Latin Square Job Scheduling for Distributed Data Processing on Warehouse-Scale Computers

Kevin Exton
School of Computing and Information Systems
The University of Melbourne
Melbourne, Australia
Email: kevin.exton@student.unimelb.edu.au

Maria A. Rodriguez
School of Computing and Information Systems
The University of Melbourne
Melbourne, Australia
Email: marodriguez@unimelb.edu.au

***Abstract*—Large-scale parallelizable data processing jobs, executed by frameworks like Spark, frequently have heavily skewed response time distributions with long tail times. This phenomenon is known as straggling and it occurs when one or more of the parallel tasks belonging to a job takes much longer than normal to complete. As a single root cause for straggling in large-scale systems is difficult to identify, we consider it to be a symptom of temporary node impairment caused by random bursts of interference. Since latin squares are commonly used in many applications that need to mitigate external sources of variation or interference, we examine how they might benefit modern distributed data processing workloads. Based on the latin square combinatorial block design, we develop a scheduling policy that we call the latin square scheduling policy. Using a combination of simulation and theoretical analysis, we argue that our policy can outperform other common schemes such as random, round-robin, Sparrow, and Peacock, when there is a sufficient quantity of time-dependent intermittent burst interference.**

I. Introduction

Data processing is an integral part of modern computing services like search, language translation, and image recognition. These services are often constructed using a complex web of highly interconnected components that are hosted on extremely large-scale computers, sometimes called warehouse-scale computers [1]. Scheduling parallelizable data processing jobs on these warehouse-scale machines, especially using frameworks like MapReduce [2], is plagued by a phenomenon known as "straggling" that is characterized by heavily skewed end-to-end job delay (or response) time distributions [3]. As response times are necessarily non-negative values, shorter schedules can be produced by eliminating extraneous sources of variation. We consider schedules to be shorter when the job response time distribution has smaller delays at all of the quantiles. This fact is plain to see if we break down the delay time, T, of a job into three components:

$$T = \tau + Z + \xi \tag{1}$$

where τ is a non-negative constant that denotes the minimum possible delay time for a job, Z is a non-negative random variable determined by the variation in computational inputs, and ξ is a non-negative random variable that is contributed by extraneous sources of variation (Z and ξ not necessarily independent).

Extraneous sources of variation can have various root causes, for large (warehouse) scale computers, Dean and Barroso [3] identify some of these to be: global resource sharing, maintenance activities, power limits, garbage collection, and dynamic energy management. In general, we can interpret the aggregate variation from all these sources as being a form of burst interference that is spread out over the dimensions of *time* and *compute resources* (disk, network, CPU etc.). Using an array of binary variables, $x_i(t)$, where each variable is equal to 1 when the i'th compute resource is experiencing burst interference at time t and 0 otherwise, we can represent this burst interference using a collection of parallel timelines, as in Fig. 1. Whenever $x_i(t)$ is equal to 1, we say that the processing capabilities of the i'th compute resource are *degraded*. Degraded resources can still execute the tasks that they have been assigned, but they will do so at a lower performance level (with respect to a given performance measurement) compared to an undegraded resource. Thus, eliminating extraneous sources of variation means that we need to limit, but not eradicate, the amount of computational work that is being performed by degraded resources at any given point in time.

By dividing the resource timelines into time slots. We can group instances of burst interference into elements of a two-dimensional array (or blocks in a two-dimensional grid), where each element of the array, $x_{i,j}$, is equal to 1 if its associated time slot contains an instance of burst interference and 0 otherwise, as in Table I. We argue that it is reasonable to assume that burst interference is a random event, since if it were a deterministic event, systems would be designed in such a way that these events could not cause any interference. Under the assumption of random burst interference, the two-dimensional array of $x_{i,j}$'s indicates to us that the processing capabilities of a computer will vary randomly between blocks but may be more consistent within each block. When variations within a block are smaller than variations between blocks, then these sources of variance are sometimes called blocking factors (especially in the design of experiments [4, Ch. 1]), and the techniques that are used to reduce the influence of blocking factors are called block designs [4, Ch. 10]. Although our job scheduling problem is different from that of the design of experiments, we will be investigating job

TABLE I
ARRAY REPRESENTATION OF BURST INTERFERENCE BLOCKS

Time Slot	0	1	2	3
x_1	0	1	1	1
x_2	0	0	1	0
x_3	0	1	0	0

scheduling algorithms based on these combinatorial block designs, especially, that of *latin square block designs* [4, Ch. 12].

A. Key Contributions

In section IV we develop the design of a scheduling policy based on latin square combinatorial block designs. Here, we provide a formal listing of a scheduling algorithm as well as some discussion on the benefits of our latin square scheduling policy relative to other schedulers in the literature. Subsequently, we will move to the analysis of a small-scale system where we will demonstrate that under Poisson arrival and mutually independent exponential service time assumptions, our latin square scheduling algorithm can provide as much as a 50% reduction to expected response times relative to a simple round-robin scheme. In section VI, we design a simulation experiment to compare our latin square scheduling policy against four other policies: round-robin[1], random[2], Sparrow [6], and Peacock [7]. We have chosen these as our reference points to compare against, as they are well-understood designs. In particular, Sparrow [6] has been used as a reference point in many subsequent designs [7]–[10], and Peacock [7] is a contemporary solution that has been demonstrated to reduce job latencies by as much as 73% relative to Eagle [10] and as much as 91% relative to Sparrow [6]. Our simulations first reproduce the results originally presented by Sparrow [6] and Peacock [7]; subsequently, we compare the job schedules produced by these and our latin square scheduling policy, both with and without the effects of burst interference. Our results demonstrate that our latin square scheduling policy produces extremely consistent schedules under a wide range of operating conditions, and therefore produces much shorter schedules than the other policies we evaluated when the frequency of burst interference is unknown, or varies over time.

II. BACKGROUND AND RELATED WORK

The end-to-end job delay time skew introduced by "straggling" has been a high-profile research problem since at least as early as 2008 with the publication of MapReduce [2]. Early solutions attribute this skew to errors within the computer such as dropped packets or server failures, funneling the focus towards task retrying strategies as implemented by MapReduce [2]. Later, repetition or replication strategies were implemented by schedulers to reduce skew by preemptively eliminating straggling tasks [11]. The repetition strategy is also referred to as speculative execution [8], and has seen a lot of adoption in the literature [11]–[13]. Recently, there has been a revival of interest in the topic of skew and straggler mitigation in the information theory community that has developed around the application of combinatorial and algebraic structures similar to traditional error-correcting codes [14]. In particular, there has been much work around reducing the end-to-end delay times of machine learning related jobs that rely heavily on distributed matrix arithmetic [15], [16]. While these algorithms seem to be effective, they are limited in their application to numerical and scientific computational workloads. While some combinatorial approaches have been studied [17], the scope of this other work remains quite focused on applications in machine learning. We argue that some of these coded distributed computing algorithms have job scheduling applications outside of strictly machine learning workloads.

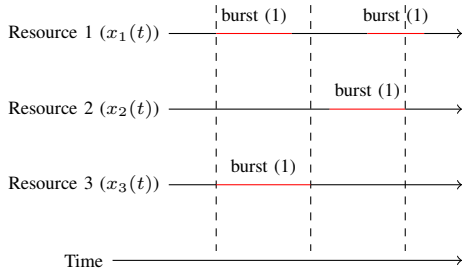

Fig. 1. Burst Interference Spread over 3 Compute Resources

The foundations of the bridge that we build between coded distributed computing algorithms and traditional job scheduling for warehouse scale computers will be laid on top of the *latin square combinatorial block design*. Latin squares have a rich history of being used to mitigate or manage unwanted sources of variation, or interference, in a variety of disciplines. Of particular note, is how they were used in wireless communications for the development of code-division multiple access schemes [18], and how they are used in the design of experiments to mitigate unwanted sources of variation called *blocking factors* [4, Ch. 1].

As they are combinatorial block designs, we argue that latin squares will be useful for far more general workloads than simply jobs consisting of distributed matrix arithmetic. We demonstrate that they are competitive against well-established approaches in the applied job scheduling literature, like Sparrow [6] and Peacock [7], especially, under the assumption of *uncorrelated* service time characteristics caused by random bursts of performance degrading interference.

III. THE WAREHOUSE-SCALE COMPUTER — SYSTEM MODEL

The model we use of a warehouse-scale computer is made of four components and six interfaces, illustrated in Fig. 2. The four components of a computer represent logical groupings

[1]Round-robin assigns each subsequent task to the next server in a rotating list of servers.

[2]Random scheduling is equivalent to a power-of-k [5] scheme with $k = 1$.

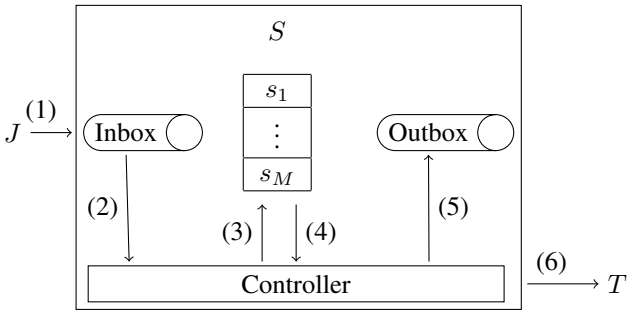

Fig. 2. A Warehouse-Scale Computer with M Parallel Servers.

of the resources necessary for executing applications on a warehouse-scale machine. The inbox (connected to interfaces (1) and (2)) and outbox (connected to interface (5)) components store input and output data for applications that are installed on the computer. The server set (connected to interfaces (3) and (4)) is a set of M parallel servers that represents the total amount of compute resources (disk, memory, CPU, etc.) available to the controller for executing application instructions. The controller (connected to all interfaces) is responsible for ensuring the unidirectional flow of data through the system over interfaces (1)-to-(5) and emitting relevant performance measurements over interface (6). The input to the computer is a sequence of jobs, $J = \{j_1, \ldots\}$, where each job $j_i \in J$ consists of $n_i \in \mathbb{N}$ operations (tasks) that must be executed by the computer to complete or "clear" the job. Upon arrival, a job is moved by the controller to the job inbox on interface (1) to be scheduled for execution. On interface (2), the controller then decides how to sequence the tasks in the inbox for scheduling. On interface (3), the controller assigns server resources (using any algorithm that does not violate the strict unidirectional flow of data from (1)-to-(5)) to tasks in the order of arrival from interface (2). On interface (4) the controller collects the task outputs from the parallel servers in the machine. On interface (5) the controller sends task output data to the job outbox, a job is completed or "cleared" when every task associated to the job has an output in the outbox. Finally, on interface (6), the controller emits performance measurements relevant to the control plane actions taken on interfaces (1)-to-(5). We have implemented a simulation of our warehouse-scale computing model that is available from [19].

IV. The Latin Square Scheduling Policy

For any $v \in \mathbb{N}$, a latin square is a $v \times v$ array filled with v distinct items such that every item appears in every row and every column exactly once. Thus, each distinct item will appear in the array a total of v times. Our policy treats each row of the latin square as a sequencing of tasks that is associated to a parallel server, i.e., the m'th parallel server is assigned tasks to be executed in an order determined by the m'th row of the latin square. The policy guarantees that, for a square with v tasks, v parallel servers will always be working on the tasks in the first column of the square. Initially, the

```
1:  procedure MakeLatinSquare(job)
2:      square ← []
3:      for i ← 0 ... v − 1 do
4:          row ← []
5:          for j ← 0 ... v − 1 do
6:              task ← job[ (j + i) mod v]
7:              Append(row, task)
8:          end for
9:          Append(square,row)
10:     end for
11:     return square
12: end procedure
    ▷ main scheduling loop
13: procedure Schedule(servers, job)
14:     square ← MakeLatinSquare(job)
15:     nremaining ← v
16:     for i ← 0 ... v − 1 do
17:         row ← square[i]
18:         AssignTask(servers[i], row[0])
19:     end for
20:     repeat
    ▷ wait for a finished task to be emitted on interface (4).
21:         task ← Await((4))
22:         nremaining ← nremaining − 1
23:         for row in square do
24:             Remove(row, task)
25:         end for
26:         for i ← 0 ... v − 1 do
27:             row ← square[i]
28:             if nremaining > 0 then
29:                 UpdateTask(servers[i], row[0])
30:             else
31:                 UpdateTask(servers[i], null)
32:             end if
33:         end for
34:     until nremaining = 0
35: end procedure
```

Fig. 3. Latin Square Scheduling Algorithm

policy assigns the task in the first column of each row to its corresponding server and then waits for a task to finish executing. Upon receiving an output emitted on interface (4), for the current task assigned to the m'th server, the controller will remove all v copies of that task from the latin square leaving a $v \times (v-1)$ rectangle that represents the sequencing of the $v-1$ unfinished tasks on v parallel servers. Finally, to guarantee that the v servers are only working on tasks in the first column of the updated array, the controller will stop (preempt) any servers that are no longer working on the correct task and start them working on the next task in the sequence from the updated array. A formal description of our latin square scheduling policy is given in Fig. 3. Consider a job, j_i, with 3 parallelizable tasks (e.g., a fork-join job with 3 parallel branches), that is scheduled on 3 available parallel servers, s_1, s_2, s_3, such that the first three rows of Table II depicts the initial 3×3 latin square schedule of tasks (line 16

TABLE II
LATIN SQUARE SCHEDULE FOR JOB WITH 3 PARALLEL TASKS.

	Index	0	1	2
Initial Schedule	s_1	$j_{i,1}$	$j_{i,2}$	$j_{i,3}$
	s_2	$j_{i,2}$	$j_{i,3}$	$j_{i,1}$
	s_3	$j_{i,3}$	$j_{i,1}$	$j_{i,2}$
$j_{i,1}$ **completes on** s_1	s_1	$j_{i,2}$	$j_{i,3}$	
	s_2	$j_{i,2}$	$j_{i,3}$	
	s_3	$j_{i,3}$	$j_{i,2}$	
$j_{i,2}$ **completes on** s_2	s_1	$j_{i,3}$		
	s_2	$j_{i,3}$		
	s_3	$j_{i,3}$		
$j_{i,3}$ **completes on** s_3	s_1			
	s_2			
	s_3			

of Fig. 3). The subsequent rows in Table II list the sequence of task assignment rectangles that our policy would generate (lines 20 to 34 of Fig. 3) from an example sequence of task completions given by: $j_{i,1}$ completes on s_1, then $j_{i,2}$ completes on s_2, then $j_{i,3}$ completes on s_3.

A. Latin Square Block Design Benefits

There is a great deal of empirical evidence to suggest that redundant or speculative execution schemes can be effective at scale [8], [11]–[13]. Under random burst interference assumptions, the redundancy introduced by these speculative execution schemes increases the probability that tasks will run in time slices that are experiencing fewer and less severe bursts of interference. Unfortunately, without a preemption mechanism, speculatively scheduled tasks will continue to consume compute resources long after they are needed. Hence, these speculative execution schemes are better suited for scheduling jobs consisting of small tasks, on account of the lower degree of wasted system capacity.

Our first improvement on these pre-existing speculation strategies comes from the inclusion of preemption. Preemption is a technique that in other contexts would be dismissed due to the resource consumption overhead [20]. However, speculative execution schemes are already consuming additional resources by launching redundant task copies. Therefore, preemption becomes a technique that reduces the total commitment of resources that speculative execution requires. In fact, under mutually independent exponential service time and Poisson arrival assumptions, the analysis by Gardner et. al. [21] demonstrates that *speculation with preemption* comes at *no cost* to system capacity, and therefore resource consumption.

Our second improvement on pre-existing speculation strategies comes from rearranging the order of execution to ensure that the sequences of redundant tasks assigned to each parallel server are orthogonal (every task appears in every column and every row *only once*). By staggering the launch of redundant tasks over time (and parallel servers), we increase the likelihood that speculation will reduce end-to-end job delays by widening the circumstances under which speculation might be effective. Specifically, while traditional speculative execution techniques will reduce end-to-end delays for workloads that have service time distributions with either constant or decreasing hazard rate functions (e.g., exponential and hyper-exponential distributions). Latin square speculation will, in addition to these, also be effective for workloads that have service time distributions with an upside-down bathtub-shaped hazard rate function[3].

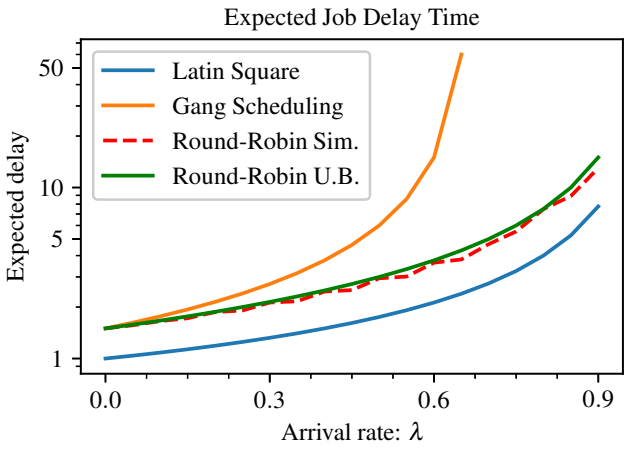

Fig. 4. Expected Job Delay for System with 2 Servers.

V. SMALL SCALE PERFORMANCE ANALYSIS

To estimate the performance improvement that our latin square scheduling policy provides, we conduct a mathematical analysis of a much smaller scale system with $M = 2$ parallel servers. We argue that the implicit assumptions in our analysis of negligible task assignment and preemption delays are reasonable at small scales as it is likely that any limitations of networked communications can be mitigated under these conditions. Hence, the magnitude of the relative improvement to system performance at small scales offers an insight into the performance of larger scale systems that are constructed from the parallel arrangement of many small-scale modules.

For the remainder of this section, we assume the following system parameterization:

- That our system is composed of $M = 2$ parallel servers.
- That task processing delays on both servers have mutually independent exponentially distributed service times with identical average service rate: $\mu = 1$.
- That all arriving jobs consist of $v = 2$ parallelizable tasks, and that jobs arrive according to a Poisson process with rate λ.

Under these assumptions, Fig. 4 illustrates expected job delay time vs arrival rate for a gang scheduling scheme [23], a round-robin scheme, and our latin square scheduling policy. From Fig. 4 we can quickly draw the conclusion that our latin square

[3]The hazard rate function of a distribution is the p.d.f. divided by the survival function: $h(x) = \frac{f(x)}{1-F(x)}$. An upside-down bathtub-shaped function is one where $h(x)$ first increases, then decreases, e.g., [22]

TABLE III
TASK SERVICE TIME PARAMETERIZATIONS

Servers	Jobs	
	Correlated Processing/Homogeneous Tasks Uncorrelated Processing/Homogeneous Tasks	Correlated Processing/Heterogeneous Tasks Uncorrelated Processing/Heterogeneous Tasks

scheduling policy significantly outperforms both a round-robin and a gang scheduling scheme. Compared to round-robin we can expect a relative reduction in expected job delays by up to 50%. We outline the analysis used to generate Fig. 4.

A. Gang Scheduling

Gang scheduling is a time-sharing based scheduling policy [23]. The parallel resources in a gang scheduling scheme are divided into a series of time slices that are time-shared among the running jobs. The time slices in a gang scheduling scheme are synchronized so that the parallel tasks of a job are always running simultaneously. By synchronizing parallel task execution to common time intervals, gang scheduling performance approximates a single server system that is using a processor sharing queuing discipline. We can estimate the expected delay of the gang scheduling policy by using the well-established expected delay formula for processor sharing queues [24, Ch. 4]:

$$E[D] = \frac{E[S]}{1-\rho}$$

where $E[S]$ is the expected service time (the delay time as $\rho \to 0^+$) of the workload, and ρ is the system utilization. Given that our servers have mutually independent exponentially distributed task service times, we can evaluate the expected service time of a job quickly by applying the expected value formula for the maximum of two i.i.d. exponential random variables [25].

$$E[S] = \frac{H_2}{\mu} = 1.5$$

where $H_2 = \sum_{i=1}^{2} \frac{1}{i}$ is the second harmonic number, and $\mu = 1$. Hence, by applying gang scheduling, the expected job delay time becomes:

$$\begin{aligned} E[D] &= \frac{E[S]}{1-\rho} \\ &= \frac{1.5}{1-\frac{\lambda}{1.5}} \end{aligned}$$

B. Round-Robin

Under our assumptions, a round-robin scheduled system has two parallel queues Q_1, Q_2 with a common Poisson arrival process of rate λ but i.i.d. exponentially distributed servicing processes with mean 1. The end-to-end delay of tasks routed to server 1 and 2 will be denoted D_1, and D_2 respectively. Therefore, the end-to-end delay of a job will be:

$$D = \max(D_1, D_2)$$

Separately, D_1 and D_2 are both exponentially distributed with mean $\frac{1}{\mu-\lambda}$, however, the common Poisson arrival process means that D_1 and D_2 are not *independent* random variables, so we can not simply apply the formula for the maximum of two i.i.d. exponential random variables as we did for gang scheduling. Instead, by applying Baccelli's [26] approximation using *associated random variables*, we can use the expected value of the maximum of two i.i.d. exponential random variables with mean $\frac{1}{\mu-\lambda}$ as an *upper bound* on the expected job delay time. Now we can apply the same formula as in the gang scheduling case to find the following upper bound for the expected job delay time for round-robin scheduling:

$$E[D] = E[\max(D_1, D_2)] \leq \frac{1.5}{1-\lambda}$$

In Fig. 4, we have used a simulation to demonstrate that under our small-scale system assumptions, this upper bound is quite a good approximation of the actual expected job delay times.

C. Latin Square

Under these small-scale system assumptions of $v = M = 2$, our latin square scheduling policy *guarantees* that job service times will be:

$$S = \sum_{i=1}^{2} \frac{X_i}{2}$$

where the X_i are i.i.d. exponential random variables with mean 1. Therefore, the job service times will be distributed according to a gamma distribution with shape, $\alpha = 2$, and rate, $\beta = 2$. Since we have assumed that $v = M = 2$, the latin square redundant execution structure also ensures that all jobs waiting in the queue will be blocked until the job at the head of the queue has finished being serviced. This means that we can evaluate the expected delay of our latin square scheduling policy by applying the Pollaczek-Khinchine formula [27, Ch. 16]:

$$\begin{aligned} E[D] &= \frac{\rho + \lambda\mu\sigma^2}{2(\mu-\lambda)} + \frac{1}{\mu} \\ &= \frac{1.5\lambda}{2(1-\lambda)} + 1 \end{aligned}$$

VI. SIMULATION DESIGN AND EVALUATION

We have implemented a simulation of the system model illustrated in Fig. 2 (available from [19]). In addition to the key components and interfaces, our simulation adds a network delay process between the servers and the controller so that all the communication delays in our system can be parameterized and included into our policy evaluation. To cover a number of different baseline reference points, we have

TABLE IV
SIMULATION PROCESS PARAMETERS, $\nu = 1$ AND $\mu = 1$.

Process	Parameterization		
Job Arrival Process	**num. tasks per job**	**job interarrival times**	**task interarrival times**
	100	Exponential $\left(\frac{1}{2} \times \frac{\mu}{10^{-2}}\right)$	Constant (0).
Task Latency Process	**num. servers in the cluster**		
	10 000		
	server task processing parameterization		
		correlated server processing	**uncorrelated server processing**
	homogeneous tasks	All tasks within a job have identical processing times sampled from an exponential distribution with rate μ.	Task processing times are distributed i.i.d. according to an exponential distribution with rate μ.
	heterogeneous tasks	Task processing times are sampled from an exponential distribution with rate μ. Distinct tasks are sampled independently but copies are identical to their originals.	Average task processing rate, ν, is sampled i.i.d. from an exponential distribution with rate μ. Task processing times are then sampled i.i.d. from an exponential distribution with rate ν.
Network Delay Process	**half round-trip delay time**	Gamma$(\alpha = 10^2, \beta = 10^4)$	

compared our latin square policy to a random, a round-robin, the Sparrow [6], and the Peacock [7] scheduling policies. Random and round-robin policies were chosen as they are very common cluster scheduling strategies, for instance, the OpenWhisk [28] serverless framework uses a random policy to assign function invocations to invokers; and, when configured to use IP Virtual Server [29], Kubernetes [30] will, by default, use a round-robin policy for layer 4 load balancing. We chose Sparrow [6], though, because it has been used as a common reference in many modern scheduler designs [8]–[10], which makes it useful as a baseline for assessing the relative improvement of our latin square policy against other scheduling policies in the literature. Peacock [7] was chosen because it is a contemporary design that has demonstrated significant performance improvements relative to Sparrow [6] and its successors: Hawk [9], and Eagle [10]. Table IV details how the simulation has been parameterized. These parameters have been chosen so that we can reproduce the results in the original Sparrow article [6]; therefore, we can provide a like-for-like comparison between our latin square and the Sparrow scheduling policy.

A. Alternative to Trace Driven Simulation

It is common in the literature for authors to evaluate their scheduling policies on simulated inputs that are taken from a workload trace [7], [9], [10]. Common traces include those taken by Google [31] and Facebook [32]. Since our latin square scheduling policy is designed to modulate *service times* rather than *waiting times*, we argue that traces like these can not be used to evaluate our policy as service times in these traces have already been realized, thus, preventing any further changes and improvements that could still be made. Furthermore, without detailed measurements taken for each and every task in a given trace, it is impossible to determine the task service time distribution in a way that accurately captures the statistical characteristics of the original workload. For instance, the approach of generating i.i.d. service times from the empirical service time distribution of a trace is unable to correctly simulate any time-based dependencies that may exist both within and between jobs. The difference in expected delay between the theoretical upper bound and the small-scale simulation of round-robin scheduling given in section V makes this phenomenon apparent. It also demonstrates that setting random variables to be equal in distribution to their source is insufficient for ensuring that a simulation will have the same scheduling performance characteristics as that source.

To overcome this limitation, we have evaluated all the scheduling policies in our experiment under four parameterizations that comprise the corner points of a quadrangle. We argue that by appropriately mixing the performance characteristics at each of these four points, the scheduling performance of any workload on any system can be approximated. The four parameterizations we use are: task servicing is correlated between servers and tasks are homogeneous within jobs, task servicing is correlated between servers and tasks are heterogeneous within jobs, task servicing is uncorrelated between servers and tasks are heterogeneous within jobs, and task servicing is uncorrelated between servers and tasks are homogeneous within jobs. We summarize these parameterizations in Table III.

Homogeneous tasks indicate that the service time of one

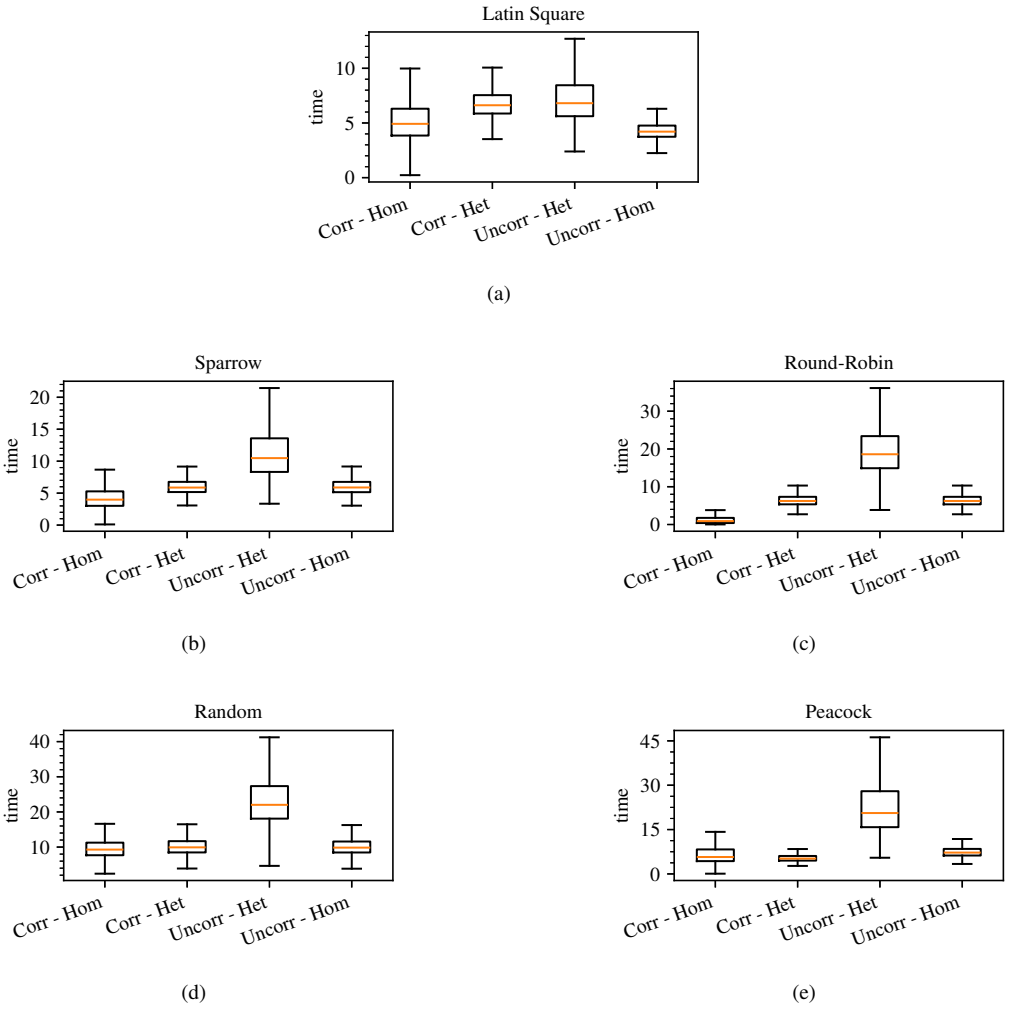

Fig. 5. Change in Job Response Time Distribution vs Change in Processing Model.

task within a job is *predictive* of the service time of all other tasks within a job, while heterogeneous tasks indicate that the service time of one task within a job *is not predictive* of the service time of other tasks within a job. To reproduce the Sparrow [6] simulation experiments, our task service times have been chosen according to an exponential distribution.

Under a homogeneous task parameterization, the service time of a task will be randomly generated from an exponential distribution with mean 1 for each job and assigned to every task within the job (every task in a job has the same size) and is how the simulation experiments were performed by Ousterhout et. al. [6]. Under a heterogeneous task parameterization, every task within a job will have an i.i.d. service time from an exponential distribution with mean 1. Correlated task servicing between servers indicates that the service time of a task on one server is *predictive* of the service time of the same task on a different server. Uncorrelated task servicing indicates that the service time of a task on one server is *not predictive* of the service time of the same task on a different server. Uncorrelated servicing also indicates the presence of burst interference that varies the task processing characteristics both between parallel servers and over time. In our experiments, we have implemented correlated servicing by setting the service time of each distinct task to be equal on every server. We have implemented uncorrelated servicing by setting the service time of each distinct task to be equal to i.i.d. exponentially distributed random variables with mean equal to the task service time at every server.

B. Results

To ensure a like-for-like comparison with the simulation results in [6], we have parameterized our latin square policy to divide the 100 tasks in each job into 50 parallel branches with 2 tasks in each branch and schedule them on 50 randomly selected pairs of servers. This ensures that our policy has been parameterized for an identical number of remote procedure calls, an identical server selection algorithm, and an identical number of replicated tasks or *probes* as in [6]. Additionally, we ensure unit average task latencies by setting $\mu = 1$ and $\nu = 1$. The simulated performance of Sparrow illustrated in Fig. 5 is then the same as the simulated performance in [6] rescaled by a unit of 100ms.

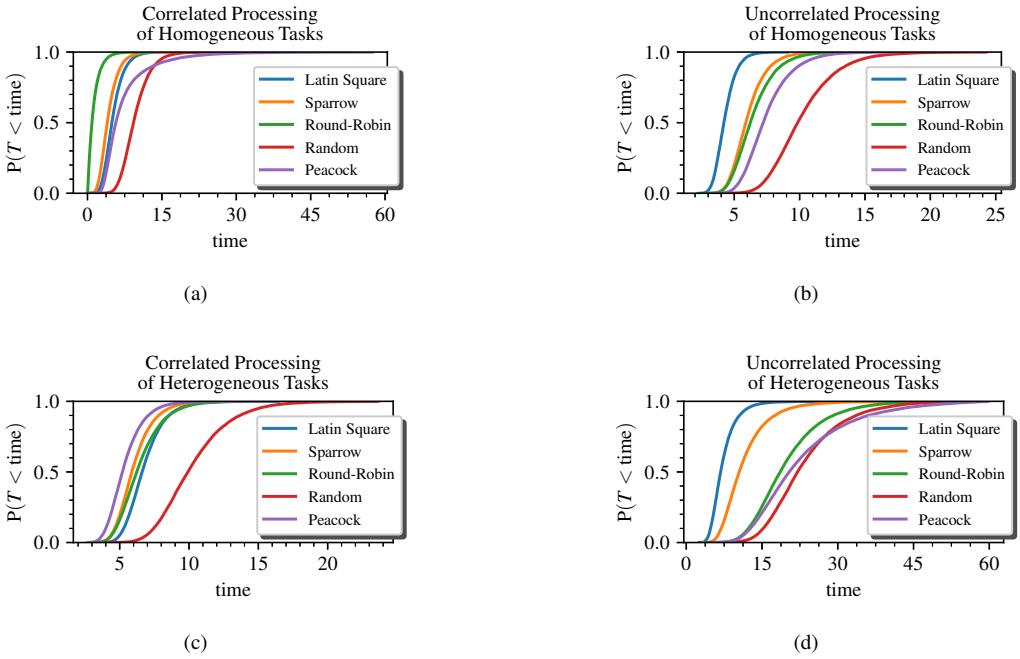

Fig. 6. Change in Job Response Time Distribution vs Change in Policy.

To simulate the schedules for each of our policies, we generate 20000 jobs to be scheduled on a cluster of 10000 parallel servers. The random job arrival times are all generated from the same seed so that for each processing time model the scheduling policies being compared process an identical sequence of jobs with identical arrival time spacings (i.e., the differences between schedules are *caused* by the choice of scheduling policy). Fig. 6 compares each of the scheduling policies we have evaluated against each other while varying the job servicing characteristics between each of the four kinds in Table III. This comparison illustrates that our latin square scheduling policy produces significantly shorter schedules than the other policies under uncorrelated servicing characteristics, and therefore, in systems that are suffering from random burst interference. We consider a schedule to be shorter when the job response time distribution has shorter delays at all the quantiles. The comparison also illustrates that under correlated servicing characteristics, simple round-robin scheduling has similar, if not better performance than much more sophisticated strategies like Peacock and Sparrow, even for jobs with highly heterogeneous task service times.

Fig. 5 illustrates how the performance of each scheduling policy changes as the job service time characteristics change between each of the four kinds in Table III. This chart demonstrates that our latin square policy produces extremely consistent and predictable schedules under a wide variety of circumstances. Of particular interest, is the performance variation of the Peacock scheduling policy, given in Fig. 5, between heterogeneous workloads that are being serviced by correlated (no burst interference) servers, and homogeneous workloads that are being serviced by uncorrelated (burst interference) servers. The performance improvements demonstrated by Peacock over Sparrow [7] were determined by trace-driven simulations with data taken from Google [33] as its input source. While driving a simulation with historical traces guarantees that the service time characteristics will have the right degree of task heterogeneity, simulations of this nature will always be for the *correlated* servicing of tasks. As the performance of Peacock is not symmetric between correlated servicing of heterogeneous tasks and uncorrelated servicing of homogeneous tasks, it indicates that Peacock should perform better in trace-driven simulations than it might in practice. Fig. 6 highlights that this difference may be large enough to be the difference between Peacock performing *better* than the state-of-the-art when implemented and deployed at scale, and performing *worse* than the state-of-the-art when implemented and deployed at scale.

VII. Discussion

A. Throughput Invariance

Since we have ensured that our average task times are all unit, the long-term average throughput of our system is the same for all four of our scheduling policies even though there is a significant amount of variance between job sizes, i.e., the expected value of the number of jobs that have completed by time t, denoted $E[N(t)]$, is asymptotically the same for all the policies we have evaluated. For instance, Table V lists the total realized simulation time (the completion time of the last job minus the starting time of the first job) for all four scheduling policies being compared. While there is some difference in total simulation time between each of the servicing characteristics, there is less than a 5% maximum

TABLE V
TOTAL SIMULATION TIME — 20000 JOBS

	Corr/Hom	Corr/Het	Uncorr/Het	Uncorr/Hom
Latin Square	415	415	448	411
Sparrow	404	402	440	417
Round-Robin	406	415	460	415
Random	417	417	450	419
Peacock	417	413	458	415

difference between policies. This is because we know that as long as the offered traffic, λ, is less than the aggregate servicing capacity of the system, the carried traffic will be equal to the offered traffic. Therefore, average throughput will be kept identical by all scheduling policies that do not degrade system capacity below the level of offered traffic. We call scheduling policies that satisfy this condition *throughput invariant scheduling policies*.

Throughput invariance is an important precondition to a fair comparison between two different scheduling policies, as comparing two policies under different throughput conditions makes it difficult to distinguish between a difference in latency introduced by a change in scheduling policy and a difference in latency introduced by a change in job throughput or system loading. As can be seen in Fig. 6 the difference between schedules produced by different policies on simulated systems with identical throughputs can be dramatically different, this implies that there is not necessarily a trade-off between the global end-to-end job latencies and average system throughput as is the common tuning advice for queue-based systems like Apache Kafka [34].

B. Considerations for Time-Varying Systems

Empirical workload traces taken by Google [31] and Facebook [32] are unlikely to exhibit perfectly stationary time-invariant characteristics. To begin with, most empirical workload traces are taken over long periods of time so that average performance trends can be identified. Over such long intervals of time, the warehouse-scale computers that these workloads are executing on will likely undergo hardware upgrade cycles that will both increase aggregate servicing capacity and reduce average job latencies over time. Other time-based trends, in particular, periodic trends like time-of-day based activities, are also hidden when aggregated measurements are presented using probability distributions. Periodic trends are particularly problematic, as there is no way to distinguish the difference between sinusoidal trend lines and uniformly distributed random noise when examining the final distribution.

This inability to distinguish between what is a trend line and what is random motion in empirical traces is what makes it difficult to design and fit a random service time model for our simulated evaluations. We argue that for most systems and workloads, there will be a mixture of both deterministic and random characteristics spread out over time. This means that for some intervals of time, the frequency of node degradation caused by burst interference may be very low, while in other intervals of time the burst interference frequency may be much higher. Although the exact weighting and distribution of this mixture of performance characteristics is unknown, we can make assumptions to examine how different scheduling policies may perform under differently weighted mixtures. Fig. 7 illustrates the performance of the scheduling policies we have evaluated under a simple uniform mixture of our four task service time parameterizations. It is notable that in addition to producing extremely consistent performance over a wide variety of operating conditions, our latin square scheduling policy also produces the shortest schedule under these uniformly mixed time-varying conditions.

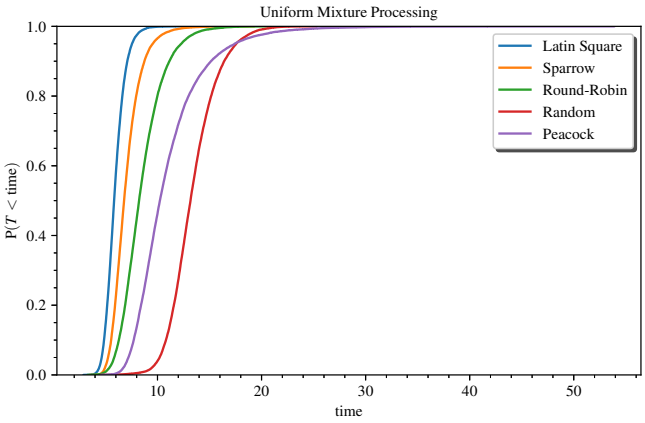

Fig. 7. Uniform Mixture Processing – Schedules

VIII. CONCLUSIONS AND FUTURE WORK

Most of the job scheduling literature we have examined is focused on improving scheduling performance by reducing waiting times stochastically, or reducing service times deterministically. For instance, the power-of-k load balancing [5] technique applied by Sparrow [6], or the combinatorial optimization techniques applied by SARS [35]. Our latin square scheduling policy is an example of a class of scheduling algorithms that are focused on reducing service times stochastically. Early examples of scheduling algorithms that have attempted this mostly revolve around the application of redundant or speculative execution. Examples include [8], [11], and [12]. More recently, there has been an upwelling of interest in applying much more sophisticated combinatorial and algebraic techniques than simple redundancy to stochastically reduce the service times of machine learning and other distributed matrix arithmetic-based workloads [15]–[17].

In designing our latin square scheduling policy, we argue that these combinatorial and algebraic scheduling techniques can have far broader use cases than simply machine learning and distributed matrix arithmetic applications. Through a combination of the small-scale analysis in section V and simulated experiments in section VI, we demonstrate that, under suitably unpredictable workloads, our latin square scheduling policy can produce significantly shorter schedules than other common

schemes like the Peacock [7], Sparrow [6], round-robin, and random scheduling policies. Our simulation results, in particular, highlight that the performance of our latin square scheduling policy improves as parallel server processing capabilities decorrelate (as they would under random burst interference assumptions).

A. Future Work

The effect block designs have on job schedules under uncorrelated servicing conditions is unlikely to be limited to latin squares. Other approaches, like Youden designs [4, Ch. 12], likely perform similarly. Additionally, as block designs emphasize the modulation of *service times* rather than *waiting times*, they can augment techniques that only modulate *waiting times*, e.g., power-of-k based approaches like [6], [9], [10]. Therefore, we can develop combination algorithms that simultaneously exploit power-of-k and block designs to get the best performance under both correlated and uncorrelated conditions.

REFERENCES

[1] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines*, 3rd ed., M. Martonosi, Ed. Springer Nature, 2022.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, p. 107–113, jan 2008.

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

[4] A. Dean, D. Voss, and D. Draguljić, *Design and Analysis of Experiments*, 2nd ed. Springer International Publishing, 2017.

[5] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, pp. 1094–1104, Oct 2001.

[6] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. New York, NY, USA: Association for Computing Machinery, 2013, p. 69–84.

[7] M. Khelghatdoust and V. Gramoli, "Peacock: Probe-Based Scheduling of Jobs by Rotating Between Elastic Queues," in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Cham: Springer International Publishing, 2018, pp. 178–191.

[8] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized speculation-aware cluster scheduling at scale," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. New York, NY, USA: Association for Computing Machinery, 2015, p. 379–392.

[9] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: hybrid datacenter scheduling," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USA: USENIX Association, 2015, p. 499–510.

[10] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, "Job-aware scheduling in eagle: Divide and stick to your probes," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*. New York, NY, USA: Association for Computing Machinery, 2016, p. 497–509.

[11] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. USA: USENIX Association, 2013, p. 185–198.

[12] L. Lei, T. Wo, and C. Hu, "CREST: Towards Fast Speculation of Straggler Tasks in MapReduce," in *IEEE 8th International Conference on e-Business Engineering*, Oct 2011, pp. 311–316.

[13] T.-D. Phan, S. Ibrahim, A. C. Zhou, G. Aupy, and G. Antoniu, "Energy-Driven Straggler Mitigation in MapReduce," in *Euro-Par 2017: Parallel Processing*, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, Eds. Cham: Springer International Publishing, 2017, pp. 385–398.

[14] J. S. Ng *et al.*, "A comprehensive survey on coded distributed computing: Fundamentals, challenges, and networking applications," *IEEE Communications Surveys and Tutorials*, vol. 23, no. 3, pp. 1800–1837, 2021.

[15] 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. 2018.

[16] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded fourier transform," in *2017 55th Annual Allerton Conference on Communication, Control, and Computing*. IEEE Press, 2017, p. 494–501.

[17] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient Coding: Avoiding Stragglers in Distributed Learning," in *Proceedings of the 34th International Conference on Machine Learning*. PMLR, Jul. 2017, pp. 3368–3376.

[18] C. Wang and G. Pottie, "Dynamic channel resource allocation in frequency hopped wireless communication systems," in *Proceedings of 1994 IEEE International Symposium on Information Theory*, Jun. 1994, pp. 229–.

[19] Kevin Exton. Stochastic scheduling simulation. Accessed April 02, 2024. [Online]. Available: https://github.com/kcexn/simulation

[20] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*. Paris France: ACM, Apr. 2010, pp. 265–278.

[21] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytia, "Reducing Latency via Redundant Requests: Exact Analysis," in *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '15. New York, NY, USA: Association for Computing Machinery, Jun. 2015, pp. 347–360.

[22] M. Shrahili and M. Kayid, "Modeling extreme value data with an upside down bathtub-shaped failure rate model," *Open Physics*, vol. 20, no. 1, pp. 484–492, 2022.

[23] M. Jette, "Performance characteristics of gang scheduling in multiprogrammed environments," in *SC '97: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, Nov 1997.

[24] L. Kleinrock, *Queuing Systems Volume II: Computer Applications*. USA: John Wiley and Sons, 1976, vol. 2.

[25] A. Rényi, "On the theory of order statistics," *Acta Mathematica Academiae Scientiarum Hungaricae*, vol. 4, no. 3, pp. 191–231, Sep. 1953.

[26] F. Baccelli, A. M. Makowski, and A. Shwartz, "The fork-join queue and related systems with synchronization constraints: stochastic ordering and computable bounds," *Advances in Applied Probability*, vol. 21, no. 3, pp. 629–660, Sep. 1989.

[27] M. Zukerman, "Introduction to Queueing Theory and Stochastic Teletraffic Models," Jun. 2023, arXiv:1307.2968 [cs, math]. [Online]. Available: http://arxiv.org/abs/1307.2968

[28] Apache OpenWhisk. Apache Software Foundation. Accessed December 19, 2023. [Online]. Available: https://openwhisk.apache.org

[29] IP Virtual Server. Accessed February 19, 2024. [Online]. Available: http://www.linuxvirtualserver.org/software/ipvs.html

[30] The Kubernetes Authors. Kubernetes. Accessed December 19, 2023. [Online]. Available: https://kubernetes.io/

[31] M. Tirmazi *et al.*, "Borg: The next generation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020.

[32] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads," *Proc. VLDB Endow.*, vol. 5, no. 12, p. 1802–1813, aug 2012.

[33] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. New York, NY, USA: Association for Computing Machinery, 2012.

[34] B. Ibryam. Fine-tune Kafka performance with the Kafka optimization theorem. Accessed April 1, 2024. [Online]. Available: https://developers.redhat.com/articles/2022/05/03/fine-tune-kafka-performance-kafka-optimization-theorem

[35] X. Wang, S. Xu, and Y. Zhao, "SARS: Towards minimizing average Coflow Completion Time in MapReduce systems," *Computer Networks*, vol. 247, p. 110429, Jun. 2024.