks N as the degree of parallelism increases.

allow the fast tasks to continue execution. We model FSP similar to partial barriers, but where jobs do not have to re-execute the work of "dropped" tasks at each barrier. In this case, once $k - N$ tasks have reached a barrier, the job initiates synchronization among all k tasks, allowing all tasks to proceed past the barrier. The remaining work of these slow tasks is then re-distributed across all the servers. We note that the original FSP model *does not* redistribute work, since it targets "naturally occurring" stragglers that are temporary and rare. Since stragglers due to transient servers are expected and frequent, we must distribute each interval to gain any speedup. Thus, using FSP in practice on transient servers would require some changes. As with partial barriers, prior work on FSP uses a more sophisticated approach that dynamically determines barrier points by monitoring task progress.

Figure 11 illustrates using FSP on transient servers with $N = 1$. The figure is the same as with partial barriers (Figure 9) except that the additional work shifted to the next barrier interval is lower, since not all the work has to be redone. In both cases, the expected case is that half the work assigned to the N slowest servers is completed. Thus, the expected running time below is equivalent to that of partial barriers (from Equation 10), except that FSP only has to shift and re-distribute the remaining half of the work to be completed in the next barrier interval. This is the reason for the additional 2 in the denominator compared to Equation 10.

$$T = \frac{W(1+k)}{b \times k \times (1+N)} \times [\sum_{i=2}^{b}[b-(i-1)](\frac{N}{2k})^{i-2}] + \frac{W(1+k)}{b \times k} \times [\sum_{i=1}^{b}(\frac{N}{2k})^{i-1}] + (n \times b \times k) \quad (11)$$

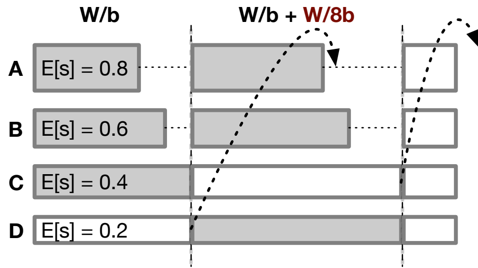

Fig. 11. Parallel job using FSP on transient servers.

Figure 12 shows the speedup (top) and cost (bottom) of using FSP on transient servers for different values of N. Of course, FSP is strictly better than partial barriers because it is equivalent, except that it does not waste resources re-executing work. In addition, FSP offers a maximum speedup near that of using backup replica tasks, but for a lower cost (see Figure 5).

While FSP offers the best performance and cost, it does pose some challenges. In particular, FSP was developed in the context of distributed ML, and, as with bounded staleness and partial barriers, may impact algorithmic convergence time and accuracy, which we do not model. In addition, as with partial barriers, FSP is a new synchronization model that likely requires algorithmic and implementation changes. Prior work has only applied FSP to specific problems, e.g., Expectation-Maximization (EM) [22] and Stochastic Gradient Descent [24]. As a result, FSP's generality is not yet clear.

Result: *Increasing the degree of parallelism k when using transient servers with FSP yields the best speedup and cost among the straggler mitigation techniques we model.*

G. Summary

Our analysis shows that users must jointly consider both speedup and cost when deciding whether and how to use transient servers for parallel jobs. While different users may value speedup and cost differently, Figure 13 plots the speedup/cost ratio for all of the straggler mitigation techniques above as the degree of parallelism k increases Since a large speedup and a low cost are preferable, higher values of the speedup/cost ratio indicate a better "bang for your buck" for parallel jobs. Interestingly, using BSP with on-demand servers is not the worst option, as using BSP with transient servers and no replicas has a lower speedup/cost for all but the smallest values of k, despite their high discount. Using partial barriers and using backup replica tasks offer a speedup/cost ratio in the middle. In this case, we use parameter values of $N = k/2$ and $r = k$, respectively, which yield the maximum speedup/cost ratio for these techniques. At lower values of k, backup replica tasks offer a higher speedup/cost ratio, while at larger values of $k \geq 16$ partial barriers yield a higher ratio.

As mentioned above, using FSP with transient servers yields the highest speed/cost ratio across all values of k. In addition, since the use of backup replica tasks is not mutually exclusive to using FSP, we were interested in whether combining these techniques offered any advantage. We omit the equation for T for this hybrid technique, as it is complex, but show the result in Figure 14 for different combinations of N and r. In

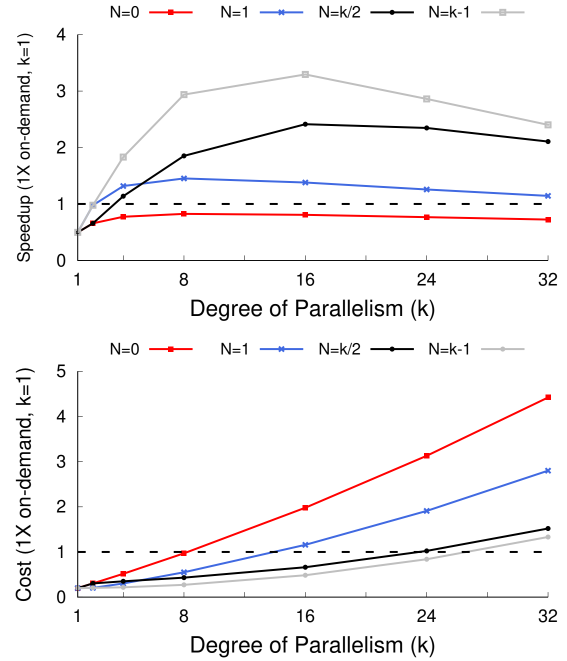

Fig. 12. The speedup (top) and cost (bottom) of executing our representative parallel job using flexible synchronous processing (FSP) for different numbers of dropped slow tasks N as the degree of parallelism increases.

all cases, we set $N = k - 1$, since it is optimal, as there is no reason for any task to ever wait with FSP. The result shows that using backup replicas in combination with FSP *does not* offer an advantage in overall speedup/cost. Replicas only add an additional cost, but offer no advantage in terms of running time, since FSP need not wait on stragglers anyway.

IV. Discussion and Future Work

Transient servers increase the severity and frequency of stragglers, making them expected rather than rare, as implicitly assumed in prior work [10], [6], [7]. Our analysis also differs from prior work on straggler mitigation in its focus on cost as a primary metric, in addition to performance. We view our model and analysis as only a starting point in understanding how to optimize the use of transient servers for parallel jobs, such as distributed ML. Our model is imperfect and does not account for the effect on the total work W (and its accuracy) from using different synchronization models, particularly for distributed ML. This effect is important, since without it, the optimal approach is to simply run parallel jobs asynchronously, which some frameworks do [3], [11]. However, this effect is difficult to analytically model because it varies, in part, based on the characteristics of a job's input data and initial conditions, as well as other algorithm-specific parameters, such as the mini-batch size. Instead, such effects must be evaluated empirically as done in prior work [4], [13], [14]. While we intend our model to only highlight high-level differences between straggler mitigation techniques, as part of future work, we plan to refine and validate our model via experimentation on real cloud platforms.

Our analysis is sensitive to the simple uniformly random model of transient server performance s we use. A different

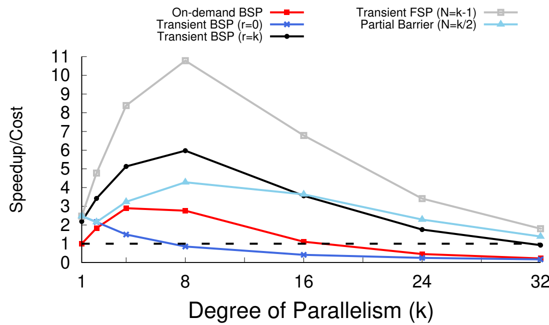

Fig. 13. The speedup/cost ratio for executing our baseline parallel job for different straggler mitigation techniques on on-demand and transient servers.

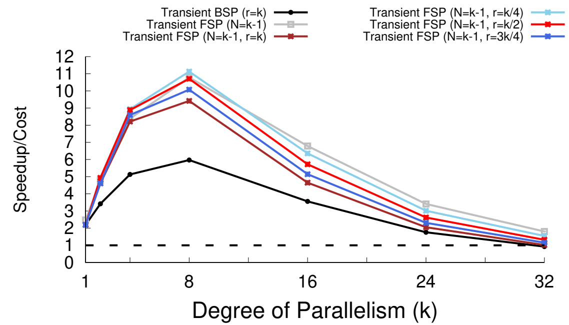

Fig. 14. The speedup/cost ratio for executing our baseline parallel job for a hybrid straggler mitigation technique that combines FSP with backup tasks.

distribution of s may change our findings. For example, an average s higher than 0.5, which would result from lower revocation rates, would increase the benefit of using transient servers for all techniques relative to using a single on-demand server. Similarly, a heavy-tailed distribution of s with a higher likelihood of selecting a high performing transient server may yield lower expected costs, especially for small values of k. Since deriving simple closed-form equations for running time T under arbitrary distributions may not be possible, we plan to compare approaches empirically via randomized simulation. Similar randomized simulations could also be applied to distributions derived from real revocation data, if cloud platforms were to release such data. Even so, transient server revocation characteristics may vary across different cloud platforms and data centers, and yield different tradeoffs. Note that, as more users optimize for and use transient servers, the revocation rates may also change due to second order effects.

Most prior work on optimizing batch jobs for transient servers has focused on configuring fault-tolerance techniques, such as replication and checkpointing, to minimize the impact of revocations on performance [23], [25], [26], [27], [28]. Our model abstracts this problem away by only considering the normalized speed s of a transient server after accounting for any fault-tolerance overhead and re-execution of lost work, which is strictly less than that of an on-demand server with an equivalent resource capacity. As our work shows, accounting for both a transient server's price discount f and its lower effective speed s when determining its overall cost savings to execute some workload is important. Since different transient servers may experience revocations at different times, clusters of transient servers are highly heterogeneous, exhibiting non-uniform performance. Thus, optimizing parallel jobs for transient servers is related to optimizing them for highly heterogeneous resources, which is a well-studied topic [10], [29]. The primary difference in the cloud is accounting for transient servers' probabilistic speed and high cost discount.

Our work also highlights some of the disadvantages of EC2's recent change in their spot pricing algorithm for transient servers [20]. Before this change, users that bid the same price on the same type of spot instances could guarantee they had uniform revocation characteristics, and thus a uniform normalized speed s, as revocations only occurred when the spot price exceeded the bid price. The new spot pricing algorithm does not require users to bid, and decouples revocations from the spot price, as the spot price only changes based on long-term changes in supply and demand. EC2 made this change to reduce revocation rates, as users out-bidding other users resulted in excessive revocations that reduced transient server performance. However, while the new approach reduced revocations, and increased overall transient server performance, it also eliminated useful knobs for users to control revocation characteristics that were important in maximizing performance. Thus, determining the best model under which to offer transient servers remains an open question.

Our model is analytical and focuses on a parallel job with the simplest possible structure. As part of future work, we plan to extend the model to more complex and realistic parallel jobs. For example, prior work [26] has shown that some parallel tasks are more likely to cause cascading recomputations, and so placing these tasks on more reliable on-demand servers is important.

V. Conclusions

We analyze the speedup and cost of executing parallel batch jobs, such as distributed ML jobs, on highly discounted transient cloud resources using many different straggler mitigation techniques. We do so in the context of a simple probabilistic model for transient server performance. Using this model, we derive the expected running time and cost for straggler mitigation techniques proposed in prior work for a simple parallel job with synchronization barriers. A key difference between our work and prior work on straggler mitigation is our focus on cost, rather than performance, on cloud platforms. Our analysis shows that i) transient servers offer complex tradeoffs compared to using on-demand servers, and can result in higher overall costs despite their highly discounted price due to their probabilistic performance; ii) common approaches to straggler mitigation, which is a well-studied problem, are less effective using transient servers that cause frequent and severe stragglers; and iii) a recent approach to flexible synchronization [22], [24] offers the best speedup per cost across all the techniques we study.

Acknowledgements. This work is funded by NSF grants #1802523, #1815412, #1763834, #1836752, and #1405826, as well as DOD ARL grant W911NF-17-2-019 and the Amazon AWS Cloud Credits for Research program.

References

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *MSST*, May 2010.

[2] 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 *OSDI*, April 2012.

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *OSDI*, November 2016.

[4] M. Li, D. Andersen, J. W. Park, A. Smola, A. Ahmed, V. Josifovski, J. Long, E. Shekita, and B.-Y. Su, "Scaling Distributed Machine Learning with the Parameter Server," in *OSDI*, November 2014.

[5] L. Valiant, "A Bridging Model for Parallel Computation," *CACM*, vol. 33, no. 8, August 1990.

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

[7] J. Dean and S. Ghemawatt, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, December 2004.

[8] F. Petrini, D. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on 8,192 Processors of ASCI Q," in *SC*, November 2003.

[9] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale," in *Cluster Computing*, September 2006.

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

[11] F. Niu, B. Recht, C. Re, and S. Wright, "HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," in *NIPS*, December 2011.

[12] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an Efficient and Scalable Deep Learning Training System," in *OSDI*, November 2014.

[13] Q. Ho, J. Cipar, H. Cui, J. K. Kim, S. Lee, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing, "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server," in *NIPS*, ser. NIPS, 2013.

[14] H. Cui, J. Cipar, Q. Ho, J. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. Ganger, P. Gibbons, G. Gibson, and E. Xing, "Exploiting Bounded Staleness to Speed Up Big Data Analytics," in *USENIX ATC*, June 2014.

[15] R. Singh, P. Sharma, D. Irwin, P. Shenoy, and K. Ramakrishnan, "Here Today, Gone Tomorrow: Exploiting Transient Servers in Data Centers," *IEEE Internet Computing*, vol. 18, no. 4, July 2014.

[16] S. Shastri, A. Rizk, and D. Irwin, "Transient Guarantees: Maximizing the Value of Idle Cloud Capacity," in *SC*, November 2016.

[17] "Microsoft Azure Low-priority VMs," https://azure.microsoft.com/en-us/pricing/details/batch/, May 2018.

[18] "Google Preemptible Instances," https://cloud.google.com/compute/docs/instances/preemptible, May 2018.

[19] "Amazon Spot Instances," https://aws.amazon.com/ec2/spot/, May 2018.

[20] R. Pary, "New Amazon EC2 Spot pricing model: Simplified purchasing without bidding and fewer interruptions," https://aws.amazon.com/blogs/compute/new-amazon-ec2-spot-pricing/, March 13th 2018.

[21] J. Albrecht, C. Tuttle, A. Snoeren, and A. Vahdat, "Loose Synchronization for Large-scale Networked Systems," in *USENIX ATC*, June 2006.

[22] Z. Wang, L. Gao, Y. Gu, Y. Bao, and G. Yu, "FSP: Towards Flexible Synchronous Parallel Framework for Expectation-Maximization based Algorithms on Cloud," in *SoCC*, September 2017.

[23] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy, "Flint: Batch-Interactive Data-Intensive Processing on Transient Servers," in *EuroSys*, April 2016.

[24] G. Zhao, L. Gao, and D. Irwin, "Sync-on-the-fly: A Parallel Framework for Gradient Descent Algorithms on Transient Resources," in *BigData*, December 2018.

[25] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda, "TR-Spark: Transient Computing for Big Data Analytics," in *SoCC*, October 2016.

[26] Y. Yang, G.-W. Kim, W. W. Song, Y. Lee, A. Chung, Z. Qian, B. Cho, and B.-G. Chun, "Pado: A Data Processing Engine for Harnessing Transient Resources in Datacenters," in *EuroSys*, April 2017.

[27] P. Sharma, D. Irwin, and P. Shenoy, "Portfolio-driven Resource Management for Transient Cloud Servers," in *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2017.

[28] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy, "SpotOn: A Batch Computing Service for the Spot Market," in *SoCC*, August 2015.

[29] F. Ahmad, S. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Tarazu: Optimizing MapReduce on Heterogeneous Clusters," in *ASPLOS*, April 2012.