A Model for 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 Read
School of Computing and Information Systems
The University of Melbourne
Melbourne, Australia
Email: maria.read@unimelb.edu.au

***Abstract*—Modern data processing workloads often have highly unpredictable end-to-end latency characteristics that are caused by heterogeneity, time-variation, and parallelized processing. The increase in unpredictability is in part attributable to job "straggling", and is symptomatic of a new class of stochastic scheduling challenges that will degrade the performance of current and future applications at scale. While the job scheduling literature for data processing frameworks is rich with ideas; there is little coordination between research groups on methodology and presentation, stunting the ability for designers to survey a collection of results and draw generalizable conclusions about good design patterns. We introduce an abstract system model for data processing on warehouse scale machines that aids in eliminating ambiguity in the job scheduling research by categorizing schedulers based on where they act on the system in the job processing data pathway. Furthermore, we demonstrate that although the scheduling problem is NP-Hard in the general case, it is still possible to derive scheduler design principles using bounds and asymptotics.**

***Index Terms*—Job Scheduling, Distributed Data Processing, Warehouse-Scale Computers**

I. Introduction

Cloud computing and the resource-as-a-service model it supports has seen an explosive increase in popularity since its inception. From 2010 to 2022, the cloud computing market grew from a mite, $\$40.7$ billion [1], to a mint $\$446.4$ billion [2]. This growth has seen cloud computing, and the warehouse-scale [3] infrastructure that drives it, become ubiquitous across a wide variety of applications such as data processing [4]–[6] and machine learning [7].

It was noticed from early on [8], that to support services that are interactive and user-facing, computational workloads need to trend towards applications that have shorter end-to-end delays or latencies. For single server systems (and to a lesser extent, multiserver systems), the precise latency characteristics of a wide variety of scheduling policies have been comprehensively studied in the context of queuing theory [9]–[11]. However, warehouse-scale computers are not as well understood since they have many additional dimensions of control required of them [3, Ch. 2], e.g., the dynamic allocation of resources and the sequencing and assignment of arriving workloads to those resources. The sheer size of these machines, which can be composed of as many as $50\,000$ to $80\,000$ physical servers [12] and draw in excess of 100MW of power [3, Ch. 4], means that new challenges have emerged as a direct consequence of scale which must be managed by the control plane of these warehouse-scale machines. One such example of an emergent complexity to job scheduling at scale is the discovery of delay skew, or job "straggling", caused by instantaneous variations in parallel node performance [13].

A first consequence of this "straggling" phenomenon is that application performance appears to become more random as the computer grows in size. From the point of view of application developers and system administrators, developing a strong intuition for the nature of a *random computer system* now becomes a necessary condition to design and maintain high-performing applications at scale. This is a very challenging task as pertinent phenomena (such as job straggling) only become measurable once applications or systems cross certain size thresholds, rendering small-scale experiments, prototypes, and test benches irrelevant due to the inability to accurately capture the random characteristics of a large-scale machine. Furthermore, the traditional intuitions, deeply embedded in highly skilled and experienced designers, to focus on small-scale metrics such as I/O performance, cache performance, number of requests, round-trip times etc., are of decreasing relevance as applications increase in scale because the number of extremely rare and highly disruptive events grow to non-negligible quantities as a consequence of the law of large numbers. For instance, an analysis by Garraghan et al. [14] argues that as few as 5% of straggling tasks in a system degrades the performance of as many as 50% of all jobs being processed.

Consider an application that uses an in-memory database, like Redis [15], to cache data. Cache misses at small scales will fall back to a co-located disk, or perhaps a networked file system with disks that are physically located nearby. The maximum request latency will be on the order of magnitude of 5-10ms worse than the minimum request latency [16, Ch. 2], and the variance in application latency will be determined by the frequency of cache misses. At larger scales, a cache miss may be able to fall back to a nearby disk, but it also may require network requests to external services to retrieve data. In turn, these external services may have their own caches that may require additional network requests to yet more external services to retrieve data. For incredibly large applications, i.e.,

ones operating at a global scale, maximum request latency may be many seconds worse than minimum latency, and variation in latency will be a function of the cost and frequency of cache misses over all the interconnected services required to process a request. Optimizing the cache performance of all these interconnected services may reduce variance in request latency, but unless a *guarantee* can be made of 0 cache misses somewhere in the data pathway, variance in request latencies *must increase* as an application scales.

A. Key Contributions

Unlike single server systems or even small-scale data centers, the prohibitive cost of acquiring a warehouse-scale computer limits the amount of knowledge that can be acquired through small-scale experimentation. As such, many researchers and designers have been turning towards mathematical models to guide their design decisions [17]–[19]. In the domain of job scheduling, much of this thinking has been focused on improving the delay characteristics of parallelizable data processing workloads for frameworks like MapReduce [4]. Although all of this work ostensibly focuses on the problem of job scheduling on warehouse-scale computers, there is a distinct lack of coordination between authors regarding the precise method in which to justify their design decisions. This has resulted in a great diversity of models, measurements, charts, language, and experiments which make it extraordinarily difficult to develop a precise understanding of what a warehouse-scale computer is, as well as compare different approaches to each other on a like-for-like basis. Taking one step towards resolving this matter, our contribution is in two parts:

1) Novel Modeling: We design an extensible model for warehouse-scale machines that defines abstractions for some of the most important design and operational features of a computer. As the focus of our research is predominantly job scheduling, we have defined only the features of a computer that are necessary to study job schedules. However, the model defines the available operations for data transfer, storage, and processing that are common to any computing problem.

We describe our model in section III. Here, we define important ideas like the job schedule, and computational input/output relationships. The objective of our model is to provide a simple, extensible, and highly general framework that can be used both as a reference for implementing simulation tools and as a means to eliminate ambiguity in the research dialogue surrounding warehouse-scale job scheduling. We have developed our own simulation tool for designing scheduling algorithms for warehouse-scale applications, which is available from [20].

2) Design Principles: We use our model to derive general principles that are to be kept in mind when designing systems and applications for very large scales. We call these our warehouse-scale job scheduling design principles. Three of these principles have been enumerated in section IV. Our list of design principles is far from comprehensive and is mostly composed of mathematical theorems that have been derived in the context of queuing theory, reliability theory, and renewal theory which we have translated to the context of warehouse-scale computers. The design principles we present are:

1) The Minimax Design Principle
2) The Principle of Capacity Constraints and Throughput Invariance
3) Gang Scheduling and the Processor Sharing Bound

II. Background and Related Work

Job scheduling for warehouse-scale applications, especially, data processing applications on frameworks like MapReduce [4], Spark [5], and Dryad [6] is an extensively studied problem. In the empirical and applied literature, many ideas and architectures have been proposed over the years [21]–[23]. Despite the common interest in data processing frameworks, many of these ideas take entirely different approaches with only a limited number of common features. In the theoretical domain, there is a similarly immense quantity of recent research [24]–[26]. Despite this work being more amenable to modeling adjustments and iteration, many of the publications in the theoretical literature can be clearly distinguished by their analytical themes depending on whether the authors are trying to solve the job scheduling problem using tools from information theory [19], scheduling theory [25], or queuing theory [26]. Authors subscribing to different ideas rarely cross over to neighboring domains, and results from different domains are difficult to directly compare, even though they are usually trying to solve the same problem.

The parallel nature of the job scheduling problem means that combinatorial optimization approaches suffer from being NP-Hard [27]. The separation of jobs into small tasks to be serviced in parallel leaves general formulations of the queuing problem similarly intractable [28]. These limitations of exact analysis may be partially responsible for splintering the job scheduling research. Much of the remaining difficulties seem to be attributable to differences in the way information is presented in the research, including differences in experimental methodology [17], [29] and the choice of units of measurement [8], [30].

Without a clear and consistent method to compare scheduling techniques from different domains of study, scheduler implementations are limited to making mutually exclusive decisions about the inclusion of scheduling algorithms based on subjective preferences rather than objective criteria. Furthermore, hybrid schedulers that multiplex many different scheduling algorithms based on workload and environmental factors have limited room for development since determining whether a combinatorial approach like Quincy [17] will outperform a queuing approach like Sparrow [8] under specific operating conditions is currently very difficult. There is also no evidence to suggest that different scheduling techniques act on system performance metrics in such a way that the simultaneous application of multiple algorithms will outperform the mutually exclusive selection of one over another.

The haphazard investigative approach towards job scheduling for warehouse-scale computers means that many researchers and designers must arrive independently at their own

guiding design principles. Tiny tasks [8], encoding techniques from information theory [19], coflows [25], and speculative execution [31], are among the many scheduling principles that have been proposed over the years. Each of these design principles will be effective under potentially different specific operating conditions determined by parameters such as workload characteristics, job arrival rate, system capacity, available data storage, and available network bandwidth. Without a common framework for comparison, it is difficult to make these nuanced considerations, leaving ideas open to rebuttal by demonstrating that under different operating conditions one or more of these principles can be invalidated [32].

To facilitate better communication between different research groups and forge clear paths towards more effective warehouse-scale job scheduling algorithms, work must be done towards developing a common framework for investigation and measurement so that different ideas and techniques can be compared on a like-for-like basis. Despite the volumes of pre-existing modeling work for related problems in queuing theory [11], reliability theory [33], scheduling theory [34], and point process theories such as renewal theory [35], we have found no work in the warehouse-scale job scheduling literature that is focused on developing clear and precise definitions for the important parameters of warehouse-scale computers, nor is there any work in the literature on developing and justifying robust methods of measurement based on objective requirements. For instance, latency measurements taken at low system utilization [18], or at high utilization for a short period of time [17], or by rescaling benchmark traces to accommodate a smaller scale deployment [21], all potentially introduce nuisance factors that skew the final measurements. Without any means to normalize the numerical values in this research to a common standard, it is very difficult to compare results between different authors.

III. An Abstract Interface Model for Simulation and Design

Differences in method and presentation across the job scheduling literature make it difficult to deduce precise qualitative and quantitative features of warehouse-scale computers from the research alone. Authors often differentiate designs based on architectures and abstractions that are intimately related to the hardware platforms and application frameworks that they are targeting. The high degree of specificity makes results derived in this way credible, the method, though, makes finding generalizable design principles by comparing results against each other terribly challenging. It is common, for instance, to divide designs between centralized [17] and decentralized [8] architectures. Also common, is to apply abstractions to model the communications network [36], job operational dependency structures [37], or job service time characteristics [19], or data storage architectures [24]. Although each of these distinctions appears very different, when taken together, it seems likely that there is a significant amount of overlap in the performance benefits provided by each.

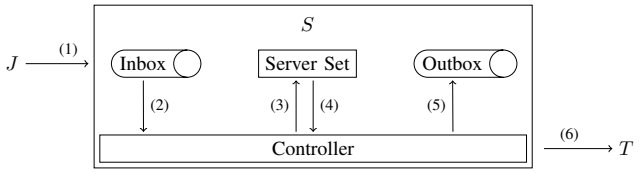


Fig. 1. Scheduled Computer Components. Typical flow of data is (1) – (6)

Our model is designed to relieve some of the difficulty that comes with attempts to directly compare different scheduling approaches in the research. We achieve this by defining warehouse-scale computers in terms of the strict unidirectional flow of data that is governed by a control plane or controller (illustrated in Fig. 1). Every scheduler in the literature must act on this system at one or more of the precise data flow interfaces in our model, allowing designers to quickly determine which approaches are likely to be mutually exclusive. For instance, the coflow-based approaches we reference and the pull-based approaches we reference both act on the system in the data flow pathway at interfaces (2) and (3) in Fig. 1.

To provide researchers and designers with flexibility, our model defines an abstract system architecture for warehouse-scale computers constructed from "abstract interface" style components that are connected together. In this spirit, for each component in our system we have defined only a minimum set of requirements that must be met to satisfy the functional needs of a warehouse-scale computer and leave many of the specific operational and implementation details open. This approach is useful when implementing scheduling algorithms for comparison in simulation, and we have implemented our own simulator (available from [20]) equipped with round-robin, random, and Sparrow [8] scheduling algorithms. Designers looking to integrate algorithms with our simulator (or any simulator built on our abstract architecture) must translate the abstractions used in their algorithms to precise actions and associated costs with respect to the data flow interfaces in the model.

A. Model, Component and Interface Overview

A warehouse-scale computer is made up of four logical components and six logical interfaces (Fig. 1). How each of these components and interfaces is implemented (no matter whether in software or in hardware) is left to the system designer. In particular, the components and interfaces may have either centralized or decentralized implementations: centralized component implementations mean that there are no networked communication costs for algorithms that are internal to the component and that the bandwidth for associated data flow interfaces may be defined by a single node, decentralized component implementations mean that internal algorithms in the component may suffer from networked communication costs and that the bandwidth for associated data flow interfaces may be spread out over many nodes. Each component in the model is defined in terms of a minimum number of necessary

TABLE I
A CONTROLLERS VIEW OF JOB SCHEDULING

Job ID	Inputs	Outputs	Arrival Time	Completion Time
01	$i_{1,1}, \ldots, i_{1,n}$	$o_{1,1}, \ldots, o_{1,n}$	0	
02	$i_{2,1}, \ldots, i_{2,m}$	$o_{2,1}, \ldots, o_{2,m}$	10	30
03	$i_{3,1}, \ldots, i_{3,k}$	$o_{3,1}, \ldots, o_{3,k}$	15	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
N	$i_{N,1}$ $i_{N,2} = o_{N,1}$ $i_{N,3} = o_{N,1}$ $i_{N,4} = o_{N,3}$ $i_{N,5} = o_{N,3}$ $i_{N,6} = o_{N,3}$ $i_{N,7} = (o_{N,2}, o_{N,4}, o_{N,5})$	$o_{N,1}, \ldots, o_{N,7}$	t	$t + \Delta t$

conditions that must be met for the warehouse-scale computer to function properly.

1) Job Inbox: Inputs to the warehouse-scale computer are abstracted in terms of an arriving sequence of jobs, $J = \{j_1, \ldots, \}$ that must be processed. The controller is responsible for organizing the arriving input data and placing it via logical interface (1) into the job inbox. Mathematically, the job inbox is an unordered set, and data that is placed in the inbox is done via the set inclusion operation. Any abstractions (such as queuing) that are built on top of the set are implemented in the controller component, not the job inbox. The inbox has infinite memory, so removing data from the inbox is never necessary.

2) Server Set: Each server in the server set is an abstract representation of all the application resources (including hardware resources) that are needed to process data from the inbox. Interface (3) is used by the controller to select and assign data to a server to process and can be a "push" or a "pull" based interface. Interface (4) is used by the controller to retrieve outputs from servers after they have finished processing, this can also be a "push" or a "pull" based interface. Each server must internally define accountable resource costs (time, memory, or other hardware resources) for data processing.

3) Job Outbox: After data has been processed at a server, the controller is responsible for retrieving the outputs over interface (4), organizing and placing these outputs in the job outbox over interface (5). The job outbox is the final destination for data in the data processing pathway. Once an input job no longer has any more outputs that need to be placed in the outbox, the job has "cleared" the system. Similar to the job inbox, the outbox is an unordered set with data added to it via set inclusion. The outbox has infinite memory, so removing data from the outbox is never necessary.

4) Controller: The controller is responsible for moving arriving job data through to the job outbox via interfaces (1)-to-(5) strictly in this order (reverse movement is an illegal operation). The secondary function of the controller is to take and report relevant performance measurements, which are emitted on interface (6). We have defined job schedules in terms of a job's end-to-end latency or delay, which is emitted from interface (6). Future extensions may wish to define new metrics that are emitted on this interface.

TABLE II
TWO VALID SEQUENCINGS OF THE JOB IN FIG. 2

Index	0	1	2	3	4	5	6
Sequence 1	o_1	o_2	o_3	o_4	o_5	o_6	o_7
Sequence 2	o_1	o_3	o_2	o_6	o_4	o_5	o_7

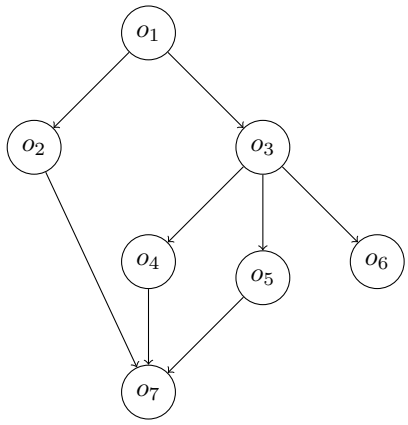


Fig. 2. Directed Acyclic Graph (DAG) of a Job with 7 Operations.

B. Jobs, Operations, and the Input/Output Relationship

Data processing jobs are composed of many operations that may have dependencies between each other. Individual operations are chunked to a degree of granularity appropriate to the scale of the job. For instance, for large-scale map-reduce jobs, an operation might be applying the map function to an individual input data partition rather than the individual arithmetic operations that are performed by the underlying CPU. Each operation has a relationship between inputs in the inbox to outputs in the outbox, and when operations have dependencies between each other, the outputs of one operation may be equal to the inputs of another. The flow of data through a job is then typically studied using a directed acyclic graph (DAG) such that every valid sequencing of operations is also a valid topological ordering of the graph. For instance, Table II lists two valid sequencings of the operations illustrated in Fig. 2.

The inputs and outputs of operations in our system can be tracked by the controller using a table or ledger where each operation in a job has associated inputs, outputs, and some additional metadata to measure latency and track job state, e.g., job 1 has inputs $i_{1,1}, \ldots, i_{1,n}$, outputs $o_{1,1}, \ldots, o_{1,n}$, an arrival time, and a completion time. The dependency structure of the operations means that some of the inputs are equal to some of the outputs, i.e., input 2 can be equal to output 1. For instance, the N'th entry in Table I illustrates the entry that will exist in the controller if the job listed in Fig. 2 completes.

For sequences of jobs, J, we use a job-shop scheduling model to characterize the control flow of work within the computer system. In the job-shop model, sequences of jobs, J, must be assigned servers to execute the operations within

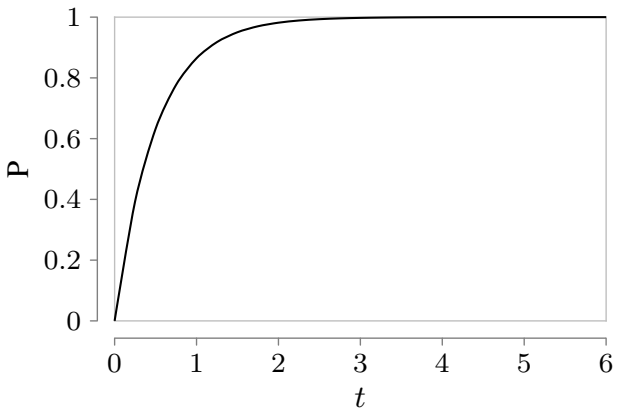


Fig. 3. The Schedule: $\mathrm{P}\,(T < t)$.

each job in a sequence that satisfies the dependencies between operations. Since computers are general purpose machines capable of handling any computational operation (instead of the alternative scenario where there is a collection of specialized machines that are only capable of handling one type of operation), the system controller is allowed to assign any number of operations from a single job to the same server. For systems with a single server, this property of assigning multiple operations from a single job to the same server in a sequence is called recirculation. In the distributed multiserver case, operations can be assigned to any server from a bank of parallel servers, and since all the servers are general purpose, operations can be sequenced and assigned to any subset of this bank multiple times. This extension to the job-shop with recirculation is known as the *flexible job-shop with recirculation* [27].

There is an important difference between the traditional flexible job-shop and the one we use in our model. In the traditional approach, the inputs of each operation must be *consumed* in order to *produce* an output. This is a natural way of thinking about production lines or factories; e.g., raw materials like iron must be consumed to produce steel. However, computer systems can *copy* instructions and data when assigning work to servers, so they do *not* consume inputs in the same way that factories do. This is reflected in our model by the way the controller can continue to track inputs even after operations have been assigned to servers, and even after the operational outputs have been produced. Since many distributed computer systems use distributed ledger technologies and model their inputs and outputs as append-only event streams, we can think of the inbox and outbox of our system as distributed ledgers. In this way, the controller keeps a copy of all the operational inputs and outputs of scheduled jobs in transaction receipts, and can reuse data whenever the scheduling policy dictates that data should be reused. For instance, in Google's MapReduce [4], slow-running tasks are relaunched by the controller towards the end of a job to reduce the tail latencies of job execution.

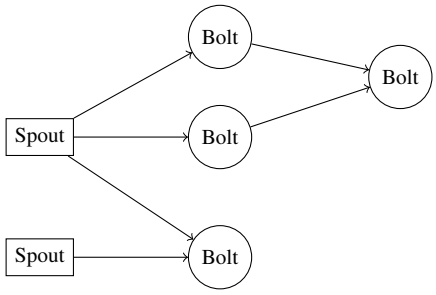


Fig. 4. Stream Processing Graph with Storm [38]

C. Example: Stream Processing with Storm

Consider how we might overlay a typical Storm stream topology comprised of Spouts and Bolts, as illustrated in Fig. 4, onto our model. Jobs arrive at the computer in a sequence, $J = \{j_1, \ldots\}$, where each job, j_i, is represented by data that has been emitted by a data source. The first action that a computer must take in our model is to assign each job a collection of inputs, outputs, and associated relations that determines the data processing dependencies. *Streams* provide the abstraction representing dependency relationships between inputs and outputs in Storm. Job data, then, is constructed from a sequence of tuples that are emitted as outputs by Spouts and Bolts, intermediate tuples that are passed between Bolts over links are outputs that are recirculated by the computer to serve as an input to the next relation in the graph.

Storm Spouts are distributed members of the control plane or controller component: they act on interface (1) of a computer by taking arriving jobs and including them into the inbox, they also act on interfaces (2) and (3) by sending the first inputs of each job to one or more Bolts. When a Bolt finishes processing an input tuple, it will emit an output tuple to the controller over interface (4). A message broker often serves as the component in the Storm controller that receives these outputs, determines if the job has "cleared" the system by cross-referencing the accumulated outputs with the stream definition over interface (5), and if not, recirculates the outputs to serve as inputs to the next Bolt in the stream over interface (3).

D. Scheduling Jobs

Although the flexible job-shop scheduling problem is well understood to be NP-Hard [27], we can exploit the *law of large numbers* to study a simpler aggregate scheduling problem. To study the aggregate performance of the system, we can measure the end-to-end execution latencies of many jobs as they are emitted by the scheduler and plot their cumulative frequencies to construct the distribution illustrated in Fig. 3. We refer to this distribution of scheduler outputs, T, as *the schedule*. Since the schedule is inclusive of all waiting and service time delays, we are not making any assumptions of time-invariance or homogeneity, this is simply the empirical distribution of end-to-end latencies. This approach is advantageous since it seems to be the natural choice for authors

when investigating scheduling performance [39], although alternative visualizations like the box-and-whisker chart are sometimes used to illustrate the distribution [8]. For jobs consisting of parallel operations, the schedule comprises the superposition of parallel random processes, as we illustrate in Fig. 3.

The end-to-end latency of each job is measured from the moment the job arrives at the inbox to the first moment that a job has no more work to be done (the job has no more outputs to be placed in the outbox). End-to-end job latencies are therefore inclusive of all waiting and service times, this latency is also referred to as the response [8], or delay time of a job [9]. In the simplest case, this system is a feed-forward control system that takes a sequence of jobs, J, as its input and produces a sequence of latencies as its output in much the same way as a queuing system. For traditional queuing problems, the output, T, can be considered to be a random variable with an empirical distribution function $F(\cdot)$ that represents the delay times of the jobs in J. While this delay time distribution is quite well understood for a wide variety of queuing problem formulations e.g., $M/M/1$, $M/G/1$, $M/M/k$, $M/G/k$ etc, our model allows for formulations that are not considered standard within the domain of queuing theory. For instance, the controller in our model can divide a job up into smaller operations over interface (1). Additionally, as jobs are placed in the inbox with no particular ordering structure, the controller must make ordering and sequencing decisions of jobs and operations that set it apart from traditional queuing problems. Finally, the theoretical model has infinite memory so that the inbox and outbox can store an infinite amount of data which the controller can reference at any time. This facilitates special operations like operation repetition and data replication that traditional queues as well as many scheduling algorithms do not take into consideration.

IV. The Scheduling Design Principles

Our model is designed to accommodate abstractions that aid designers to accurately, and efficiently, reason about very large-scale systems and applications. We demonstrate this by presenting three design principles for job scheduling on warehouse-scale computers. Our design principles emphasize the role of proven results drawn from the deep pool of ideas in closely related fields of study, especially, queuing theory, and renewal theory.

A. The Minimax Design Principle

The minimax design principle provides designers with guidance on *when* and *where* they should prioritize optimization during the design and implementation of systems or applications. Roughly speaking, it says that to achieve the best possible overall performance, the best-case performance should be statically optimized by the system or application design, and the worst-case performance should be dynamically optimized by the system or application scheduler. In essence, the best-case performance of a system or application is the performance that the system or application achieves under ideal conditions, so it should be optimized as much as possible in the design and implementation phases. Conversely, the worst-case performance of a system or application comprises the worst outcomes achieved under "real-world" or dynamic conditions. Usually, this has little to do with system or application design and more to do with outside factors like system faults or resource contention, and so it must be managed by the system or application scheduler.

Principle 1 (The Minimax Schedule Design Principle)**.** *Recall that a schedule, T, is a non-negative random variable with an empirical distribution function $F(\cdot)$. A minimax schedule design works in two phases:*

1) *Optimize the schedule so that the left-hand side of the support (the minimum value that can be taken by T) of $F(\cdot)$ is as close to 0 as possible.*
2) *Design a scheduling policy that minimizes or reduces the frequency of all outcomes, t in the support of $F(\cdot)$, chosen such that $1 - F(t) \leq p$ for an arbitrarily chosen $p \in (0, 1)$.*

To justify this design principle, we need to define a *stochastic ordering* of possible schedules and understand a sufficient condition for two job schedules to be *stochastically ordered*.

1) Stochastic Order Comparison of Job Schedules: Two schedules, T_1, and T_2, are *stochastically ordered* such that T_1 precedes or is *shorter* than T_2, denoted $T_1 \preceq T_2$, if and only if, for every fixed time, $t > 0$, the proportion of jobs that have not yet finished by t in T_1 are less than or equal to the proportion of jobs that have not yet finished by t in T_2,

$$T_1 \preceq T_2 \iff \mathrm{P}(T_1 \geq t) \leq \mathrm{P}(T_2 \geq t), \text{ for all } t > 0 \quad (1)$$

two important first consequences for schedules that can be ordered in this way are:

- By definition, if T_1 is shorter than T_2, $T_1 \preceq T_2$, then for any fixed time $t > 0$, the proportion of jobs in T_1 that finish by time t will always be greater than the proportion of jobs in T_2 that finish by time t.

$$T_1 \preceq T_2 \iff \mathrm{P}(T_1 \leq t) \geq \mathrm{P}(T_2 \leq t) \text{ for all } t > 0$$

- By application of an expected value formula for non-negative random variables:

$$E[T] = \int_0^\infty 1 - F(t)\, dt = \int_0^\infty \mathrm{P}(T > t)\, dt$$

  it can be clearly seen that if T_1 is shorter than T_2, then the expected value of T_1 will be smaller than the expected value of T_2:

$$T_1 \preceq T_2 \implies E[T_1] \leq E[T_2]$$

2) A Sufficient Condition for the Ordering of Job Schedules: To complete our argument, we need to understand the sufficient condition for job schedules to be ordered given by [40, Thm 1.A.17]:

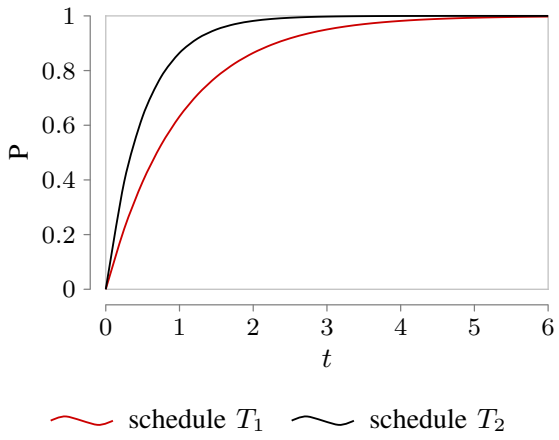


Fig. 5. Stochastic Ordering such that $T_2 \preceq T_1$

Theorem 1. *Let X be a random variable, and let ϕ_1 and ϕ_2 be two functions that satisfy:*

$$\phi_1(x) \leq \phi_2(x) \text{ for all } x \in \mathbb{R}.$$

then:

$$\phi_1(X) \preceq \phi_2(X)$$

for which special cases are when $\phi_1(X) = X$ or $\phi_2(X) = X$.

To interpret this theorem, we consider a schedule T_1 with an empirical distribution function $F(t)$. If we can design a scheduling policy to produce a schedule, T_2 with empirical distribution function $G(t)$, such that the following relationship holds:

$$F(\phi(t)) = G(t) \text{ such that } t \leq \phi(t) \text{ for all } t > 0$$

then it follows from Theorem 1 that $T_2 \preceq T_1$. Since there are no conditions placed on the function ϕ, especially, that it does not need to be linear, and it does not need to be continuous, we can construct a schedule T_2 from any given schedule T_1, by taking events that occur at arbitrary times $t > 0$ in T_1 and reducing their frequency (see Fig. 5 for example). From a practical point of view, it seems likely that the less frequently an event occurs, the easier it will be to further reduce the frequency.

B. The Principle of Capacity Constraints and Throughput Invariance

A common piece of folk advice among system administrators is that there is always a trade-off that must be made between application *throughput* and application *latency* [41]. The history and source of this advice is difficult to track down. However, if we examine the Pollaczek-Khinchine formula for the expected end-to-end delay of an $M/G/1$ queue [9, Ch. 16]:

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

and consider that the maximum average throughput or capacity of this system is given by μ, while the average latency is given by the reciprocal of the maximum average throughput plus an average waiting time, the advice that there is a trade-off between throughput and latency appears true for single-server systems. It seems plausible, then, that this advice is a rusted on remnant of older and simpler systems. We argue that this relationship between capacity, throughput, and latency becomes less important as the number of parallel servers in the system increases. This forms the basis of the second principle of design.

Principle 2 (The Principle of Capacity Constraints and Throughput Invariance)**.**
Given two scheduling policies that produce schedules T_1 and T_2 respectively. It is possible that these different schedules will have different aggregate service rates, or system capacities, denoted μ_1 and μ_2 respectively. Then:

1) *As long as the job arrival rate, λ, remains smaller than the $\min(\mu_1, \mu_2)$, the average throughput, or carried traffic of the system is equal to λ for both policies and the schedules are considered to be throughput invariant.*
2) *A scheduler is free to use any forms of operation repetition, data redundancy, or resource over-subscription rules available to it as long as the average job arrival rate, λ, remains below the system capacity, μ.*

To understand the second principle of design, we need to be precise about the definition of *throughput* and understand some of its characteristics.

1) Throughput, Latency, and the Renewal Theorem: The average number of jobs that a system can complete in each second is an intuitive notion of system throughput. A typical way of measuring system throughput is to count the number of jobs that are completed over a fixed interval of time $(0, t]$, and then divide by the time t. This definition can be made formal by using a *counting random process*. Let $N(t)$ be the total number of jobs that have finished processing in the time interval from $(0, t]$. Then for any choice of t, $N(t)$ will be a *random variable*, and the *family* of random variables[1] defined by $\{N(t), 0 < t < \infty\}$ is called a *counting random process* [33, Ch. 2]. If the intervals between completions, or the *inter-completion* times of a computer system are i.i.d., then the average or expected number of completions in this interval is called the *renewal function* [35, Ch. 4], $E[N(t)] = m(t)$. Subsequently, we know from the *renewal theorem* that the average system throughput is asymptotically equal to

$$\lim_{t\to\infty} \frac{m(t)}{t} = \frac{1}{E[\Delta C]}$$

where ΔC is the random variable associated to the time interval between completions. Since the average throughput given by the renewal theorem is an average over *all time*, it can not be changed by simple rearrangements, as is the common approach for proving results in the finite domain of scheduling

[1] set of distinct, but not necessarily independent random variables

theory. We can illustrate this by way of a finite time analogy that demonstrates how parallelizing a job changes the average delay while holding the average throughput fixed.

Consider the job schedule of 3 operations as illustrated in Fig. 6a. The completion times of operations o_1, o_2, o_3 are C_1, C_2, C_3 respectively. the average number of operations served per second is equal to:

$$\text{average operation throughput} = \frac{1}{E\left[\Delta C\right]} \tag{3}$$

We will assume that servers can only process one operation at a time, the operations must be processed sequentially and as the job has 3 operations, the average job throughput is found by taking the sum of the operational inter-completion times giving:

$$\begin{aligned}\text{average job throughput} &= \frac{1}{E\left[\sum_{i=1}^{3} \Delta C_i\right]} \\ &= \frac{1}{3E\left[\Delta C\right]}\end{aligned} \tag{4}$$

and the average job delay is simply the reciprocal of job throughput:

$$\text{average job delay} = 3E\left[\Delta C\right]$$

The introduction of parallel servers changes this relationship. Consider a different realization of the same job but in a system with 3 servers as illustrated in Fig. 6b. Since the time spent processing operations can overlap, the inter-completion times of operations is the difference between the time the next operation finishes on any server and the previous time an operation finished on any server (the notation, however, stays the same, ΔC). The schedule of operations in the multiserver scenario is no longer the sum of the individual operation processing times, but the *maximum* time taken to process all the operations in the job. The average operation and job throughputs, though, stay the same as in equations (3) and (4), while the average job delay becomes:

$$\begin{aligned}\text{average job delay} &= E\left[\max\left(C_1, C_2, C_3\right)\right] \\ &= E\left[C_3\right]\end{aligned}$$

which is not linearly related to capacity.

For jobs with batch arrivals of multiple operations, the average throughput will be equal to

$$\lim_{t\to\infty} \frac{m\left(t\right)}{t} = \frac{1}{E\left[\Delta C\right] \times \text{avg. no. operations in a job}}$$

where ΔC is the time difference between the completion times of subsequent operations, and we must multiply by the additional scaling factor of the average number of operations in a job.

An important distinction should be made between the average throughput of a system and the *capacity* of a system. Average throughput can be found by counting the number of jobs that are completed in a fixed window of time $(0, t]$ and dividing by t, as long as t is sufficiently large. The *capacity* of a system is the *maximum average throughput* of a system, i.e.:

$$\hat{\mu} = \limsup_{t\to\infty} \frac{m\left(t\right)}{t}$$

Average system throughput is then always a proportion of the system capacity and can be written down as

$$\lim_{t\to\infty} \frac{m\left(t\right)}{t} = \rho\hat{\mu}$$

where ρ is a *utilization* or *loading* factor. Under queuing assumptions, the system capacity is simply the maximum average arrival rate before the queue becomes unstable and is equal to the aggregate average servicing rate of the servers,

$$\hat{\mu} = \sum_i \mu_i$$

the average throughput, instead, is the *carried traffic* and is equal to the *offered traffic*, λ, as long as the queue is stable ($\lambda < \hat{\mu}$). When the average arrival rate is higher than the capacity, the average throughput is simply equal to the capacity. Since different scheduling policies can have different capacities, varying the scheduling policy will not vary the average throughput as long as the arrival rate remains below the capacity so that there is no throughput "clipping".

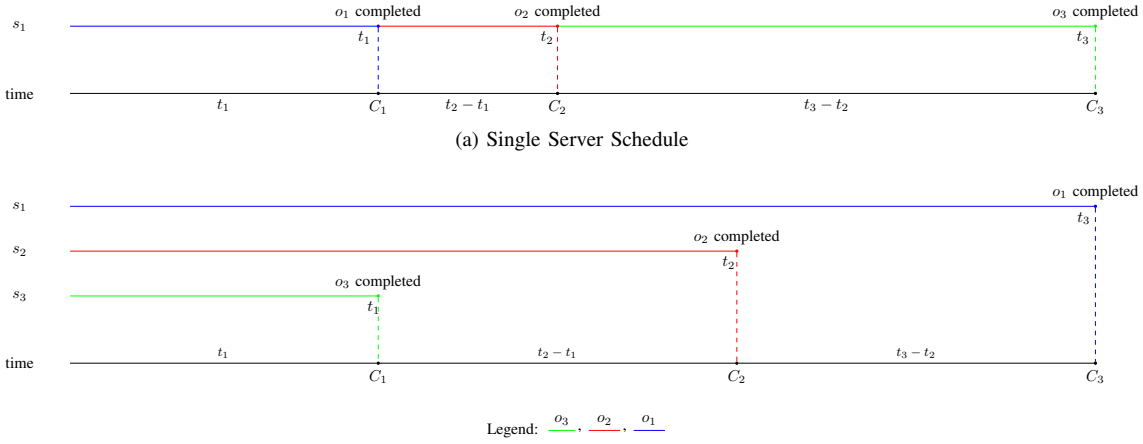
C. Gang Scheduling and the Processor Sharing Bound

Processor sharing is an old technique that when applied to single server queues has the effect of reducing the squared coefficient of variation in the expected delays of arriving jobs with long-tailed service time distributions. For single server queues with Poisson arrivals, the expected delay and waiting times for systems employing a processor-sharing queuing discipline are given by [11, Ch. 4]:

$$\begin{aligned}E\left[D\right] &= \frac{E\left[S\right]}{1-\rho} \\ E\left[W\right] &= \frac{\rho E\left[S\right]}{1-\rho} = E\left[W_{M/M/1}\right]\end{aligned}$$

where $E\left[S\right]$ is the average job service time, ρ is the system utilization or loading, and $E\left[W_{M/M/1}\right]$ is the expected wait time of an equivalent $M/M/1$ server with service rate $1/E\left[S\right]$ and arrival rate λ.

By first dividing the available processing time for each parallel server into a series of time slices that can be shared among jobs and then applying synchronization to ensure that parallel tasks belonging to the same job will always run in the same time slices, the multiserver system can be modeled as if it were a single server system. This synchronized scheduling policy for parallel processing is sometimes referred to as gang scheduling [42], and it enables the system to be studied as if it were an $M/G/1$ queue equipped with a processor-sharing queuing discipline. While the strict synchronization requirements of this model means that some time slices are wasted when jobs are composed of heterogeneously sized operations, it has the advantage of being easy to analyze which means we can use the performance of this system



(a) Single Server Schedule

(b) Multi-Server Schedule

Fig. 6. Schedule of a Single Job with 3 Operations.

as a common reference point when comparing alternative scheduling policies. This is the foundation for our final job scheduling principle.

Principle 3 (Gang Scheduling and the Processor Sharing Bound)**.**
By synchronizing operations and applying a gang scheduling policy across all k servers in a warehouse-scale computer, the expected delay will be equal to:

$$E[D] = \frac{E[S]}{1-\rho} \tag{5}$$

where $E[S]$ is the expected service time of jobs, and ρ is the system utilization or loading.

Since gang scheduling is a practically achievable design [42], equation (5) can be used as a worst-case expected delay bound for new designs.

V. Concluding Discussion

Our objective in developing this warehouse-scale computer model and corresponding job scheduling principles is to support the design, development, and improvement of scheduling and other resource management algorithms for warehouse scale systems and applications, including data processing frameworks like MapReduce [4], Spark [5], and Dryad [6]. To accommodate both future new ideas and the diversity of existing ideas, our model must be highly flexible. To achieve this, our model is designed using "abstract interface" style components which define warehouse-scale computing in terms of a minimum set of requirements that must be satisfied to process data. Scheduling and resource management algorithms are then defined in terms of how they act on the flow of data at each of the six interfaces in the model. The strict unidirectional (non-reversible) flow of data through a warehouse-scale computer means that there can never be any ambiguity in terms of what is meant by an algorithm that acts on interfaces (2) and (3). Eliminating ambiguity in the job scheduling research literature makes it feasible for researchers and designers to compare differing designs and architectures on a like-for-like basis, as well as determine whether specific algorithms may be mutually exclusive in terms of how they act on the data flow pathway.

The strict unidirectional flow of data in a warehouse-scale computer does not interfere with the operation of repetition or speculation-based approaches, like Dolly [31], as the infinite memory of the job inbox allows data to be copied from the inbox and assigned to the server set multiple times for processing. Interfaces (3) and (4), and the input/output relationship between data in the job outbox, and data in the job inbox, also provide opportunities for dynamic feedback-based control algorithms without violating the strict unidirectional flow of data through the system. To facilitate the comparison of different scheduling algorithms on a like-for-like basis, we have defined job latency, and job schedules, as a metric that can be emitted by the controller on interface (6). Future extensions to this abstract model may wish to define new metrics that can be used for comparing differing resource management approaches.

The warehouse-scale job scheduling principles we have presented are not an exhaustive list of design principles, rather, they form the beginnings of a much larger piece of work in attempting to enumerate provable principles of system and application design at scale. Given the generally NP-Hard nature of the warehouse-scale job scheduling problem, it is likely that job scheduling design principles will be discovered *first* by experimentation and practice, and confirmed *later* by mathematical and logical arguments. As such, we argue that the sensible place to begin a systematic search for design principles is in the design justifications of existing schedulers.

References

[1] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "The rise of RaaS: the resource-as-a-service cloud," *Commun. ACM*, vol. 57, no. 7, p. 76–84, jul 2014.

[2] L. Sustar, R. Kwon, and H. Joshi, "The public cloud market outlook, 2022 to 2026," Forrester, Tech. Rep., 2022. [Online]. Available:

https://www.forrester.com/report/the-public-cloud-market-outlook-2022-to-2026/RES178311
[3] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines*, ser. Synthesis Lectures on Computer Architecture. Cham: Springer International Publishing, 2019.
[4] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, p. 107–113, jan 2008.
[5] M. Zaharia, M. Chowdhury, T. Das *et al.*, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. USA: USENIX Association, 2012, p. 2.
[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, p. 59–72, mar 2007.
[7] Álvaro. López García, J. M. De Lucas, M. Antonacci *et al.*, "A cloud-based framework for machine learning workloads and applications," *IEEE Access*, vol. 8, pp. 18 681–18 692, 2020.
[8] 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*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 69–84.
[9] 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
[10] D. Gross and C. M. Harris, *Fundamentals of queueing theory*, 3rd ed. New York: Wiley, 1998, edition: 3rd ed. Publisher: Wiley.
[11] L. Kleinrock, *Queuing Systems Volume II: Computer Applications*. USA: John Wiley and Sons, 1976, vol. 2.
[12] P. Johnson. (2017) With the public clouds of amazon, microsoft and google, big data is the proverbial big deal. Accessed June 28'th, 2024. [Online]. Available: https://www.forbes.com/sites/johnsonpierr/2017/06/15/with-the-public-clouds-of-amazon-microsoft-and-google-big-data-is-the-proverbial-big-deal/
[13] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, p. 74–80, Feb 2013.
[14] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu, "Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters," *IEEE Transactions on Services Computing*, vol. 12, no. 1, pp. 91–104, Jan 2019.
[15] Redis. Accessed April 30, 2024. [Online]. Available: https://redis.io/
[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach*, 5th ed. Elsevier, 2012.
[17] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 261–276.
[18] Y. Zhao, C. Tian, J. Fan, T. Guan, X. Zhang, and C. Qiao, "Joint reducer placement and coflow bandwidth scheduling for computing clusters," *IEEE/ACM Trans. Netw.*, vol. 29, no. 1, pp. 438–451, Feb 2021.
[19] 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.
[20] K. Exton. Stochastic scheduling simulation. Accessed April 02, 2024. [Online]. Available: https://github.com/kcexn/simulation
[21] 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*, ser. USENIX ATC '15. USA: USENIX Association, 2015, p. 499–510.
[22] 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*, ser. SoCC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 497–509.
[23] 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*, ser. SIGCOMM '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 379–392.
[24] S. Li, M. A. Maddah-Ali, Q. Yu, and A. S. Avestimehr, "A fundamental tradeoff between computation and communication in distributed computing," *IEEE Trans. Inf. Theory*, vol. 64, no. 1, pp. 109–128, Jan. 2018.
[25] M. Chowdhury and I. Stoica, "Coflow: a networking abstraction for cluster applications," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XI. New York, NY, USA: Association for Computing Machinery, 2012, p. 31–36.
[26] 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.
[27] S. Dauzère-Pérès, J. Ding, L. Shen, and K. Tamssaouet, "The flexible job shop scheduling problem: A review," *European Journal of Operational Research*, vol. 314, no. 2, pp. 409–432, 2024.
[28] 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.
[29] E. Boutin, J. Ekanayake, W. Lin *et al.*, "Apollo: scalable and coordinated scheduling for cloud-scale computing," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 285–300.
[30] 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.
[31] 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*, ser. nsdi'13. USA: USENIX Association, 2013, p. 185–198.
[32] E. Totoni, S. R. Dulloor, and A. Roy, "A Case Against Tiny Tasks in Iterative Analytics," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: Association for Computing Machinery, May 2017, pp. 144–149.
[33] I. Gertsbakh, *Reliability Theory*. Berlin, Heidelberg: Springer, 2005.
[34] J. Y.-T. Leung, Ed., *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. New York: Chapman and Hall/CRC, Apr. 2004.
[35] D. R. Cox, *Renewal theory*, ser. Monographs on statistics and applied probability. London: Chapman and Hall, 1967, publisher: Chapman and Hall.
[36] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, p. 443–454, aug 2014.
[37] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–15.
[38] Streams. Apache Software Foundation. Accessed August 9, 2024. [Online]. Available: https://storm.apache.org/releases/2.6.3/Tutorial.html
[39] J. Xu, J. Wang, Q. Qi, H. Sun, J. Liao, and D. Yang, "Effective Scheduler for Distributed DNN Training Based on MapReduce and GPU Cluster," *Journal of Grid Computing*, vol. 19, no. 1, p. 8, Feb. 2021.
[40] M. Shaked and J. G. Shanthikumar, Eds., *Stochastic Orders*, ser. Springer Series in Statistics. New York, NY: Springer, 2007.
[41] 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
[42] M. Jette, "Performance characteristics of gang scheduling in multiprogrammed environments," in *SC '97: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, Nov 1997.