Straggler Mitigation at Scale

Mehmet Fatih Aktaş and Emina Soljanin, *Fellow, IEEE*

***Abstract*— Runtime performance variability has been a major issue, hindering predictable and scalable performance in modern distributed systems. Executing requests or jobs redundantly over multiple servers have been shown to be effective for mitigating variability, both in theory and practice. Systems that employ redundancy has drawn significant attention, and numerous papers have analyzed the pain and gain of redundancy under various service models and assumptions on the runtime variability. This paper presents a cost (pain) vs. latency (gain) analysis of executing jobs of many tasks by employing replicated or erasure coded redundancy. The tail heaviness of service time variability is decisive on the pain and gain of redundancy and we quantify its effect by deriving expressions for cost and latency. Specifically, we try to answer four questions: 1) How do replicated and coded redundancy compare in the cost vs. latency tradeoff? 2) Can we introduce redundancy after waiting some time and expect it to reduce the cost? 3) Can relaunching the tasks that appear to be straggling after some time help to reduce cost and/or latency? 4) Is it effective to use redundancy and relaunching together? We validate the answers we found for each of these questions via simulations that use empirical distributions extracted from a Google cluster data.**

***Index Terms*— Coded and replicated redundancy, straggler relaunch, cost vs. latency tradeoff in distributed computing.**

I. Introduction

PROVIDING *predictable* performance is an important ongoing challenge for distributed computing systems. In distributed settings, a *job* is split into multiple smaller *tasks*, which get spread over separate resources for parallel execution. Task execution times in modern systems are known to exhibit significant runtime variability due to many factors such as power management, software or hardware failures, maintenance, and most importantly, resource sharing [3]–[9]. Runtime variability may cause some tasks to *straggle* and take much longer to complete than other tasks in the job. Since a distributed job completes only when all its tasks complete, straggler tasks significantly delay the job completion. As the number of tasks in a job increases so does the chance that at least one of them will be a straggler, thus the impact of stragglers on the job completion time is greater at scale [7], [10].

Straggler problem has received significant attention from the systems research community. Existing solution techniques fall into two categories: i) Squashing runtime variability via preventive actions such as blacklisting faulty machines that frequently exhibit high variability [4], [10] or learning the characteristics of task-to-node assignments that lead to high variability and avoiding such problematic task-node pairings [11], ii) Speculative execution by launching the tasks together with replicas and waiting only for the fastest copy to complete [5], [12]–[15]. Because runtime variability is caused by intrinsically complex reasons, preventive measures for stragglers could not fully solve the problem and runtime variability continued plaguing the compute workloads [10], [14]. Speculative task execution on the other hand has proved to be an effective remedy, and indeed the most widely deployed solution for stragglers [7], [16]. For instance with task replication, median runtime slowdown experienced by the tasks within a job is brought down from 8 (and 7) to 1.08 (and 1.1) in Facebook's production Hadoop cluster (and Bing's Dryad cluster) [16].

Executing tasks with greater number of copies will surely reduce the chance of having to wait for a straggler. However, task replicas occupy system resources that could otherwise be used to execute other tasks. Furthermore, if task replicas are employed excessively, they can overburden the system and further aggravate the runtime variability, given that the primary cause of runtime variability is resource sharing. Therefore, replicas are employed with care in practice, e.g., replica tasks are used only for jobs with a few tasks [14], or only tasks that straggle beyond some threshold are replicated [12]. More recently, replicas are proposed to be dispatched for single-task jobs only if any server is found idle, which is shown, with a queueing theoretic analysis, to not drive the system to instability by dispatching excessive number of replicas [17].

This paper focuses on two important performance metrics for distributed job execution: 1) *Latency,* measuring the time to complete the job, and 2) *Cost,* measuring the total resource time spent to execute the job. Job execution is desired to be fast and with low cost, but these are often conflicting objectives. Cost of executing a job depends on the number of tasks[1] and the time each task takes to finish. Executing a job with task replicas is expected to reduce the time spent by the tasks in the system, while also increasing the total number of tasks involved in completing the job, which is likely to increase the cost. It is important to understand the effect of added redundancy not only on the latency but also on the cost because the load exerted on the system by a job execution is determined by its cost (as elaborated in Sec. II-A).

Manuscript received September 7, 2018; revised May 6, 2019 and September 6, 2019; accepted September 11, 2019; approved by IEEE/ACM Transactions on Networking Editor B. Shrader. Date of publication October 28, 2019; date of current version December 17, 2019. This work was supported by the National Science Foundation under Grant CIF-1717314. A preliminary version of this paper appeared in the ACM SIGMETRICS Performance Evaluation Review [1], [2]. *(Corresponding author: Mehmet Fatih Aktaş.)*

The authors are with the Department of Electrical and Computer Engineering, Rutgers University, Piscataway, NJ 08854 USA (e-mail: mehmet.aktas@rutgers.edu; emina.soljanin@rutgers.edu).

This article has supplementary downloadable material available at http://ieeexplore.ieee.org, provided by the authors.

Digital Object Identifier 10.1109/TNET.2019.2946464

[1]Resource usage of tasks vary across different jobs or might vary even within the same job in practice [18]. We abstract this complexity by assuming that each task uses one unit of resource per unit time.

Erasure coding implements a more general form of redundancy than simple replication, and has been considered for straggler mitigation both in data download [19]–[21], and more recently in distributed computing context [22]–[28]. With coding, a job of k tasks is expanded into a job of n tasks with $n-k$ *parity* tasks. Parity tasks are constructed by encoding the initial k tasks, which is done by embedding redundancy either in the computational procedure collaboratively implemented by the tasks (e.g., [22]) or in the data the tasks consume during execution (e.g., [26]). If coded tasks are created with *MDS* code, the most commonly used encoding model, any k of the n tasks would be sufficient to recover the desired outcome of the job, thus only the fastest k tasks would be sufficient for completing the job.

Modeling task execution times and the variability they exhibit is crucial for the theoretical analysis of straggler mitigation techniques to match with the experimental measurements. In the analysis of straggler mitigation techniques, variability in execution times is commonly expressed with a fixed straggling factor. The straggling factor for each task is typically assumed to be independently drawn from a fixed random variable, which we also adopt in this paper. However, runtime variability is known to be to a large part caused by resource sharing in practice [7]–[9], and the redundant tasks added into system exert additional load on the system resources, which is expected to aggravate the runtime variability. Therefore, we believe that the model of variability should account for the redundancy added into system. In Sec. IV, we consider a model where the tail of task execution times changes with the level of redundancy added into system, and we study the cost and latency of redundancy under this model.

There are various decisions to make while employing redundancy for straggler mitigation. The first natural step is to decide adding whether replicated or coded tasks, and how many of them. Secondly, waiting for some time before launching the replica tasks has been considered to reduce the cost of redundancy [29]. A natural question is that does waiting before launching the redundant (replicated or coded) tasks help in general to reduce cost. In this paper, we analyze the cost vs. latency tradeoff to find out the best practice in making these decisions. As an alternative to adding redundancy, cancelling and relaunching the tasks that appear to be straggling after waiting some time has been considered [12]. This is justified by the heavy tailed nature of task execution times as observed in practice [7], [30], [31]. We quantify the effect of straggler relaunch on the cost vs. latency tradeoff in terms of the tail heaviness pronounced by the service time variability. We also consider employing straggler relaunch together with redundancy, and analyze its effects on cost and latency. Parts of the results presented in this paper were published in [1], [2].

This paper is structured as follows. In Sec. II, we explain the system model that is used for the presented analysis, and formally define the cost and latency of distributed job execution. In Sec. III, we examine the effect of the type and level of redundancy, and the launch time of redundant tasks on the cost vs. latency tradeoff. In Sec. IV, we evaluate the performance of job execution with redundancy when the redundant tasks added into system changes the tail of service time variability. In Sec. V, we study straggler relaunch and investigate its impact on the cost and latency. In Sec. VI, we consider employing straggler relaunch together with redundancy. In Sec. VII, we summarize our key findings, discuss the shortcomings of our analysis and possible future directions.

Summary of Observations: Coding allows increasing the level of added redundancy with finer steps than replication, which translates into greater achievable cost vs. latency region. Waiting for some time before launching the redundant tasks is not effective in trading off latency for reduced cost when the employed redundancy is coding, that is, one can obtain lower latency for the same cost by launching less number of coded tasks rather than delaying their launch time. When the employed redundancy is replication, some cost reduction is possible by launching the replica tasks after waiting some time. Coding is more efficient than replication in the cost vs. latency tradeoff; adding coded tasks into job execution yields higher reduction in latency per incurred cost (hence per incurred additional load on the system) compared to adding replicated tasks. Execution with redundancy reduces the cost and latency together when enough tail heaviness is pronounced by the service time variability. The required tail heaviness is smaller when coding is employed compared to replication. The advantage of coding over replication becomes greater when the job is executed at higher scale, i.e., when the job consists of greater number of parallel tasks.

Relaunching tasks that appear to be straggling after some time reduces the cost and latency when relaunching is performed at the right time and enough tail heaviness is pronounced by the service time variability. Redundancy and straggler relaunch serve the same purpose of mitigating stragglers, hence employing both together require greater tail heaviness in service time variability in order to reduce the cost and latency.

II. System Model

We adopt a system model that is an extension of what is adopted in [29]; execution time (duration from its launch time to completion) of each task is modeled with a single random variable. All k tasks of a job are launched simultaneously and the execution time of each is assumed to be identically and independently distributed (i.i.d.). We use two canonical distributions to model task execution times: 1) Shifted exponential $\mathrm{SExp}(s, \mu)$ with a positive minimum value s and a tail decaying exponentially at rate μ, and 2) $\mathrm{Pareto}(s, \alpha)$ with a positive minimum value s and a power law tail with index α. Minimum value of the distribution models the minimum service time of the tasks (i.e., task size), while tail of the distribution models the slowdown due to runtime variability; smaller μ or α implies greater chance for stragglers.

Task execution times in modern compute systems are known to exhibit heavy tail [7], [30], [31]. In Fig. 2, we plot the tail distribution of the task execution times[2] that we extracted from a Google Trace data for jobs with 15, 400, or 1050 tasks [30]. Note that both axes in the plots are in log scale, hence an exponential tail would have appeared as an exponentially decaying curve, while a true power law tail (e.g., tail of Pareto) would have pronounced a linear decay at a

[2]Task execution times are calculated as the difference between the timestamps for SCHEDULE and FINISH events for each task as given in [30].

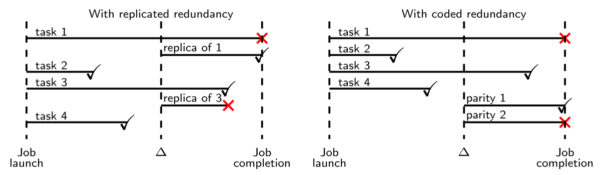


Fig. 1. A job of four tasks is executed by launching replicated (Left) or coded (Right) tasks, some time Δ after launching job's initial tasks. Check marks represent task completions while crosses represent cancellations. With replication, exact clones of the remaining tasks are launched, while with coding, parity tasks can be used as a "clone" for any task, therefore, stragglers do not have to be tracked down.

constant rate [32]. Tail distributions shown in the figure exhibit exponential decay at small values and a linearly decaying trend at larger values, which indicates a heavy tailed runtime variability [33]. Note that the steep decay of the tail at the far right edge is due to the bounded support of the distributions.

Assuming execution times to be identically distributed for tasks within the same job is appropriate since jobs in practice are known to be a collection of one or more usually identical tasks [30], [34]. However, when task execution times are modeled using a distribution with a minimum value of zero (e.g., exponential distribution), assuming independent execution times across the initial and redundant tasks proved to be problematic because added redundancy in this case can make job execution time arbitrarily small. This is in contrast to reality where tasks have an inherent size and due to this, job execution times are lower bounded by a positive value regardless of the level of added redundancy. Modeling task execution times with a minimum value of zero have previously led to theoretical results that are at odds with experimental measurements. Implications of this are discussed in detail in [17] and authors propose a better model for service times in which the time due to task size is decoupled from the time due to runtime variability. Specifically, service times are modeled as $s \times Sl$ where s represents the inherent task size and Sl is the slowdown factor, which is assumed to be i.i.d. across tasks and servers with a minimum value of 1. Distributions that we adopt for modeling task execution times can be expressed using the decoupling method introduced in [17]; $\mathrm{SExp}(s, \mu)$ can be written as $s \times \mathrm{SExp}(1, \mu)$ or $\mathrm{Pareto}(s, \alpha)$ as $s \times \mathrm{Pareto}(1, \alpha)$.

In our model, redundant tasks are added into execution only if the job does not complete within some time Δ. Redundancy is introduced either in the form of task replicas or coded parity tasks. When replication is employed, c replicas are launched for every remaining task at time Δ. When coding is employed, $n - k$ MDS coded parity tasks are launched at time Δ (see Fig. 1). When straggler relaunch is implemented, tasks (initial or redundant) remaining at time Δ are canceled and fresh replacements are immediately launched in their place.

We define the cost of executing a job as the sum of the lifetimes of all the tasks (including the redundant ones) involved in its execution. Lifetime of a task is the duration from its launch to its completion or cancellation. Depending on the application domain, there are two possible cost definitions: 1) *Cost with task cancellation*; outstanding redundant tasks are canceled as soon as the job completes (as illustrated in Fig. 1), which is a viable option for distributed job execution, 2) *Cost without task cancellation*; outstanding redundant tasks are left to run until they complete, which for instance is the only option for routing messages with redundancy in an opportunistic network [35]. We assume that task cancellation takes place instantly and does not incur any delay. In the following subsection, we elaborate on the meaning and consequences of the job execution cost.

A. On the Cost of Job Execution

Cost, as is defined here, reflects the total resource time spent while executing a job. Lower cost translates into executing the same job by occupying less area in system capacity $\times$ time space. Thus, reducing the cost of job executions allows fitting more jobs per area and leads to higher system throughput [36].

As we show in the following sections, adding redundant tasks into a job execution can increase or *decrease* the cost depending on the variability pronounced in task execution times, and the type and level of introduced redundancy. Since redundancy can lead to higher cost, it should be employed with care. Executing jobs at a higher cost implies occupying greater portion of system's overall capacity per job, which increases the load on the system. This may translate into greater congestion in the system resources, hence aggravate job slowdowns or even drive the system to instability. For instance, [17] shows, with a queueing theoretic analysis, how excessive replication of single-task job arrivals can drive system to instability. As another example, [14] introduces a system, named as *Dolly*, which launches replicas only for small jobs that consist of ≤ 10 tasks. This is shown to achieve significant reduction in latency without overburdening the system according to the traces collected on two clusters at Facebook and Microsoft Bing. The underlying reason for the success of their replication scheme is the workload characteristics; small jobs were observed to tend to have short duration, thus, replicating them did not introduce substantial cost overhead in the system, while returning substantial reduction in the latency of short jobs.

The workload and system characteristics considered in [14] are not universal; execution with redundancy is relevant in general not only for small jobs but also for jobs that run at higher scale for large duration. Job slowdowns due to stragglers is an emerging problem for future high performance computing (HPC) systems . Exascale computing is expected to be implemented by systems that are much larger in size and will enable execution at unprecedented levels of parallelism. These future systems are anticipated to be prone to much higher node level runtime variability [37]. Moreover, to implement high resource utilization, resource scheduling in these systems is suggested to be realized with time-sharing rather than today's de facto batch scheduling [38]. As resource sharing is pointed out as the primary cause of stragglers in data centers [7], performance of future HPC systems is likely to greatly suffer from stragglers. Simulations over the traces collected on Edison Supercomputer demonstrate that jobs with larger number of tasks and shorter duration experience higher slowdowns due to runtime variability under batch scheduling, while under time-sharing based resource scheduling, slowdowns are observed to be relatively uniform regardless of the number of tasks or the job duration [38].

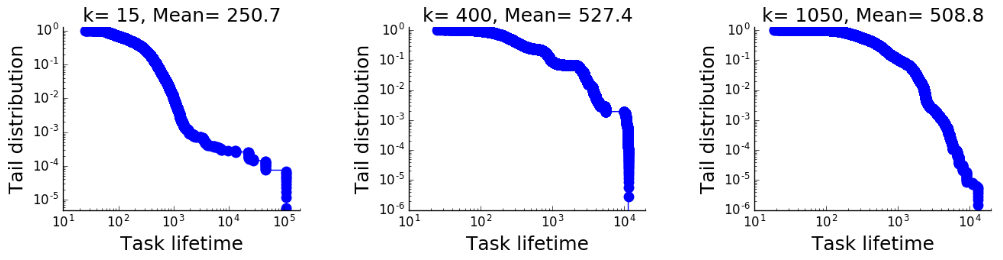


Fig. 2. Empirical tail distribution of task execution times for Google cluster jobs with number of tasks $k = 15, 400, 1050$.

B. Notation and Tools for Analysis

Expected cost (C) and latency (T) are the two metrics that we use to quantify the pain and gain of distributed execution of jobs with redundancy and/or straggler relaunch. Thus, the cost and latency by themselves imply their expected values throughout, and any other quantity associated with them is made explicit. Note that the cost and latency depend on the number of tasks k that constitute the job, task sizes s, runtime variability (determined by μ or α), the level of redundancy added into the job (c task replicas or $n-k$ coded tasks), as well as the time Δ at which redundant tasks are launched and/or straggler relaunch is performed.

Derivations of the cost and latency expressions make frequent use of the law of total probability since we consider adding redundancy and/or performing straggler relaunch after waiting some time Δ. Results from order statistics are essential for the derivations since only a subset of the launched tasks is necessary for job completion when redundancy is employed. Derivations presented in the paper require tedious algebra at times and the expressions involve some special functions that commonly appear while working with order statistics. For completeness, we kept every non-trivial step in the proofs. This made some proofs lengthy and we placed them in the Appendix that is made available as a supplement to this paper.

We here give an overview of the notation and special functions that appear throughout the paper. For their detailed definitions and interesting properties, we refer the reader to [39]. $X_{n:i}$ denotes the ith order statistic of n i.i.d. samples drawn from a random variable X. H_n, the nth harmonic number, is defined as $\sum_{i=1}^{n} 1/i$ for $n \in \mathbb{Z}^+$ or as $\int_0^1 (1-x^n)/(1-x)\mathrm{d}x$ for $n \in \mathbb{R}$. H_{n^2}, the nth generalized harmonic number of order two, is defined as $\sum_{i=1}^{n} 1/i^2$. Incomplete Beta function $B(q; m, n)$ is defined for $q \in [0,1]$, $m, n \in \mathbb{R}^+$ as $\int_0^q u^{m-1}(1-u)^{n-1}\mathrm{d}u$, Beta function $B(m,n)$ as $B(1;m,n)$ and its regularized form $I(q;m,n)$ as $B(q;m,n)/B(m,n)$. Gamma function $\Gamma(x)$ is defined as $\int_0^\infty u^{x-1}e^{-u}\mathrm{d}u$ for $x \in \mathbb{R}$ or as $(x-1)!$ for $x \in \mathbb{Z}^+$.

III. Coding vs. Replication

In this section, we study the cost vs. latency tradeoff in executing a distributed job by adding task replicas or coded tasks after waiting some time Δ. Note that we do not consider straggler relaunch until Sec. V. Theorems given below firstly present expressions for the cost and latency assuming exponential task execution times, which we then use to derive the cost and latency for shifted-exponential task execution times.

Consider Executing a Job of k Tasks by Adding c Replicas for Each Remaining Task After Waiting Some Time Δ:

Theorem 1: Suppose task execution times are i.i.d. with $\mathrm{Exp}(\mu)$. *Distribution of job execution time is given as*

$$\Pr\{T \le t\} = \Big(1 - \mathbb{1}(t \le \Delta)(e^{-\mu t} - e^{-\mu\Delta}) - \mathbb{1}(t > \Delta)e^{-\mu((c+1)(t-\Delta)+\Delta)}\Big)^k \quad (1)$$

Latency is well approximated as

$$\mathbb{E}[T] \approx \frac{1}{\mu}\left(H_k - \frac{c}{c+1}H_{k-kq}\right). \quad (2)$$

Cost with (C^c) *or without* (C) *task cancellation is given as*

$$\mathbb{E}[C^c] = \frac{k}{\mu}, \qquad \mathbb{E}[C] = (c(1-q)+1)\frac{k}{\mu}. \quad (3)$$

where $q = 1 - e^{-\mu\Delta}$.

Theorem 2: Suppose task execution times are i.i.d. with $\mathrm{SExp}(s,\mu)$. *Distribution and the expected value of job execution time are given as*

$$\Pr\{T > t\} = \Pr\{T_e > t - s\}, \qquad \mathbb{E}[T] = s + \mathbb{E}[T_e]. \quad (4)$$

where T_e is the job execution time when task execution times are distributed as $\mathrm{Exp}(\mu)$, *for which the distribution and expected value are given in Thm. 1.*

Cost with task cancellation is given as

$$\mathbb{E}[C^c] = \begin{cases} k(c+1)\left(s + \dfrac{1}{\mu} \times\left(1 - \dfrac{c}{c+1}(e^{-\mu\Delta} + \mu\Delta)\right)\right) & \Delta \le s, \\ k\left(s + \dfrac{1}{\mu}\left(1 + c\left(1 - q - e^{-\mu\Delta}\right)\right)\right) & o.w. \end{cases} \quad (5)$$

Cost without task cancellation is given as

$$\mathbb{E}[C] = k\left(c(1-q)+1\right)(s + 1/\mu). \quad (6)$$

where $q = \mathbb{1}(\Delta > s)\left(1 - e^{-\mu(\Delta - s)}\right)$.

Consider Executing a Job of k Tasks by Adding $n-k$ Coded Tasks After Waiting Some Time Δ:

Theorem 3: Suppose task execution times are i.i.d. with $\mathrm{Exp}(\mu)$. *Distribution and the expected value of job execution time are well approximated as*

$$\Pr\{T > t\} \approx \mathbb{1}(t \le \Delta)\left(q^k - (1 - e^{-\mu t})^k\right) + I\left(\mathbb{1}(t > \Delta)e^{-\mu(t-\Delta)}; n-k+1, k(1-q)\right) - q^k I\left(\mathbb{1}(t > \Delta)e^{-\mu(t-\Delta)}; n-k+1, 0\right),$$

$$\mathbb{E}[T] \approx \Delta - \frac{1}{\mu}\left(B(q; k+1, 0) + H_{n-kq} - H_{n-k}\right). \quad (7)$$

Cost with (C^c) or without (C) task cancellation is given as

$$\mathbb{E}[C^c] = \frac{k}{\mu}, \qquad \mathbb{E}[C] = \frac{k}{\mu}q^k + \frac{n}{\mu}\left(1 - q^k\right), \quad (8)$$

where $q = 1 - e^{-\mu\Delta}$.

Theorem 4: Suppose task execution times are i.i.d. with $\mathrm{SExp}(s, \mu)$. *Distribution and the expected value of job execution time are given as*

$$\begin{aligned}\Pr\{T > t\} &= \Pr\{T_e > t - s\},\\ \mathbb{E}[T] &= s + \mathbb{E}[T_e],\end{aligned} \quad (9)$$

where T_e *is the job execution time when task execution times are distributed as* $\mathrm{Exp}(\mu)$, *for which the distribution and the expected value are given in Thm. 3.*

Cost with (C^c) or without (C) task cancellation is given as

$$\mathbb{E}[C] = \begin{cases} n\,(s + 1/\mu) & \Delta \le s,\\ \left(k + (1 - \tilde{q}^k)(n-k)\right)(s + 1/\mu) & o.w.\end{cases}$$

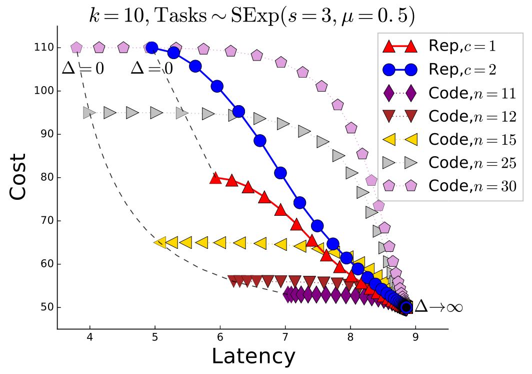
$$\mathbb{E}[C^c] = \begin{cases} k/\mu + \ ns - (n-k)q^k \\ \quad \times \left(\Delta + k\mu\left(\frac{\zeta}{\mu q^k} - \Delta\left(\frac{1}{q} - 1\right)\right)\right) & \Delta \le s,\\ (\approx)\ \mathbb{E}[C] - \frac{n-k}{\mu}\Big(1 - q^k + \zeta^{-k(1-q)} \\ \quad \times B(\zeta; k - kq + 1, 0)\left(\tilde{q}^k - q^k\right)\Big) & o.w.\end{cases}$$

where $q = \mathbb{1}(\Delta > s)\left(1 - e^{-\mu(\Delta - s)}\right)$, $\tilde{q} = 1 - e^{-\mu\Delta}$ *and* $\zeta = 1 - e^{-\mu s}$.

In distributed computing systems, outstanding redundant tasks can be canceled by signaling the computing nodes as soon as the job completes, hence we always refer to the cost with task cancellation in the discussions throughout the paper.

When task execution times are exponentially distributed, expressions in Thm. 1 and 3 tell us that job execution cost neither depends on the time Δ at which redundant tasks are launched nor the level of employed replicated (c) or coded (n) redundancy. Therefore, according to our model with exponentially distributed task execution times, launching all the available redundant tasks at the beginning (i.e., $\Delta = 0$) achieves the minimum latency with zero penalty in cost.

Recent research proposes waiting for some time before replicating the tasks to reduce the cost of redundancy [29]. Using the expressions given in Thm. 2 and 4, Fig. 3 plots the cost vs. latency tradeoff in executing the same job with different levels of replicated or coded redundancy, by varying the launch time Δ of the redundant tasks between 0 and ∞. First, let us focus on the case with SExp task execution times as shown in Fig. 3 (Top). Cost monotonically decreases while latency monotonically increases with Δ. Let $C(c, \Delta)$, $T(c, \Delta)$ be the cost and latency when c replicas are added for each remaining task after waiting some time Δ. Increasing Δ initially allows significant reduction in cost while causing a slight increase in latency. However, as soon as $T(c, \Delta)$ exceeds $T(c-1, 0)$ (plot shows this for $c = 2$) increasing Δ further does not make sense; one can achieve less cost for the same latency by reducing c rather than increasing Δ.



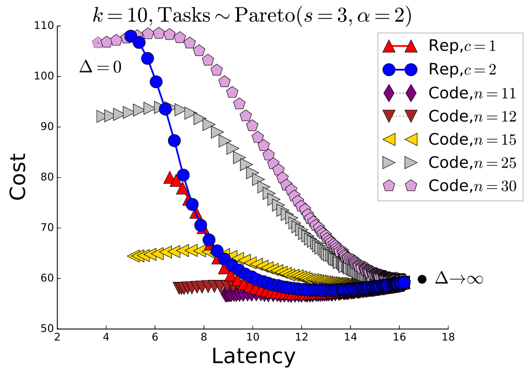


Fig. 3. Achievable cost (with task cancellation) vs. latency in executing a job of k tasks with replicated ($c = 1, 2$) or coded ($n \in [k+1, 3k]$) redundancy. Each cost vs. latency curve is drawn for a fixed number of redundant tasks by varying the launch time Δ of redundant tasks. Task execution times are i.i.d. with SExp (Top) and Pareto (Bottom).

This behavior of cost vs. latency tradeoff is more apparent when coded redundancy is employed. Increasing Δ from zero initially returns no visible cost reduction while incurring significant increase in the latency, while it does yield visible reduction in cost only after Δ reaches a certain value. In other words, adding coded tasks with delay can yield significant cost reduction only after significant sacrifice in latency. Consider the cost and latency value at a sufficiently large value of Δ on a curve for a number of coded tasks $n - k = r > 1$, then the curve below for $n - k = r - 1$ attains the same latency at less cost at a smaller value of Δ. Thus, given a job execution with a sufficiently large value of Δ and $r > 1$ coded tasks, same latency can be achieved for less cost by reducing Δ and decrementing r. The same conclusions hold for the case with Pareto task execution times (shown in Fig. 3 (Bottom)).

The remainder of this section is concerned with the cost vs. latency tradeoff when redundant tasks are launched together with the original tasks (i.e., $\Delta = 0$), which we refer to as *zero-delay redundancy*. For the case with $\Delta > 0$, we could derive the cost and latency expressions only when the distribution of task execution times, which we refer to as X here, has exponential tail. This is because in the absence of memoryless property (e.g., when X is heavy tailed), derivations require working with the order statistics of samples drawn from two different distributions[3]; residual execution time of the

[3]This issue disappears when the remaining tasks at time Δ are relaunched. This is why in Sec. V, we will be able to derive the cost and latency expressions for the case of jointly performing straggler relaunch and launching redundant tasks after waiting some time Δ.

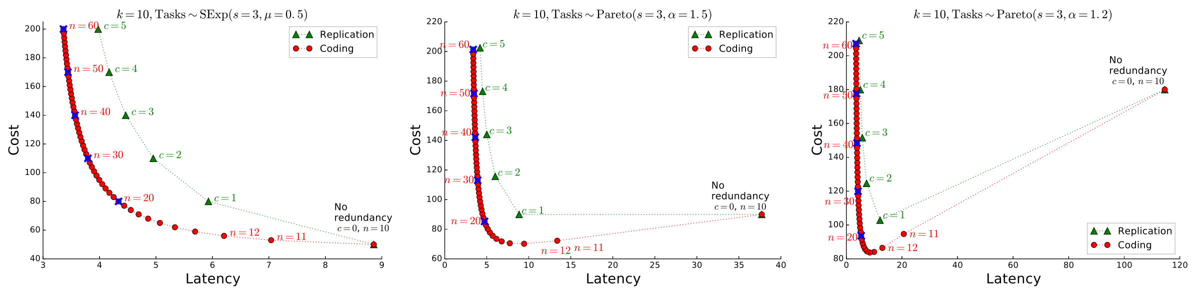


Fig. 4. Cost vs. latency of executing a job of $k = 10$ tasks by employing zero-delay replicated or coded redundancy. The level of employed redundancy, c for replication and n for coding, varies along each curve. Tail heaviness of task execution times increases from left to right.

remaining task copies after time Δ is distributed as $X|X > \Delta$, while the execution time of copies that are newly launched at time Δ is distributed as X. Order statistics of independent but non-identical random variables has been studied in the literature [40]. Using the results available in the literature, cost and latency expressions in the case with non-exponential X could be written out, however, the expressions are unwieldy (relatively bearable for job execution with replicas compared to execution with coded tasks). We did not pursue deriving such cumbersome expressions because our purpose was to observe the effect of Δ on the cost vs. latency tradeoff, which was very well served by the expressions we derived for the case with shifted-exponential X. When $\Delta = 0$, cost and latency expressions can be derived with fairly tractable steps when X has either exponential or heavy tail.

Theorem 5: Let the cost (with task cancellation) and latency of executing a job of k tasks be C_c, T_c when each task is launched together with c replicas, and let them be C_n, T_n when job is launced with $n-k$ additional coded tasks. When task execution times are i.i.d. with $\mathrm{SExp}(s, \mu)$,

$$\mathbb{E}[T_c] = s + \frac{H_k}{(c+1)\mu}, \quad \mathbb{E}[C_c] = k\left((c+1)s + \frac{1}{\mu}\right),$$
$$\mathbb{E}[T_n] = s + \frac{1}{\mu}(H_n - H_{n-k}), \quad \mathbb{E}[C_n] = ns + \frac{k}{\mu}.$$

When task execution times are i.i.d. with $\mathrm{Pareto}(s, \alpha)$,

$$\mathbb{E}[T_c] = sk!\frac{\Gamma\left(1 - 1/\left((c+1)\alpha\right)\right)}{\Gamma\left(k+1-1/\left((c+1)\alpha\right)\right)},$$
$$\mathbb{E}[C_c] = sk(c+1)\frac{\alpha}{\alpha - 1/(c+1)},$$
$$\mathbb{E}[T_n] = s\frac{n!}{(n-k)!}\frac{\Gamma(n-k+1-1/\alpha)}{\Gamma(n+1-1/\alpha)},$$
$$\mathbb{E}[C_n] = s\frac{n}{\alpha-1}\left(\alpha - \frac{\Gamma(n)}{\Gamma(n-k)}\frac{\Gamma(n-k+1-1/\alpha)}{\Gamma(n+1-1/\alpha)}\right).$$

Using the expressions given in Thm. 5, Fig. 4 plots the cost vs. latency tradeoff in executing the same job by introducing varying levels of zero-delay replicated or coded redundancy. Under both SExp and Pareto task execution times, coding always achieves less latency for the same cost compared to replication. This observation is formally stated in Thm 6.

Theorem 6: Consider launching a job of k tasks with redundant tasks. Cost and latency is lower when kc MDS coded tasks are added compared to adding c replicas for each task.

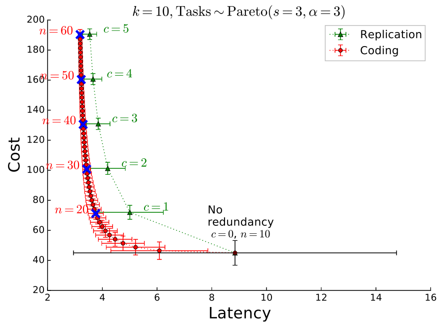


Fig. 5. Cost vs. latency for zero-delay redundancy systems. The width of horizontal error bars is equal to the standard deviation of latency and the width of vertical bars is equal to the standard deviation of cost.

When task execution times are light tailed, adding redundant tasks into the job reduces its latency but increases its cost. In [29], replication is demonstrated to reduce the cost and latency together when task execution times are heavy tailed. Using the exact expressions in Thm. 5, Fig. 4 illustrates that redundancy can reduce cost and latency together when the tail of task execution times is *heavy enough*. Reduction in the cost and latency is greater with coding compared to replication. We elaborate at the end of this section on the tail heaviness required to achieve reduction in the cost and latency together.

Although closed form expressions are formidable to derive, second moments of the cost and latency can be exactly computed as described in Thm. 7, which enables us to compute the standard deviation of the cost and latency. Fig. 5 plots the expected cost and latency values with error bars of width equal to the standard deviation in respective dimensions. Variability in the cost and latency naturally decreases with increasing levels of redundancy. Fixing the number of added redundant tasks, coding achieves less variability compared to replication.

Theorem 7: Consider launching a job of k tasks with redundant tasks. Let us denote the cost and latency as C_c, T_c when c replicas are added for each task, and as C_n, T_n when $n-k$ coded tasks are added. For $X \sim$ $\mathrm{Exp}(\mu)$ *and $j \geq i$, we have*

$$\mathbb{E}[X_{n:i}X_{n:j}] = \frac{1}{\mu^2}\Big(H_{n^2} - H_{(n-i)^2} + (H_n - H_{n-i})(H_n - H_{n-j})\Big).$$

as given in [41, Pg. 73]. Let $Y \sim$ $\mathrm{Exp}((c+1)\mu)$. *When task execution times are i.i.d. with* $\mathrm{SExp}(s, \mu)$, *second moments of* *the cost and latency are given as*

$$\begin{aligned}
\mathbb{E}[T_c^2] &= \left(s + \frac{H_k}{(c+1)\mu}\right)^2 + \frac{H_{k^2}}{(c+1)^2\mu^2},\\
\mathbb{E}[C_c^2] &= (k(c+1)s)^2 + 2k(c+1)s\frac{k}{\mu}\\
&\quad + (c+1)^2\sum_{i,j=1}^{k}\mathbb{E}[Y_{n:i}Y_{n:j}]\\
\mathbb{E}[T_n^2] &= \frac{H_{n^2} - H_{(n-k)^2}}{\mu^2} + \left(s + \frac{H_n - H_{n-k}}{\mu}\right)^2,\\
\mathbb{E}[C_n^2] &= (ns)^2 + 2ns\frac{k}{\mu} + (n-k)^2\mathbb{E}[X_{n:k}^2]\\
&\quad + 2(n-k)\sum_{i=1}^{k}\mathbb{E}[X_{n:i}X_{n:k}] + \sum_{i,j=1}^{k}\mathbb{E}[X_{n:i}X_{n:j}].
\end{aligned}$$

For $X \sim \text{Pareto}(s,\alpha)$*, given* $\alpha > \max\{2/(n-i+1), 1/(n-j+1)\}$ *and* $j \geq i$*, we have*

$$\begin{aligned}
&\mathbb{E}[X_{n:i}X_{n:j}]\\
&= s^2\frac{n!}{\Gamma(n+1-2/\alpha)}\\
&\quad\times\frac{\Gamma(n-i+1-2/\alpha)}{\Gamma(n-i+1-1/\alpha)}\frac{\Gamma(n-j+1-2/\alpha)}{\Gamma(n-j+1)}.
\end{aligned}$$

as given in [42, Pg. 62]. Let $Y \sim \text{Pareto}(s,(c+1)\alpha)$*. When task execution times are distributed as* $\text{Pareto}(s,\alpha)$*, second moments of the cost and latency are given as*

$$\begin{aligned}
\mathbb{E}[T_c^2] &= \mathbb{E}[Y_{k:k}^2],\\
\mathbb{E}[C_c^2] &= (c+1)^2\sum_{i,j=1}^{k}\mathbb{E}[Y_{k:i}Y_{k:j}],\\
\mathbb{E}[T_n^2] &= \mathbb{E}[X_{n:k}^2]\\
\mathbb{E}[C_n^2] &= (n-k)^2\mathbb{E}[X_{n:k}^2] + 2(n-k)\sum_{i=1}^{k}\mathbb{E}[X_{n:i}X_{n:k}]\\
&\quad + \sum_{i,j=1}^{k}\mathbb{E}[X_{n:i}X_{n:j}].
\end{aligned}$$

Proof Sketch: Derivations follow from the cost and latency formulation given in the proof of Thm. 5. □

When task execution times are heavy tailed, it is possible to reduce latency by adding redundant tasks and still pay for the baseline cost of executing the job with no redundancy (cf. Fig. 4). We refer to this as *latency reduction at no cost.*

Corollary 1: Suppose task execution times are i.i.d. with $\text{Pareto}(s,\alpha)$*. Launching a job of* k *tasks by adding* c *replicas for each task can reduce its latency up to a minimum value* $\mathbb{E}[T_{\min}]$ *without incurring any additional cost if and only if* $\alpha < 1.5$*, and for* $c_{\max} = \max\{\lfloor 1/(\alpha-1)\rfloor - 1, 0\}$*, we have*

$$\mathbb{E}[T_{\min}] = sk!\frac{\Gamma\left(1 - 1/\left(\alpha(c_{\max}+1)\right)\right)}{\Gamma\left(k+1-1/\left(\alpha(c_{\max}+1)\right)\right)}. \tag{10}$$

A sufficient condition to reduce latency with no additional cost by adding $n-k$ *coded tasks is given as*

$$\alpha^\alpha \leq \frac{n}{n-k+1}, \tag{11}$$

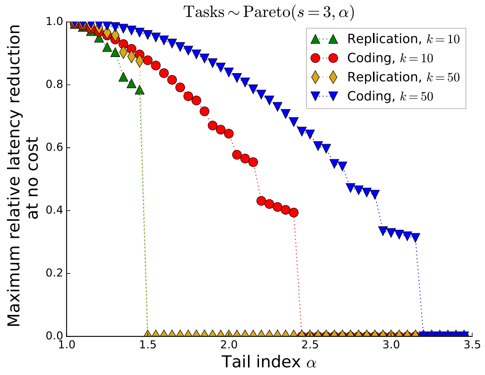


Fig. 6. Maximum relative latency reduction at no cost by employing replicated or coded redundancy vs. the tail of task execution times.

a necessary condition is given as

$$\alpha^\alpha \leq \frac{n+1}{n-k}, \tag{12}$$

the minimum latency at no additional cost is given as

$$\mathbb{E}[T_{\min}] = f(n_{\max}). \tag{13}$$

such that

$$\begin{aligned}
f(n) &= s\frac{n!}{(n-k)!}\frac{\Gamma(n-k+1-1/\alpha)}{\Gamma(n+1-1/\alpha)},\\
n_{\max} &= \max\left\{ n \;\middle|\; f(n) - \frac{f(k)}{(n-k)} - \alpha \leq 0 \right\},
\end{aligned}$$

or it is bounded as follows

$$\mathbb{E}[T_{\min}] < s\left(\alpha + k!\frac{\Gamma(1-1/\alpha)}{\Gamma(k+1-1/\alpha)}\right). \tag{14}$$

Fig. 6 plots the maximum relative latency reduction at no cost in executing the same job under varying degree of tail heaviness in task execution times. Maximum relative latency reduction at no cost is defined as $\left(\mathbb{E}[T_0] - \mathbb{E}[T_{\min}]\right)/\mathbb{E}[T_0]$ where $\mathbb{E}[T_{\min}]$ is the minimum possible latency at no cost, and $\mathbb{E}[T_0]$ is the baseline latency of executing the job with no redundancy. As stated in Cor. 1, when the employed redundancy is replication, latency reduction at no cost is possible only when the tail index of task execution times is less than 1.5, that is, only when the tail of task execution times is quite heavy. Employing coded redundancy relaxes this requirement on the tail heaviness, as also shown in the plot. When the employed redundancy is replication, the tail heaviness requirement is independent of the number of tasks k that constitute the job, while employing coded redundancy relaxes the requirement on the tail index further at larger k, i.e., the upper threshold on the tail index increases with k. This can be explained as follows. A task replica can only replace its original copy, while a coded task can replace any of the k initial tasks. Thus, coded tasks can mitigate stragglers more effectively when the job is executed at higher scale (larger k), while the effectiveness of task replicas is not associated with the scale of execution. Consequently for jobs that run at higher scale, coding can reduce latency at no cost even under lighter tailed task execution times, while the scale of execution does not change the tail heaviness requirement for replication.

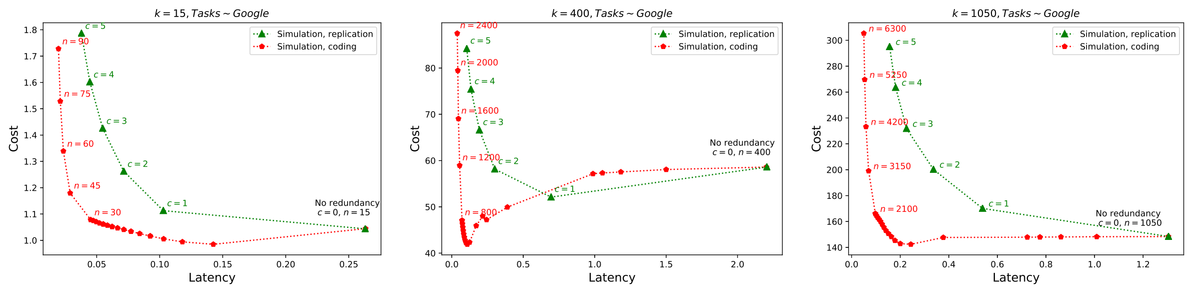


Fig. 7. Simulated cost vs. latency curves for executing jobs with $k = 15, 400, 1050$ tasks by employing zero-delay replicated or coded redundancy. Task execution time distributions used in the simulations are extracted from a Google Cluster Trace data [30].

Demonstration Using Google Cluster Data: We simulated job executions with replicated or coded redundancy by using task execution time distributions extracted from a Google Cluster data [34]. Google released this data from a cluster running a mixed workload of short or long running MapReduce batch jobs, services and interactive queries [30].

Fig. 7 plots the cost vs. latency curves using the three empirical distributions for jobs with $k = 15, 400, 1050$ tasks that were previously illustrated in Fig. 2. In all three, coding is doing better than replication in the cost vs. latency tradeoff. Execution with redundancy could reduce the cost and latency together because each of these distributions pronounces heavy tail at large values (cf. Fig. 2). In the execution of jobs with 15 or 1050 tasks, employing replication does not allow for latency reduction at no cost but coding does. In the execution of job with 400 tasks, although replication seems to achieve less cost and latency at first, coding outperforms replication beyond a certain level of redundancy.

IV. When Redundancy Changes the Tail

So far we have ignored the impact of redundancy on the system. Redundant tasks exert extra load on the system, which is likely to aggravate the existing contention in the system resources. Given that resource contention is the primary cause of runtime variability [7], the added redundant tasks are likely to increase the variability in task execution times.

Compute servers are typically shared by the tasks of jobs that simultaneously execute on the cluster [18]. Two canonical server sharing strategies are 1) Processor sharing: tasks time-share the server according to a round-robin scheduling, 2) Queueing: tasks wait in a queue and are accepted into service one at a time. Modern Operating Systems implement a mix of processor sharing and First-come First-served (FCFS) queueing to host multiple processes on a server, e.g., scheduling classes SCHED_FIFO and SCHED_RR in the Linux Kernel [43]. A compute server in reality hosts several shared resources (e.g., CPU, memory, I/O bus, etc.) and each with its own scheduling scheme. For simplicity, we here model servers to host only CPU. We adopt *limited processor sharing* model in which tasks are allowed to time-share the server (while being served over multiple CPU cores or threads) until a limited number of them accumulate, beyond which the remaining tasks wait in a FCFS queue. Limited processor sharing is shown to implement robust performance (in terms of the tail of response time) for both heavy and light tailed task sizes [44].

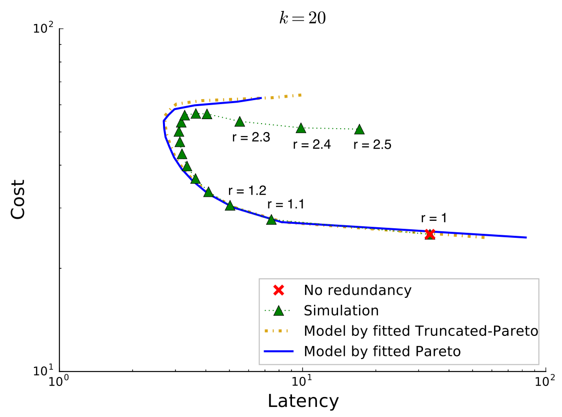


Fig. 8. Cost vs. latency for a particular type of job with 20 tasks of unit size. Arriving jobs are expanded with coded tasks at a multiplicative factor of r. Simulated values are given for increasing values of r; we start with $r = 1$ (No redundancy) and then increase r by 0.1 at each step.

In order to understand the impact of added redundancy on the system's runtime variability, we simulated a cluster of servers, each implementing a limited processor sharing queue. Jobs of varying number of tasks and size (minimum task execution time) arrive to cluster according to a Poisson process. Distribution of task sizes and number of tasks within real compute jobs are known to exhibit heavy tail [30], [45]–[47]. Therefore in our simulation: i) Task size for each arriving job is independently sampled from a Truncated-Pareto (a canonical continuous heavy tailed) distribution with minimum value of 1, maximum value of 10^{10} and tail index of 1.1. The choice of Truncated-Pareto distribution and the values for its parameters come from the distribution of real compute task sizes presented in [48]. ii) Number of tasks that constitute each arriving job is independently sampled from a Zipf (a canonical discrete heavy tailed) distribution. Each arriving job is expanded with the same rate $r > 1$; a job of k tasks gets expanded into $n = \lfloor rk \rfloor$ tasks by adding $\lfloor rk \rfloor - k$ coded tasks, and the resulting n tasks are dispatched to the n servers with the least number of tasks in the cluster. As soon as any k of the n tasks of a job is completed, the job completes and its remaining $n - k$ outstanding tasks (either in service or waiting in a queue) get immediately removed from the cluster. Expanding jobs with the same rate r ensures fairness by introducing redundancy in proportion to the scale k at which a job is executed.

Cost and latency values for a particular type of job with a fixed number of tasks of unit size are plotted in Fig. 8 for increasing values of r. Each simulated server in the cluster implements limited processor sharing queue with a limit of 8 tasks. The latency of the job is the time span between its arrival and departure to and from the system. The cost of the job is the sum of the service time of every task involved in its execution. Simulated curve shows that redundancy initially reduces latency significantly with little change in cost, then reduces latency but increases cost, and finally beyond a level increases latency with little change in cost. In order to evaluate the appropriateness of modeling task execution times with canonical heavy tailed distributions, we first fitted Pareto and Truncated-Pareto distributions on the task execution times sampled from the simulation, then substituted these fitted models in the analytical cost and latency expressions. We presented cost and latency expressions for Pareto task execution times in Thm. 5. Cost and latency are formidable to derive in closed form for Truncated-Pareto task execution times, but their computation involves a single integral which we evaluate numerically (refer to [42, Pg. 63]). Parameters of the Pareto (minimum value and tail index) and Truncated-Pareto (minimum and maximum values, and tail index) models are estimated using the unbiased MLE estimators that are respectively presented in [49] and [50, Thm. 1].

The comparison given in Fig. 8 between the simulated and fitted values of cost and latency shows that modeling task execution times with Pareto distribution is fairly appropriate to study the cost vs. latency tradeoff. This is not surprising since the asymptotic approximations of the tail of waiting times in FCFS or processor sharing queues have demonstrated that heavy tailed task sizes result in heavy tailed delay [51]–[53]. However one caveat of the model is that it cannot capture the case we observe in (the top right of) Fig. 8 in which adding more redundancy increases latency with little change in cost. In the remainder of this section, we study the cost vs. latency tradeoff by adopting a Pareto task execution time model that is dependent on the rate r at which redundancy is added into all the jobs executing in the system. Expansion of a job with task replicas at (an integer) rate of r refers to launching $r-1$ replicas for each of the k tasks within the job.

Redundant tasks added into the system are expected to aggravate resource contention, and consequently increase the variability in task execution times. Therefore, the impact of redundant load exerted on the system should be incorporated in the $\mathrm{Pareto}(s, \alpha)$ distribution that we use to model task execution times. Under stability, an arriving job, with nonzero probability, can find the system empty and complete execution without having to share servers with any other job. Thus, we assume that minimum task execution time s solely reflects the task size and is not affected by resource contention. Then, the impact of added redundant load should be captured by the only remaining parameter, the tail index α. Smaller α implies greater variability (implying greater chance and impact of resource contention), so α is expected to get smaller as more redundancy is added into the jobs, which is indeed what we observe in the simulations. We directly use the job expansion rate r to quantify the level of added redundancy and model α as a function of r. Note that we do not study the exact trend which describes how α changes with r, but rather try to understand the requirements on the relationship between α and r that leads to gain or pain in the cost vs. latency tradeoff.

We firstly present sufficient conditions in terms of α and r to yield a reduction or incur an increase in latency.

Theorem 8: Suppose that task execution times are i.i.d. with Pareto *with tail index α_i when jobs arriving to the system are expanded with redundant tasks by a multiplicative factor of $r_i > 1$. Consider increasing r_i to r_j. If jobs are expanded with coded tasks, a sufficient condition to reduce the latency of a job of k tasks by the change $r_i \to r_j$ is*

$$\alpha_i/\alpha_j \leq \log\left(\frac{n_i}{n_i-k+1}\right) / \log\left(\frac{n_j+1}{n_j-k}\right), \tag{15}$$

a sufficient condition to incur an increase in job's latency is

$$\alpha_i/\alpha_j \geq \log\left(\frac{n_i+1}{n_i-k}\right) / \log\left(\frac{n_j}{n_j-k+1}\right), \tag{16}$$

where $n_i = \lfloor kr_i \rfloor$ and $n_j = \lfloor kr_j \rfloor$. If jobs are expanded with task replicas, a necessary and sufficient condition for the change $r_i \to r_j$ to reduce latency is given for any job as

$$\alpha_i/\alpha_j < r_j/r_i. \tag{17}$$

Condition (15) for the case of expanding jobs with coded tasks is sufficient to reduce latency, but it may not give tight guarantees. It can be made easier to interpret by expressing the expansion rate r as n/k for a given job of k tasks. Increasing the rate from n/k to $(n+1)/k$, the condition (15) becomes

$$\frac{\alpha_n}{\alpha_{n+1}} \leq \log\left(\frac{n}{n-k+1}\right) / \log\left(\frac{n+2}{n-k+1}\right) < 1.$$

This says that if the tail heaviness of task execution times (or α) stays the same or becomes lighter as r increases, increasing r reduces latency for all jobs regardless of k. This is not informative since we already know that latency monotonically decreases in n when the tail heaviness of task execution times stays the same let alone when it gets lighter (cf. Thm. 5).

Next we derive an *approximate* necessary and sufficient condition to reduce latency of a particular job by increasing r, in the case where jobs are expanded with coded tasks. Presented approximation yields close estimates for large enough values of r, in particular when $r > 2$. Approximating the quotient of Gamma functions with Sterling's approximation [54], latency of executing a job of k tasks in a system with coded expansion rate $r = n/k$ is approximately given as

$$\mathbb{E}[T_n] \approx s\left(1 + \frac{k}{n-k+1}\right)^{1/\alpha_n},$$

which gives us the following approximation for the ratio

$$\frac{\mathbb{E}[T_{n+1}]}{\mathbb{E}[T_n]} \approx \left(1+\frac{k}{n-k+2}\right)^{1/\alpha_{n+1}} \left(1+\frac{k}{n-k+1}\right)^{-1/\alpha_n}.$$

This gives us the following approximate necessary and sufficient condition on the growth of tail index to reduce the latency for jobs of k tasks by increasing r from n/k to $(n+1)/k$,

$$\frac{\mathbb{E}[T_{n+1}]}{\mathbb{E}[T_n]} \lesssim 1 \iff \frac{\alpha_{n+1}}{\alpha_n} \gtrsim \frac{\log(1 + k/(n-k+2))}{\log(1 + k/(n-k+1))}.$$

The condition above and the ones given in Thm. 8 are quantitative expressions of our intuition; when the redundant load exerted on the system increases the runtime variability, it gets harder to reduce latency by executing jobs with more redundancy as the level of employed redundancy gets higher. When jobs are expanded with task replicas, the condition to reduce latency with more redundancy does not depend on the number of tasks k (scale) within the job; higher level of replication achieves less latency as long as the relative growth in the job expansion rate r is larger than the relative reduction in the tail index α (i.e., relative growth in tail heaviness) of task execution times. In Sec. III, coded redundancy is shown to be more effective for jobs that run at greater scale. Similarly here when jobs are expanded with coded tasks, increased runtime variability due to redundant load can be better compensated by jobs that run at greater scale. In addition, the threshold for redundancy to start incurring higher latency grows at a slower rate in r when coded tasks are used compared to using task replicas. This is due to the fact that coded redundancy is more efficient; it yields greater reduction in latency per introduced redundant task compared to replication.

V. Straggler Relaunch

Throughout this section, we assume task execution times are heavy tailed. There are two properties of heavy tailed task execution times that greatly affect the distributed job execution [33]. Firstly, the longer a task has taken to execute, the longer its average residual lifetime is expected to be. Secondly, the majority of the mass in a set of sample observations drawn from a heavy tailed distribution is contributed by few samples. This suggests that among all tasks within a job, few of them are expected to be stragglers with much longer completion time compared to the non-stragglers.

After launching a job, let us wait for a reasonably large Δ amount of time and check whether the job is completed or not. If the job is still running, we expect only a few tasks remaining which we refer to as stragglers. Heavy tailed nature of the task execution times suggests that the tasks straggling beyond time Δ are expected to take at least Δ more to complete on average. It also suggests that if a fresh copy is launched at time Δ for each straggling task, fresh copies are likely to complete before their corresponding old copies.

In this section, we show that *straggler relaunch*, that is, replacing the straggling tasks with fresh copies after waiting for some time, can yield significant reduction in cost and latency when the task execution times are *heavy tailed enough*. We investigate the level of tail heaviness required for straggler relaunch to be effective. The selection of the tasks to be relaunched is decided by the time Δ we wait before relaunching the remaining tasks. Untimely relaunch might be either late and cause delayed cancellation of the stragglers, or might be early and cause killing the non-straggler tasks as well. We find an approximation for the optimal time to perform straggler relaunch, which turns out to have a simple and insightful form. Lastly, we consider performing straggler relaunch jointly with adding redundant tasks into the job execution.

Exact expressions for the cost and latency of job execution with straggler relaunch are given in Thm. 9. Note that we assume relaunching tasks takes place instantly and does

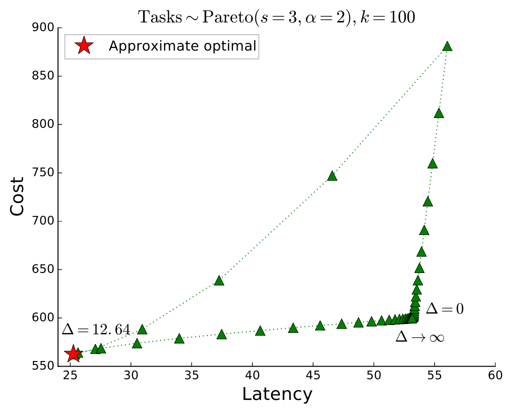


Fig. 9. Cost vs. latency of executing a job of 100 tasks by relaunching the remaining tasks after waiting some time Δ. Relaunch time Δ is varied from 0 to ∞ along the curve.

not incur any additional delay. Performing straggler relaunch before the minimum task completion time s causes meaningless work loss and further delays the job completion, while performing straggler relaunch at the right time significantly reduces the latency. The cost is a direct function of the latency in the absence of redundant tasks, hence reduced latency implies reduced cost as well (as illustrated in Fig. 9).

Theorem 9: Suppose task execution times are i.i.d. with $\mathrm{Pareto}(s, \alpha)$. *Consider executing a job of* k *tasks by relaunching all the remaining tasks after waiting some time* Δ. *Then, the distribution of job completion time is given as*

$$\begin{aligned}&\Pr\{T > t\}\\ &= 1 - \left(\mathbb{1}(t > s)\left(1 - (s/t)^{\alpha}\right)\right)^{k}\\ &\quad + \left(q + \mathbb{1}(t > \Delta)\left(1 - (\Delta/t)^{\alpha}\right)(1-q)\right)^{k}\\ &\quad + \left(q + \mathbb{1}(t > \Delta + s)\left(1 - (s/(t-\Delta))^{\alpha}\right)(1-q)\right)^{k}. \qquad (18)\end{aligned}$$

Latency is given as

$$\mathbb{E}[T] = \begin{cases} \Delta + L & \Delta \le s, \\ \Delta(1 - q^{k}) + L\big((s/\Delta - 1) \\ \qquad \times I(1 - q; 1 - 1/\alpha, k) + 1\big) & o.w. \end{cases} \qquad (19)$$

Cost with (C^c) *or without* (C) *task cancellation is given as*

$$\begin{aligned}\mathbb{E}[C^c] &= \begin{cases} k\Delta + \dfrac{1}{\alpha - 1}(ks\alpha - L) + k(1-q)\Delta & \Delta \le s, \\ \dfrac{\alpha}{\alpha - 1}\left(k(1-q)(s - \Delta) + ks\right) & o.w. \end{cases}\\ \mathbb{E}[C] &= \begin{cases} k\Delta + ks\dfrac{\alpha}{\alpha - 1} & \Delta \le s, \\ \dfrac{\alpha}{\alpha - 1}\left(ks(2-q)\right) - \dfrac{k\Delta(1-q)}{\alpha - 1} & o.w. \end{cases}\end{aligned} \qquad (20)$$

where $q = \mathbb{1}(\Delta > s)\left(1 - (s/\Delta)^{\alpha}\right)$, *and* $L = sk!\Gamma(1 - 1/\alpha)/\Gamma(k + 1 - 1/\alpha)$ *is the baseline latency of executing the job without straggler relaunch.*

Lemma 1: Suppose task execution times are distributed as $\mathrm{Pareto}(s, \alpha)$, *and let* T_{norel} *denote the baseline completion time for executing a job of* k *tasks without straggler relaunch. A sufficient condition for reducing the cost and latency of job execution by performing straggler relaunch is given by*

$$\mathbb{E}[T_{norel}] > 4s. \qquad (21)$$

This gives a looser sufficient condition on the tail index as

$$\alpha < \ln(k)/\ln(4). \tag{22}$$

Optimal relaunch time to execute the job with minimum cost and latency is approximately given as

$$\Delta^* \approx \sqrt{s\mathbb{E}[T_{norel}]} = s\sqrt{\frac{k!\Gamma(1-1/\alpha)}{\Gamma(k+1-1/\alpha)}}. \tag{23}$$

This implies that average fraction of the tasks that are relaunched by the optimal strategy is approximately given as

$$p^* \approx (s/\mathbb{E}[T_{norel}])^{\alpha/2} \approx \frac{\Gamma(1-1/\alpha)^{-\alpha/2}}{\sqrt{k+1}}. \tag{24}$$

Sufficient conditions and the approximations given above are asymptotic and become exact in the limit $k \to \infty$.

An approximate expression for the relaunch time Δ^* that minimizes the cost and latency of job execution is given in Lemma 1. The given approximation converges to the true optimal as k gets larger, e.g., approximate Δ^* is very close to the true optimal for the case shown in Fig. 9 with $k = 100$. Optimal relaunch time is an increasing function of the number of tasks k and the task sizes s, which intuitively makes sense. Also it is a decreasing function of α, meaning that it is better to relaunch earlier when the tail of task execution times is lighter, while for heavier tail, delaying relaunch further helps to identify the stragglers better. This is because relaunching tasks is a choice of canceling the work that is already completed in order to get possibly lucky and execute the replacement copies much faster. When the task execution times are heavier in tail, the residual lifetime of the straggler tasks is expected to be much larger, and the gain of relaunching stragglers can compensate for the work loss. However with lighter tailed task execution times, it is better to try our chance with relaunching earlier and keep the work loss limited.

As discussed above Δ^* gets smaller as α increases, but this does not imply relaunching a larger fraction of the job's tasks. When relaunching is performed after waiting Δ^*, fraction p^* of the tasks that are relaunched monotonically decreases[4] with α, that is, fewer tasks get relaunched on average by the optimal strategy as the tail gets lighter. In addition, p^* decreases with k, which means for jobs with larger number of tasks, optimal strategy dictates relaunching smaller fraction of the tasks. For instance, suppose $\alpha = 2$ and $k = 10$, then $p^* \approx 0.17$, which implies 17% of the tasks would need to be relaunched on average with the optimal strategy, while if $k = 100$, then only 6% of the tasks would need to be relaunched on average.

We assume relaunching tasks does not introduce any additional delay. Given that, the cost of job execution directly changes with the latency, thus, optimal relaunch time that minimizes latency also minimizes the cost. Note that cost of relaunching may not be ignored in practice, which is why results presented here on the performance of straggler relaunch can only be taken as optimistic guidelines.

For relaunching to be effective, work loss due to the cancellation of already running tasks should be compensated by the

[4]p^* is a monotonically decreasing function of α. As the tail of task execution times becomes very heavy; $\lim_{\alpha \to 1} \Gamma(1-1/\alpha)^{-\alpha/2} = 1$, and as the tail becomes very light; $\lim_{\alpha \to \infty} \Gamma(1-1/\alpha)^{-\alpha/2} \approx 0.749$.

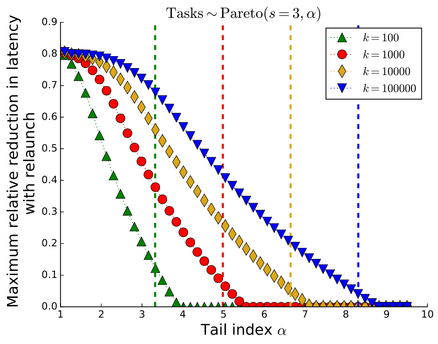


Fig. 10. Maximum reduction in the latency of executing a job of k tasks with straggler relaunch (relative to the baseline without relaunch) depends on the tail of the task execution times. Vertical dashed lines indicate the sufficient condition given on α in Lemma 1.

gain of not having to wait very long for the stragglers. In other words, straggler relaunch is effective only if the tail of task execution times is heavy beyond a level. Otherwise relaunching tasks hurts performance; it incurs additional cost and latency in the job execution. For instance, relaunching always hurts when task execution times have light tail, i.e., when the tail decays at least exponentially fast.

Lemma 1 presents an asymptotic sufficient condition for straggler relaunch to be effective, which has a particularly nice form; if the baseline latency without relaunching is greater than 4 times the minimum task completion time s, then relaunching stragglers at the right time will reduce the cost and latency of job execution. Reformulation of this condition in terms of the tail index α suggests that straggler relaunch is effective as long as α is less than a threshold, which is the same as saying the tail of task execution times should be heavy beyond a level. Note that this condition on the tail index α does not depend on the minimum task completion time s and is only proportional with the logarithm of the scale k of job execution, which we also validate by numerically computing the exact necessary and sufficient condition on α (see Fig. 10).

VI. Redundancy Together With Relaunch

In this section, we consider employing redundant tasks and straggler relaunch jointly for straggler mitigation.

A. Zero-Delay Redundancy With Relaunch

Firstly, we consider launching the redundant tasks (c replicas for each task or $n - k$ coded tasks) together with the k initial tasks of a job, then relaunching each remaining task (initial or redundant) after waiting some time Δ. Thm. 10 gives exact expressions for the latency and Lemma 2 presents an asymptotic sufficient condition for relaunching to be effective in reducing latency, and also presents an approximate value for the optimal relaunch time. Relaunch time Δ does not affect the level of added redundancy, hence the optimal relaunch time that minimizes latency also minimizes cost.

Theorem 10: Suppose task execution times are i.i.d. with $\text{Pareto}(s, \alpha)$. *Consider launching a job of k tasks together with redundant tasks then relaunching all remaining tasks after waiting some time Δ. Let $\mathbb{E}[T_{norel}]$ denote the baseline latency without straggler relaunch as given in Thm 5.*

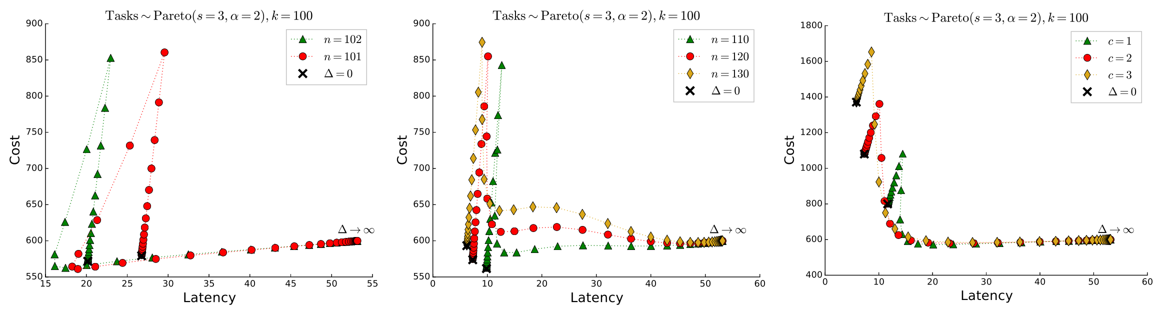


Fig. 11. Cost vs. latency curves for executing a job of 100 tasks by adding redundancy and performing straggler relaunch after time Δ. Each curve is plotted by interpolating between the incremental steps of time Δ.

When job is launched by adding c replicas for each task,

$$\mathbb{E}[T] = \begin{cases} \Delta + \mathbb{E}[T_{norel}] & \Delta \leq s, \\ \Delta(1-q^k) + \mathbb{E}[T_{norel}]\Big(1+ & \\ \quad (s/\Delta - 1)I(1-q; 1-1/\alpha, k)\Big) & o.w. \end{cases} \tag{25}$$

where $q = \mathbb{1}(\Delta > s)\left(1-(s/\Delta)^{(c+1)\alpha}\right)$.

When job is launched by adding $n-k$ *coded tasks,*

$$\mathbb{E}[T] = \begin{cases} \Delta + \mathbb{E}[T_{norel}] & \Delta \leq s, \\ \Delta I(1-q; n-k+1, k) + \mathbb{E}[T_{norel}]\Big(1+ & \\ \quad (s/\Delta - 1)I(1-q; n-k+1-1/\alpha, k)\Big) & o.w. \end{cases} \tag{26}$$

where $q = \mathbb{1}(\Delta > s)\left(1-(s/\Delta)^{\alpha}\right)$.

Lemma 2: Suppose task execution times are i.i.d. with $\mathrm{Pareto}(s, \alpha)$. *Let* $\mathbb{E}[T_{norel}]$ *denote the latency for a job of k tasks that is launched together with task replicas or coded tasks (without straggler relaunch), which is given in Thm. 5. A sufficient condition that guarantees reduction in cost and latency by also performing straggler relaunch is given as*

$$\mathbb{E}[T_{norel}] > 4s. \tag{27}$$

A looser sufficient condition is, when task replicas are used

$$\alpha < \frac{\ln(k)}{(c+1)\ln(4)}, \tag{28}$$

or when coded tasks are used

$$\alpha < \frac{\ln\left(n/(n-k+1)\right)}{\ln(4)}. \tag{29}$$

Optimal relaunch time for minimum cost and latency (either when task replicas or coded tasks are used) is approximately

$$\Delta^* \approx \sqrt{s\mathbb{E}[T_{norel}]}. \tag{30}$$

Sufficient conditions and the approximations given above are asymptotic and becomes exact in the limit $k \to \infty$.

Proof Sketch: Very similar to the proof of Lemma 1. □

Launching a job with redundant tasks mitigates the effect of stragglers, so does relaunching the stragglers after waiting some time. Therefore, the relative latency and cost reduction harvested from straggler relaunch decreases when it is used jointly with redundancy. Straggler relaunch is effective only when the effect of stragglers is significant, i.e., when the tail of task execution times is heavy beyond a level (cf. Lemma 1). Adding redundant tasks into the job execution already "cuts" some of the tail, hence the initial tail heaviness that is required for relaunching to be effective increases with the level of added redundancy. Sufficient conditions (28) and (29) given on the tail heaviness are asymptotic representations of this observation. The upper threshold given on the tail index as the sufficient condition decays (i.e., required tail heaviness increases) with the level of added redundancy faster when task replicas are used (decays as $1/(c+1)$) compared to using coded tasks (decays as $\ln\left(n/(n-k+1)\right)$).

B. Delayed Redundancy With Relaunch

Secondly, we consider adding redundant tasks and performing straggler relaunch jointly after waiting some time Δ. Cost and latency of job execution in this case are are presented in Thm. 11. Using these expressions, Fig. 11 plots the cost vs. latency tradeoff by varying Δ from 0 to ∞ for different levels of added redundancy.

When the number of added coded tasks is low, there exists an optimal time Δ that minimizes the cost and latency of job execution (Left, Fig. 11). This is the same observation that we previously made for the case of performing straggler relaunch without adding any redundancy (cf. Fig. 9). As the number of added coded tasks increases, redundancy becomes a greater effect on the cost and latency than straggler relaunch, hence waiting for some time before adding redundant tasks becomes ineffective to reduce the cost (Middle, Fig. 11). This is the same observation that we made previously for the case of employing delayed redundancy without straggler relaunch (cf. Fig. 1). When task replicas are used rather than coded tasks, regardless of the number of added replicas, delaying Δ is ineffective to reduce the cost (Right, Fig. 11). This is because replicating each remaining task after some time Δ even by one is enough to dominate the effect of straggler relaunch on the cost vs. latency tradeoff.

Theorem 11: Suppose task execution times are i.i.d. with $\mathrm{Pareto}(s, \alpha)$. *Let* $\mathbb{E}[T_{nored}]$ *denote the latency of executing a job of k tasks by relaunching each remaining task after some time* Δ *(without adding redundant task) as given in Thm. 9.*

Consider relaunching and adding c replicas for each remaining task after some time Δ. *Then, latency is given as*

$$\mathbb{E}[T] \approx \begin{cases} \Delta + sk!\dfrac{\Gamma(1-1/\tilde{\alpha})}{\Gamma(k+1-1/\tilde{\alpha})} & \Delta \leq s, \\ \mathbb{E}[T_{nored}] + f(\tilde{\alpha}) - f(\alpha). & o.w. \end{cases} \tag{31}$$

for $f(\alpha) = s\,\frac{\Gamma(1-1/\alpha)}{\Gamma(-1/\alpha)}B(k-kq+1, -1/\alpha)$.

Cost with (C^c) or without (C) task cancellation is given as

$$\mathbb{E}[C^c] = \begin{cases} k\Delta + ks(c+1)\dfrac{\tilde{\alpha}}{\tilde{\alpha}-1} & \Delta \le s, \\ \dfrac{k\alpha}{(\alpha-1)}(s - \Delta(1-q)) & \\ +k(1-q)\Delta + ks(c+1)(1-q)\dfrac{\tilde{\alpha}}{\tilde{\alpha}-1} & o.w. \end{cases}$$

$$\mathbb{E}[C] = \begin{cases} k\Delta + ks(c+1)\dfrac{\alpha}{\alpha-1} & \Delta \le s, \\ \dfrac{k\alpha}{(\alpha-1)}(s - \Delta(1-q)) & \\ +k(1-q)\Delta + ks(c+1)(1-q)\dfrac{\alpha}{\alpha-1} & o.w. \end{cases}$$

where $\tilde{\alpha} = (c+1)\alpha$ and $q = \mathbb{1}(\Delta > s)(1 - (s/\Delta)^{\alpha})$.

Consider adding $n-k$ coded tasks instead of task replicas. Then, latency is given as

$$\mathbb{E}[T] \approx \begin{cases} \Delta + s\dfrac{n!}{(n-k)!}\dfrac{\Gamma(n-k+1-1/\alpha)}{\Gamma(n+1-1/\alpha)} & \Delta \le s, \\ \Delta(1-q^k) + s\Big(\dfrac{B(n-kq+1,-1/\alpha)}{B(n-k+1,-1/\alpha)} & \\ +kB(q;k,1-1/\alpha) - q^k\Big) & o.w. \end{cases} \tag{32}$$

Cost with (C^c) or without (C) task cancellation is given as

$$\mathbb{E}[C^c] = \begin{cases} k\Delta + s\dfrac{n}{\alpha-1}\left(\alpha - \dfrac{\Gamma(n)}{\Gamma(n-k)}\dfrac{\Gamma(n-k+1-1/\alpha)}{\Gamma(n+1-1/\alpha)}\right) & \Delta \le s, \\ \dfrac{\alpha}{\alpha-1}(k(1-q)(s-\Delta) + ns) & \\ + k(1-q)\Delta - s(n-k)q^k & o.w. \\ - \dfrac{s}{\alpha-1}(n-k)\dfrac{B(n-kq+1,-1/\alpha)}{B(n-k+1,-1/\alpha)}. & \end{cases}$$

$$\mathbb{E}[C] = \begin{cases} k\Delta + ns/(1-1/\alpha) & \Delta \le s, \\ \dfrac{\alpha}{\alpha-1}\big(ks(1-q+q^k) & \\ + ns(1-q^k)\big) - \dfrac{k\Delta(1-q)}{\alpha-1} & o.w. \end{cases} \tag{33}$$

where $q = \mathbb{1}(\Delta > s)(1 - (s/\Delta)^{\alpha})$.

VII. Conclusions

This paper presented a theoretical performance evaluation of the two most widely deployed straggler mitigation techniques for distributed job execution: i) adding redundant tasks (together with the original tasks or after waiting some time) into the job and waiting only for a sufficient subset of all launched tasks for job completion, ii) waiting for some time after launching the job and relaunching its remaining tasks. We derived the cost and latency expressions for executing the job by applying either one of these techniques or both jointly. Using the derived expressions, we found the following guidelines for the application of these techniques: i) Waiting for some time before launching redundant tasks is not effective to reduce the cost of redundancy. ii) Launching a job with redundant tasks can reduce not only its latency but also its cost. iii) Launching a job with MDS coded tasks achieves less cost (hence incurs less additional load on the system) and latency than using task replicas. iv) Relaunching remaining tasks after waiting some time is effective only if the tail of task execution times is heavier beyond a level, and employing redundant tasks together with straggler relaunch increases this tail heaviness requirement.

In our system model, we abstract away the job dispatching and resource sharing dynamics by modeling execution times of tasks within a job as i.i.d. random variables. This rather lumped model allows deriving insightful expressions that allows evaluating and comparing widely deployed straggler mitigation techniques. However, application of these techniques modifies the system dynamics and it is necessary to augment the model to reflect the impact of this modification on task execution times. This is an ongoing challenge for us and Sec. IV presented a simulation driven attempt in this direction.

References

[1] M. F. Aktas, P. Peng, and E. Soljanin, "Effective straggler mitigation: Which clones should attack and when?" *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 45, no. 2, pp. 12–14, 2017.

[2] M. F. Aktas, P. Peng, and E. Soljanin, "Straggler mitigation by delayed relaunch of tasks," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 45, no. 3, pp. 224–231, 2018.

[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 59–72, 2007.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] G. Ananthanarayanan *et al.*, "Reining in the outliers in map-reduce clusters using mantri," in *Proc. OSDI*, Oct. 2010, vol. 10, no. 1, p. 24.

[6] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Design Implement.*, 2012, p. 2.

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

[8] X. Ouyang, P. Garraghan, R. Yang, P. Townend, and J. Xu, "Reducing late-timing failure at scale: Straggler root-cause analysis in cloud datacenters," in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2016, pp. 1–3.

[9] H. Zhou, Y. Li, H. Yang, J. Jia, and W. Li, "BigRoots: An effective approach for root-cause analysis of stragglers in big data system," 2018, *arXiv:1801.03314*. [Online]. Available: https://arxiv.org/abs/1801.03314

[10] J. Dean. (2012). *Achieving Rapid Response Times in Large Online Services*. [Online]. Available: https://research.google.com/people/jeff/latency.html

[11] N. J. Yadwadkar and W. Choi, "Proactive straggler avoidance using machine learning," Univ. California, Berkeley, Berkeley, CA, USA, White Paper, 2012.

[12] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proc. OSDI*, 2008, vol. 8, no. 4, pp. 29–42.

[13] S. Melnik *et al.*, "Dremel: Interactive analysis of Web-scale datasets," *Proc. VLDB Endowment*, vol. 3, pp. 330–339, Sep. 2010.

[14] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement.*, 2013, pp. 185–198.

[15] A. Vulimiri *et al.*, "Low latency via redundancy," in *Proc. 9th ACM Conf. Emerg. Netw. Exp. Technol.*, 2013, pp. 283–294.

[16] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized speculation-aware cluster scheduling at scale," in *Proc. ACM Conf. Special Interest Group Data Commun.*, 2015, pp. 379–392.

[17] K. Gardner, M. Harchol-Balter, A. Scheller-Wolf, and B. Van Houdt, "A better model for job redundancy: Decoupling server slowdown and job size," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3353–3367, Dec. 2017.

[18] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Queue*, vol. 14, no. 1, p. 10, 2016.

[19] G. Joshi, Y. Liu, and E. Soljanin, "Coding for fast content download," in *Proc. 50th Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Oct. 2012, pp. 326–333.

[20] L. Huang, S. Pawar, H. Zhang, and K. Ramchandran, "Codes can reduce queueing delay in data centers," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2012, pp. 2766–2770.

[21] S. Kadhe, E. Soljanin, and A. Sprintson, "When do the availability codes make the stored data more available?" in *Proc. 53rd Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Sep./Oct. 2015, pp. 956–963.

[22] S. Dutta, V. Cadambe, and P. Grover, "Short-Dot: Computing large linear transforms distributedly using coded short dot products," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 2100–2108.

[23] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Trans. Inf. Theory*, vol. 64, no. 3, pp. 1514–1529, Mar. 2017.

[24] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient coding," 2016, *arXiv:1612.03301*. [Online]. Available: https://arxiv.org/abs/1612.03301

[25] S. Li, S. M. M. Kalan, A. S. Avestimehr, and M. Soltanolkotabi, "Near-optimal straggler mitigation for distributed gradient methods," 2017, *arXiv:1710.09990*. [Online]. Available: https://arxiv.org/abs/1710.09990

[26] C. Karakus, Y. Sun, S. Diggavi, and W. Yin, "Straggler mitigation in distributed optimization through data encoding," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5434–5442.

[27] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding," 2018, *arXiv:1801.07487*. [Online]. Available: https://arxiv.org/abs/1801.07487

[28] W. Halbawi, N. Azizan, F. Salehi, and B. Hassibi, "Improving distributed gradient descent using Reed–Solomon codes," in *Proc. IEEE Int. Symp. Inf. Theory*, Jun. 2018, pp. 2027–2031.

[29] D. Wang, G. Joshi, and G. Wornell, "Using straggler replication to reduce latency in large-scale parallel computing," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 3, pp. 7–11, 2015.

[30] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Towards understanding heterogeneous clouds at scale: Google trace analysis," Intel Sci. Technol. Center Cloud Comput., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. ISTC-CC-TR-12-101, 2012, p. 84.

[31] G. Ananthanarayanan *et al.*, "GRASS: Trimming stragglers in approximation analytics," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement.*, 2014, pp. 289–302.

[32] J. Nair, A. Wierman, and B. Zwart, "The fundamentals of heavy-tails: Properties, emergence, and identification," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 1, pp. 387–388, 2013.

[33] M. E. Crovella, "Performance evaluation with heavy tailed distributions," in *Proc. Workshop Job Scheduling Strategies Parallel Process.*, 2001, pp. 1–10.

[34] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: Format + schema," Google, Mountain View, CA, USA, White Paper, 2011.

[35] Y. Wang, S. Jain, M. Martonosi, and K. Fall, "Erasure-coding based routing for opportunistic networks," in *Proc. ACM SIGCOMM Workshop Delay-Tolerant Netw.*, 2005, pp. 229–236.

[36] G. Joshi, "Boosting the throughput of a multi-server system via adaptive task replication," in *Proc. ACM SIGMETRICS*. MAMA, 2017.

[37] D. L. Brown *et al.*, "Cross cutting technologies for computing at the exascale," US DoE Office Adv. Sci. Comput. Res., Nat. Nucl. Secur. Admin., Washington, DC, USA, Tech. Rep. PNNL-20168, 2010. [Online]. Available: https://digital.library.unt.edu/ark:/67531/metadc841613/

[38] S. Hofmeyr, C. Iancu, J. Colmenares, E. Roman, and B. Austin, "Time-sharing redux for large-scale HPC systems," in *Proc. IEEE 18th Int. Conf. High Perform. Comput. Commun., IEEE 14th Int. Conf. Smart City, IEEE 2nd Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, Dec. 2016, pp. 301–308.

[39] F. W. J. Olver *et al.*, Eds., *NIST Digital Library of Mathematical Functions, Release 1.0.24*. 2019. [Online]. Available: http://dlmf.nist.gov/

[40] R. Bapat and M. Beg, "Order statistics for nonidentically distributed variables and permanents," *Sankhyā: Indian J. Statist., A*, vol. 51, no. 1, pp. 79–93, 1989.

[41] B. C. Arnold, N. Balakrishnan, and H. N. Nagaraja, *A First Course in Order Statistics*. Philadelphia, PA, USA: SIAM, 2008.

[42] B. C. Arnold, *Pareto Distribution*. Hoboken, NJ, USA: Wiley, 2015.

[43] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: From I/O Ports to Process Management*. Sebastopol, CA, USA: O'Reilly Media, 2005.

[44] J. Nair, A. Wierman, and B. Zwart, "Tail-robust scheduling via limited processor sharing," *Perform. Eval.*, vol. 67, no. 11, pp. 978–995, 2010.

[45] W. Leland and T. J. Ott, *Load-Balancing Heuristics and Process Behavior*, vol. 14, no. 1. New York, NY, USA: ACM, 1986.

[46] M. Harchol-Balter and A. B. Downey, "Exploiting process lifetime distributions for dynamic load balancing," *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 253–285, 1997.

[47] Y. Chen, A. Ganapathi, R. Griffith, and R. H. Katz, "Analysis and lessons from a publicly available Google cluster trace," Dept. EECS, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2010-95, 2010, vol. 94.

[48] N. Bansal and M. Harchol-Balter, "Analysis of SRPT scheduling: Investigating unfairness," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 29, no. 1, pp. 279–290, 2001.

[49] M. Rytgaard, "Estimation in the Pareto distribution," *ASTIN Bull., J. IAA*, vol. 20, no. 2, pp. 201–216, 1990.

[50] I. B. Aban, M. M. Meerschaert, and A. K. Panorska, "Parameter estimation for the truncated Pareto distribution," *J. Amer. Stat. Assoc.*, vol. 101, no. 473, pp. 270–277, 2006.

[51] A. P. Zwart, *Queueing Systems With Heavy Tails*. Eindhoven, The Netherlands: Technische Univ. Eindhoven, 2001.

[52] T. Sakurai, "Approximating M/G/1 waiting time tail probabilities," *Stochastic Models*, vol. 20, no. 2, pp. 173–191, 2004.

[53] M. Olvera-Cravioto, J. Blanchet, and P. Glynn, "On the transition from heavy traffic to heavy tails for the M/G/1 queue: The regularly varying case," *Ann. Appl. Probab.*, vol. 21, no. 2, pp. 645–668, 2011.

[54] F. Tricomi and A. Erdélyi, "The asymptotic expansion of a ratio of gamma functions," *Pacific J. Math.*, vol. 1, no. 1, pp. 133–142, 1951.

[55] R. Garrappa, "Some formulas for sums of binomial coefficients and gamma functions," *Int. Math. Forum*, vol. 2, no. 15, pp. 725–733, 2007.

[56] W. Gautschi, "Some elementary inequalities relating to the gamma and incomplete gamma function," *Stud. Appl. Math.*, vol. 38, nos. 1–4, pp. 77–81, 1959.

Mehmet Fatih Aktaş received the B.Sc. degree in electrical engineering from Bilkent University, Ankara, Turkey, and the M.Sc. degree in computer engineering from Rutgers University, where he is currently pursuing the Ph.D. degree. His research interests include distributed computer systems, applied probability, and reinforcement learning.

Emina Soljanin (S'90–M'95–SM'03–F'14) was a Distinguished Member of Technical Staff for 21 years in Mathematical Sciences Research with Bell Labs before moving to Rutgers in January 2016. She is currently a Professor with Rutgers University. She is also a coding, information, and, more recently, queuing theorist. Her interests and expertise are wide. Over the past quarter of the century, she has participated in numerous research and business projects, as diverse as power system optimization, magnetic recording, color space quantization, hybrid ARQ, network coding, data and network security, and quantum information theory. Dr. Soljanin is a co-organizer of the DIMACS 2001–2005 Special Focus on Computational Information Theory and Coding and 2011–2015 Special Focus on Cybersecurity. She is a 2017 outstanding alumnus of the Texas A&M School of Engineering, a 2016–2017 Distinguished Lecturer for the IEEE Information Theory Society, and the 2019 President for the Society. She served as an Associate Editor for Coding Techniques for the IEEE TRANSACTIONS ON INFORMATION THEORY, on the Information Theory Society Board of Governors, and in various roles on other journal editorial boards and conference program committees.