START: Straggler Prediction and Mitigation for Cloud Computing Environments using Encoder LSTM Networks

Shreshth Tuli[1], Sukhpal S. Gill[2], Peter Garraghan[3], Rajkumar Buyya[4], Giuliano Casale[1], and Nicholas R. Jennings[1,5]

Abstract—Modern large-scale computing systems distribute jobs into multiple smaller tasks which execute in parallel to accelerate job completion rates and reduce energy consumption. However, a common performance problem in such systems is dealing with straggler tasks that are slow running instances that increase the overall response time. Such tasks can significantly impact the system's Quality of Service (QoS) and the Service Level Agreements (SLA). To combat this issue, there is a need for automatic straggler detection and mitigation mechanisms that execute jobs without violating the SLA. Prior work typically builds reactive models that focus first on detection and then mitigation of straggler tasks, which leads to delays. Other works use prediction based proactive mechanisms, but ignore heterogeneous host or volatile task characteristics. In this paper, we propose a Straggler Prediction and Mitigation Technique (START) that is able to predict which tasks might be stragglers and dynamically adapt scheduling to achieve lower response times. Our technique analyzes all tasks and hosts based on compute and network resource consumption using an Encoder Long-Short-Term-Memory (LSTM) network. The output of this network is then used to predict and mitigate expected straggler tasks. This reduces the SLA violation rate and execution time without compromising QoS. Specifically, we use the CloudSim toolkit to simulate START in a cloud environment and compare it with state-of-the-art techniques (IGRU-SD, SGC, Dolly, GRASS, NearestFit and Wrangler) in terms of QoS parameters such as energy consumption, execution time, resource contention, CPU utilization and SLA violation rate. Experiments show that START reduces execution time, resource contention, energy and SLA violations by 13%, 11%, 16% and 19%, respectively, compared to the state-of-the-art approaches.

Index Terms—Straggler Prediction, Straggler Mitigation, Cloud Computing, Deep Learning, Surrogate Modelling.

1 Introduction

Emerging applications of Cloud Data-Centers (CDCs) in domains such as healthcare, agriculture, smart cities, weather forecasting and traffic management produce large volumes of data, which is transferred among different devices using various kinds of communication modes [1]. Due to this continuous increase in data volume and velocity, large-scale computing systems may be utilized [2]–[4], which exacerbates the need for scalable, automated scheduling and intelligent task placement methods. This work focuses on this problem by studying, in particular, strategies to mitigate straggler tasks. Stragglers are tasks within a job that take much longer to execute than other tasks and can cause a significant increase in response time due to the need for synchronizing the outputs of the tasks. Their presence can lead to the so-called Long Tail Problem [5].

More precisely, the Long Tail Problem occurs when the completion time of a particular job is significantly affected by a small number of straggler tasks in a negative way. Task stragglers can occur within any highly parallelized system that processes jobs consisting of multiple tasks. Google's MapReduce framework [6] or the Hadoop framework [7] are examples of such systems, where solutions for straggler prevention are common [1], [8], [9]. Both MapReduce and Hadoop allow for scalability of the system to vast clusters of commodity servers. The parallel execution of tasks increases the speed of execution and handles the failures automatically without human intervention following the principles of IBM's autonomic model [10], [11]. However, stragglers can still occur because of software/hardware faults as autonomic models are often slow in handling failures and can result in long down-times in resource-constrained devices [1]. These lead to unexpected delays in task execution due to resource unavailability or data loss and cause such tasks to hog resources which in non-preemptive execution leads to higher response times. Thus, efficient techniques are required to mitigate stragglers to prevent high response times and SLA violations. We now discuss what types of failures lead to stragglers tasks.

There are two types of failures that can occur during the execution of jobs: task failures and node failures. The former occurs when a specific task within a job fails, due to diverse sources of software and hardware faults [12]. The latter occurs when one of the resources of a specific node, which executes the job's task, fails [1]. This can be caused by a myriad of possible OS or hardware level faults. As an example of straggler mitigation techniques, MapReduce attempts to mitigate task failures by relaunching the task

- *S. Tuli, G. Casale and N. R. Jennings are with the [1]Department of Computing, Imperial College London, United Kingdom.*
- *N. R. Jennings is also with [5]Loughborough University, United Kingdom.*
- *S.S. Gill is with the [2]School of Electronic Engineering and Computer Science, Queen Mary University of London, United Kingdom.*
- *P. Garraghan is with the [3]School of Computing and Communications, Lancaster University, United Kingdom.*
- *R. Buyya is with the [4]Cloud Computing and Distributed Systems (CLOUDS) Laboratory, University of Melbourne, Australia.*
  E-mail: s.tuli20@imperial.ac.uk, s.s.gill@qmul.ac.uk, p.garraghan@lancaster.ac.uk, rbuyya@unimelb.edu.au, g.casale@imperial.ac.uk, n.r.jennings@lboro.ac.uk.

Manuscript received —; revised —.

once it fails [13]. In terms of a node failure, MapReduce re-executes all the tasks that were originally scheduled to be executed on that node. In terms of node failures, when the performance of a node degrades, either due to an OS or hardware fault or the node completely fails, a specific task's (straggler) execution time can be bloated, causing any other tasks that depend on it to wait for its completion [14]. At the job level, for the job to be considered complete, all the tasks comprising the job must finish. If a straggling task prevents other sibling tasks from successfully completing, the job will not be complete until all the straggler tasks are complete [15]. Furthermore, straggler tasks can keep other tasks dependent on their output waiting and hence consume additional resources, further impacting the performance of the computing system.

Stragglers not only affect performance but also deployment costs. Popular cloud service providers such as Amazon, Google, Netflix and Apple face the challenge of straggler tasks leading to delayed response or resource wastage. This requires avoidable scaling-up of the cloud infrastructures, which in turn increase the deployment costs [14], [16]. The high latency episodes called “tail-tolerant” or “latency-tail-tolerant”, also affect the performance of cloud services [17]. Latency tail-tolerant jobs reduce resource utilization and increase energy consumption. Characterization studies such as [1], [2], [5], [6], [10], [12], [18], show that resource contention is the main reason for stragglers, occurring when different jobs are waiting for shared resources. Different applications executing on different nodes may also contend for shared global resources [17].

Prior work [19], [20] focuses on solving the problem of straggler tasks by detecting and mitigating which tasks are stragglers only after the jobs are executed. Straggler mitigation refers to the prevention of any impact of straggler tasks on QoS or SLA. This not only requires continuous computation resources, but these monitoring tasks themselves can be so data-intensive that they can themselves lead to resource contention, delays and prevent scalability of the system [21]. However, modern technologies like deep learning allow us to build scalable models to not only detect, but predict beforehand, which tasks might be straggler and run mitigation algorithms to save time and improve QoS. Here, straggler prediction means the prediction of straggler tasks before execution. In particular, [22], [23] use deep learning based solutions to predict straggler tasks and efficiently manage them.

Deep learning based straggler prediction methods face large prediction errors due to two major problems. First, these models ignore the underlying distribution of task execution times which is crucial to determine straggler tasks [1], [2]. Specifically, diversity in task execution times leads to the presence of tasks with extremely high or low execution times. This makes the state space of the neural network very large when modelling the distribution of task response times and hence it is often omitted in practical approaches [22], [23]. Second, these approaches ignore the heterogeneous host capabilities, which can also lead to poor scheduling or mitigation decisions [21]. Therefore, a new method is required which can both proactively predict straggler tasks and efficiently mitigate them. As an example of a heterogeneous execution environment, fog-cloud environments leverage resource capabilities from both edge devices and cloud nodes [21]. This leads to high diversity in the computational resources among host devices in the same environment. This host heterogeneity impacts the response time as scheduling in a constrained device may significantly increase its response time.

These issues motivate us to develop a novel online SLA-aware STrAggler PRediction and Mitigation (START) technique. START uses a machine learning model in tandem with an underlying distribution or task response time for automatic and accurate straggler prediction. To allow mapping of heterogeneous environments, encoder networks have shown to be a promising solution [24]. Moreover, prior works also show that in dynamic environments, Long-Short-Term-Memory (LSTM) based neural networks help to adapt to environment changes [25]. Hence, we use an Encoder-LSTM network to analyze the state of a cloud environment. Here, the state of the cloud setup is characterized as a set of host and task parameters like SLA, CPU, RAM, Disk and bandwidth consumption. These parameters are motivated by prior work [26]. Further, as prior work has shown that response times of tasks in large-scale cloud setups follow a Pareto distribution [1], we use the Encoder-LSTM network to predict this distribution in advance to alleviate the straggler problem proactively.

START also uses speculation and rerun-based approaches for Straggler Mitigation during the execution of jobs. Prediction allows early mitigation, reducing the SLA violation rate and execution time and maintaining QoS at the required level. Our performance evaluation is carried out using CloudSim 5.0 [27] and compares our technique with well-known existing techniques (SGC [9], Dolly [20], GRASS [8], NearestFit [6], Wrangler [17], and IGRU-SD [22]) in terms of QoS parameters such as energy consumption, execution time, resource contention, CPU utilization and SLA violation rate. Experimental results demonstrate that START gives lower execution time and SLA violations than existing techniques, also offering low computational overhead.

The rest of the paper is structured as follows. Section 2 presents related work. Section 3 details START. Sections 4 and 5 describe the evaluation setup and experimental results. Finally, Section 6 concludes and outlines future research directions.

2 Related Work

Existing straggler analysis and mitigation techniques can be mainly divided into two main categories: detection and mitigation [1], [2]. The former primarily identify stragglers from utilization metrics and traces from a job execution environment like a CDC. Most of these techniques leverage offline analytics and real-time monitoring methods. Examples of such techniques include NearestFit [6] and SMT [28]. Within this category, other techniques use prediction models to apriori determine the set of tasks in a job that might be stragglers. Examples include RPPS [23] and IGRU-SD [22]. When considering mitigation, approaches either avoid straggler tasks or prevent high response times by methods such as rescheduling, balancing load or running job replicas (clones). Examples of such strategies include Dolly [20], GRASS [8],

Table 1
Comparison of existing models with START

Technique	Straggler Detection	Straggler Mitigation	Proactive Mechanism	Straggler Prediction	Impact on QoS and Utilization	Dynamic	Heterogeneous Environment
Detection Only Methods							
NearestFit [6]	X					X	
SMT [28]	X				X		
SMA [14]	X						
RDD [19]	X						
Mitigation Only Methods							
LATE [29]		X			X		X
Dolly [20]		X	X				
GRASS [8]		X	X		X		
Dolly [20]		X	X				
GRASS [8]		X	X		X		
Wrangler [17]		X				X	
Prediction based Mitigation Methods							
SGC [9]	X	X	X	X	X		X
IGRU-SD [22]	X	X	X	X		X	
START (this work)	X	X	X	X	X	X	X

LATE [29] and Wrangler [17]. Table 1 summarizes the comparison of START with prior approaches. The table shows which works use straggler prediction, mitigation and/or detection. Further, proactive mechanism shows if methods use prediction data to proactively mitigate straggler tasks or wait till completion of other tasks. Impact on QoS and Utilization shows whether these methods utilize QoS and host utilization metrics as feedback to improve prediction or mitigation performance. Dynamic refers to whether these methods are able to adapt to changing host/task characteristics. Heterogeneous environment refers to whether a method assumes resources to have the same computational characteristics.

Straggler Detection. The NearestFit strategy aims at improving the performance of distributed computing systems by resolving data skewness and detecting straggler tasks or unbalanced load. Through this model, [6] proposes a fully-online nearest neighbor regression method that uses statistical techniques to pro le the tasks running in the system. This model gathers pro les using ef cient data streaming algorithms and acts as a progress indicator and it is therefore suited to applications with long run times. Even though this indicator is able to pro le complex and large-scale systems, it is not suitable for heterogeneous resource types as it does not differentiate hosts on the basis of computational capacities. Further, it does not take into account task failures or load on each host.

Straggler Prediction. The work in [23] proposes a resource prediction and provisioning scheme (RPPS) using the Autoregressive Integrated Moving Average (ARIMA) model, which is a statistical model for the prediction of future workload characteristics of various tasks running in a CDC. The work in [22] very recently proposed a technique called Improved Gated Recurrent Unit with Stragglers Detection (IGRU-SD) to predict average resource requests over time. They use this prediction scheme to then run detection algorithms for predicting which tasks might be a straggler. However, they do not consider host heterogeneity, nor do they consider the underlying task distribution, both of which are crucial for predicting if a task is likely to become a straggler.

Straggler Mitigation. The work in [20] explores straggler mitigation techniques and proposes, Dolly, a speculative execution-based approach that launches multiple clones of expected straggler tasks and takes the results of the clone, which nishes execution rst without waiting for the other ones to complete execution. However, there needs to be a careful balance maintained as over-cloning requires extra resources and could lead to contention. On the other hand, under-cloning could lead to slower task execution and no effective improvement. The authors designed and experimented with short workloads with a small number of jobs. They identify that the cloning of a small number of jobs that have short execution times improves reliability without using too much additional resources. Dolly introduces a budgeted cloning strategy to only give an excess of 5% resource consumption for a total of up to 46% improvement in average job response time.

The work in [8] proposes a strategy called Greedy and Resource Aware Speculative Scheduling (GRASS). GRASS uses a similar strategy to Dolly, of spawning multiple clones of slow tasks but also uses greedy speculation to approximate which tasks need to be cloned, and dedicate speculation resources to improve the average deadline-bound job response time by up to 47% and error-bound jobs by up to 38%. The work in [29] explores the MapReduce framework to investigate the occurrence of straggler tasks and optimizes its performance in a heterogeneous cloud environment. Further, the work in [5] proposes the Longest Approximate Time to End (LATE) scheduling algorithm, which uses heuristics to search for the optimum task scheduling policy with latency and cost estimates. They also estimate the response times of all tasks of a job and assume that the one with the longest time is a straggler and execute a copy on a powerful host to reduce overall job response time. However, these works [5], [8], [29] do not adapt to dynamic environments.

The work in [17] proposes a proactive straggler management approach called Wrangler. The underpinning predictive model uses a statistical learning technique on cluster utilization counter-data. To overcome modeling errors and maintain high reliability, Wrangler computes con dence bounds on the predictions and exploits them in the straggler management process. Speci cally, Wrangler relies on a Ganglia based node monitoring to delay the execution of tasks on nodes that have straggler con dence above a threshold value. Experiments on a Hadoop-based EC2 cluster show that Wrangler is able to reduce response times by as much

as 61%, with 55% less resources when compared to other speculative cloning based strategies. However, we show in our experiments that in certain load regimes, e.g., with low resource utilisations or with highly volatile workloads, Wrangler suffers from lower accuracy.

Straggler Prediction and Mitigation. The work in [9] presents a Stochastic Gradient Coding (SGC) based approach which uses approximate gradient coding to reduce the occurrence of straggler tasks. They utilize a pair-wise balanced scheme to determine the jobs to run as a clone or redundant tasks. The SGC algorithm runs in a distributed fashion, sharing a datapoint with multiple hosts to compute independent gradients on the data which is aggregated by the master. This approach prevents the straggler analysis itself from becoming slow and hence is appropriate for volatile environments. However, in large-scale setups, monitoring data across all host machines is inefficient and can create network bandwidth contention, negatively impacting job response times. The work in [30] proposes a task replication approach for job scheduling to minimize the effect of the Long-Tail problem. The authors analyze the impact of this approach in a heterogeneous platform. Their algorithm predicts the mean service times for single and multi-fork scenarios and chooses the optimal forking level. This allows their model to run multiple instances in datacenters with powerful computational resources. However, the approach can handle only a single job system with the same workload characteristics and fails in the presence of diverse workloads as pointed by [30].

3 System Model

We now describe the system model, which predicts the number of straggler tasks to avoid the Long Tail problem. The prediction problem requires a model to know beforehand which tasks, or at least what number of tasks may adversely impact the performance of the system. This depends on not only the types of job being executed on the CDC, but also the characteristics of the physical machines. We first discuss a Pareto distribution based model that is able to predict the number of straggler tasks based on user specifications and hyper-parameters. Later, we describe another deep learning (DL) based approach that generates these hyper-parameters of the Pareto distribution based on the characteristics of the jobs and physical cloud machines.

A summary of our system model components and interaction is shown in Figure 1. Here, the Cloud Environment consists of a cloud scheduler and host machines. The scheduler allocates tasks onto the hosts, which are then executed and utilization metrics are captured by the resource monitoring service of the cloud environment. The utilization metrics of hosts and active tasks are then used to develop feature vectors by the Feature Extractor. The user also provides new jobs for which the feature vectors are instantiated as 0. The host and task feature vectors are then combined to form matrices that are then forwarded to a Straggler Prediction module. The expected tasks flagged as stragglers by the prediction module are then mitigated using a task speculation or a re-run strategy as we describe later.

We consider a bag-of-tasks job model where a bounded timeline is divided into equal sized scheduling intervals.

Figure 1. START System Architecture

At the start of each interval, the model receives a set of independent jobs. SLA deadlines are defined for each job at the time it is sent to the model. Each job consists of q dependent or independent tasks, where $0 < q \leq q^0$. We now describe the modeling of the response times of tasks using the Pareto distribution.

3.1 Pareto Distribution Model

As observed in prior work such as [1], [2], [5], the task execution times in a cloud computing environment can be assumed to follow a Pareto Distribution for which the Cumulative Distribution Function (CDF) is

$$F_X(x) = \begin{cases} 1 - \left(\frac{x}{\alpha}\right)^{-\beta} & x \geq \alpha \\ 0 & x < \alpha; \end{cases} \quad (1)$$

where α is the least time taken among tasks, and β is the tail index parameter ($\alpha > 0, \beta > 0$). $X_1, X_2, \ldots, X_q$ are the times taken by q tasks of a particular job running on the Cloud Environment. The Log-Likelihood Estimate [31] is then

$$\log(L(X_1, \ldots, X_q)) = q\log(\beta) + q\beta\log(\alpha) - (\beta+1)\sum_{i=1}^{q}\log(X_i); \quad (2)$$

where L is the likelihood function for the random variables $X_1, \ldots, X_q$.

As $\beta > 0$, to maximize the log likelihood, α is obtained as the largest possible value such that $X_i \geq \alpha \;\forall i$. Thus, $\alpha = \min_i(X_i)$. For β, if we set a partial derivative of the likelihood with respect to β as 0, we get

$$\beta = \frac{q}{\sum_{i=1}^{q}\log(X_i) - q\log(\alpha)}. \quad (3)$$

For a given job execution, the task execution times determine the (α, β) parameters of the assumed distribution. Thus, at the time of training, we run multiple jobs and fit the parameters using Equation 3. These parameters are then used to predict the number of straggler tasks based on a straggler parameter K, by calculating the number of tasks which in expectation could have completion times greater than K. Thus, for $\beta > 1$ (for a well defined mean of the distribution) and q tasks, $q(1 - F_X(K))$ gives us the expected number of straggler tasks, where F_X is the cumulative distribution function. For mathematical simplicity, we keep the straggler parameter as a multiple of the mean execution time, given as $K = k\,\mu = \left(\frac{\beta}{\beta-1}\right)$. This gives the expected number of straggler tasks (E_S),

$$E_S = q\left(\frac{K}{\alpha}\right)^{-\beta} \quad (4)$$

Figure 2. Empirical results for different hyper-parameter values comparing F1 scores of straggler classification on test data. k; I and T are defined in Sections 3.1 and 3.2. F1 score is defined as per Eq. 5.

(a) M_H (b) M_T

Figure 3. Matrix Representation of Model Inputs

Empirically [1], we find that $k = 1.5$ strikes a good balance between the cases and hence this value is used in the experiments, but can be changed as per user requirements. Figure 2 demonstrates results corresponding to simple grid search on the three parameters k, I and T. The latter two parameters are defined in Section 3.2. For $k = 1.5$, the prediction performance (F1 score) is the highest. For each task in the system, we check whether the predicted class is true or not, i.e., if the completion time of the task is $> K$. The number of correct class labels is denoted as tp and incorrect ones as fp, then the F1 score is defined as

$$\frac{tp}{tp + \frac{1}{2}(fp + tp)}. \quad (5)$$

For $k < 1.5$ the model has high false negatives, whereas for $k > 1.5$, the model has high false positives.

3.2 Encoder Network

The previous subsection shows how the Pareto distribution can be used to determine the expected number of straggler tasks in a job. However, the parameters $(\ ;\ )$ are not known beforehand for a job. As motivated in Section 1, to predict these parameters, we use an encoder network that analyzes the tasks and the workloads at different machines in the CDC for a finite amount of time.

1. As given in Figure 2, based on the method described in [17] and a dataset extracted from traces on a desktop system with 64-bit Ubuntu 18.04 operating system, which is equipped with the Intel® Core™ i7-10700K processor (No. of Cores = 8, Processor Base frequency = 3.80 GHz and turbo frequency = 5.10 GHz), 64 GB of RAM, and 1 TB NVMe storage. We have used Hadoop MapReduce for manage and execute word count application.

Figure 4. Straggler prediction model

We first identify a job j as a set of tasks $\{T_i\}_{i=1}^{q}$, where $q < q'$ if less than q' tasks then rest $q' - q$ rows are 0. For each task T_a, p feature values are used to form a feature vector. Similarly, for each host out of n hosts $\{H_i\}_{i=1}^{n}$, m feature values are used. The features used for hosts include utilization and capacity of CPU, RAM, Disk and network bandwidth. The feature vector also includes the cost, power characteristics, and the number of tasks to which this host is allocated. The features used for tasks include CPU, RAM, Disk and bandwidth requirements and the host assigned in the previous interval. These were used to characterize the system state for deep learning models as is common in prior art [32]–[34]. These feature vectors of hosts (M_H) and tasks (M_T), as shown in Figure 3, are then used to predict the Pareto parameter values. The neural network model and the working of the system is shown in Figure 4. The input matrices are first passed through an encoder network, the output of which is sent to a Long Short Term Memory (LSTM) network [35]. To prevent the LSTM model from diverging, we take an exponential moving average of each matrix using a 0.8 weight to the latest resource matrix (as in [36]). For time-series prediction, multiple machine learning models could be used, including Echo State Networks (ESN) or LSTMs [37]. However, as ESNs control the degree of delays using a manually chosen constant (leaking rate), this typically lowers the generalization ability when applied to different load traces [38]. Hence, we use LSTMs to develop our parameter estimation model.

The Encoder network is a 4 layer fully-connected network with the following details (adapted from prior art [24], [32], [33]):

- Input layer of size $|M_H| + |M_T|$. The non-linearity used here is softplus[2] as in [32]. The matrices are flattened, concatenated and given as an input to the encoder network.
- Fully connect layer of size 128 with softplus activation.
- Fully connect layer of size 128 with softplus activation.
- Fully connect layer of size 32 with softplus activation.

We run inference using a neural network model for each job. Specifically, for each job j, we provide the model with the inputs M_H for host characteristics and M_T for all running tasks in j. For each job, we generate $\ ;\ $ parameters of the Pareto distribution to evaluate the number of straggler tasks. The LSTM network has 2 layers with size 32 nodes. The predicted output of the LSTM network becomes an input for a fully connected layer with 2 nodes, which

2. The definitions of these activation functions can be seen at the PyTorch web-page: https://pytorch.org/docs/stable/nn.html

Table 2
Notation

Symbol	Meaning
q	Maximum number of tasks in a job
α; β	Parameters of the Pareto distribution
K	Straggler parameter in START
E_S	Expected number of straggler tasks
I	Time-period of START inference in seconds
T	Time-duration of START inference in seconds
n	Number of hosts

outputs the $(\alpha; \beta)$ values after a Rectified Non-linear Unit (ReLU) so that these values are positive (with addition of 1 to α so that the mean of the distribution is defined). This is sent to the LSTM Network. To implement the proposed approach, we use PyTorch Autograd package [39] to run the back-propagation procedure for network training. We keep sending the input matrices for a finite time of T, periodically after every I seconds. The LSTM cell takes in two inputs, the hidden state of the previous interval and the output of the encoder network. Considering the output of the previous iteration, i.e., the hidden state h_{t-1} and the output of the encoder network x, the output for the current interval becomes $h_t = LSTM(h_{t-1}; x)$ (see Figure 4). Here, $h_0 = 0$ and $t \in \{0, I, 2I, ..., TI\}$. Using grid-search, for the experiments we set $I = 1$ and $T = 5$, which empirically gives the best results[1].

The output of LSTM network gives us the parameters for the Pareto distribution, which are then used to find expected straggler tasks (E_S). This constitutes the Straggler Prediction module in Figure 1. The objective of the model training is to predict the appropriate distribution parameters using the utilization metrics and use this distribution to calculate the expected number of straggler tasks as described in Section 3.1. E_S determines the number of tasks to mitigate using rerun/speculation-based methods, as explained in the next subsection. Out of the q tasks, first the parameters $(\alpha; \beta)$ are calculated after T time-steps and then $\lfloor E_S \rfloor$ tasks are mitigated. This ensures that if E_S is very small (< 1), we do not mitigate any tasks, saving computational resources. Hence, after execution of $q - \lfloor E_S \rfloor$ tasks, we apply mitigation techniques on the remaining tasks to prevent delays in result generation. Compared to other methods, our model nearly eliminates the detection time and hence is able to provide a faster response to users (as shown in Section 5). The main symbols and their meanings are summarized in Table 2.

Algorithm 1 Straggler Prediction and Mitigation Algorithm

Inputs:
J Set of all jobs being executed currently $[j_1; j_2; ...; j_r]$
T^n_m Set of tasks of job j_n where $m \in \{1, 2, 3, ..., q\}$
M_{time} Max allocated time to release the resource.
Variables:
J_n Set of normal jobs J without straggler tasks
J_s Set of jobs J with > 0 straggler tasks
Procedure PREDICTSTRAGGLER(job)
  for time t from 0 to T with step I
    q Number of tasks in input job
    Extract feature vectors of host machines as M_H
    Extract feature vectors of tasks of input job as M_T
  Predict $(\alpha; \beta)$ using the Neural network
  Find E_S as $q \frac{K}{\alpha}$
  Run job till completion of $q - \lfloor E_S \rfloor$ tasks
  return incomplete tasks
Procedure SPECULATION (task list)
  for task t in task list
    Run a copy of t on a different node
Procedure RERUNSTRAGGLERTASK(task list)
  for task t in task list
    Run the same task t on different node
Begin
for job j_i in J
  stragglerTasks PREDICTSTRAGGLER(j_i)
  if stragglerTasks is empty
    add j_i to J_n
    continue
  else
    add j_i to J_s
    Wait for specific time (M_{time}), if j_i does not respond then generate alert for action
  if j_i is deadline oriented
    SPECULATION (stragglerTasks)
  else
    RERUNSTRAGGLERTASK(stragglerTasks)

Figure 5. Comparison of START with detection based approaches.

3.3 Speculation and Task Rerun

To mitigate the Long Tail problem, we use the following two strategies (as in prior work [1], [30]) for the straggler tasks detected by our prediction model.

1) Speculation: We run a copy of the straggler task on a separate node and use the results we get first. This is crucial for deadline driven tasks that need results as soon as possible. Thus, this method gives us the least response time at the cost of running multiple nodes.
2) Re-Run Task: We stop execution of the straggler task on the respective node and run a new instance of the same task in a new node. This method is suitable for tasks that are not deadline critical as it runs only one copy of the task at a time which reduces energy consumption and prevents congestion.

The choice of the separate or new node is performed by the underlying scheduling scheme (further details in Section 4). We do not consider task cloning as it has significant overheads in large-scale environments [40]. In both approaches mentioned above, we select the new node that has the lowest moving average of the number of straggler tasks for the current time-step. Algorithm 1 describes in detail the complete approach of straggler prediction and mitigation and is run periodically to eliminate the long tail

problem. As shown, START first determines the host and task feature matrices for every job (lines 8 and 9), which are then analyzed for T time-steps to predict the number of straggler tasks (line 13). For each job which has $b\epsilon_S c > 0$, mitigation techniques are run for remaining tasks when only $b\epsilon_S c$ of them are left (lines 30 and 32). Figure 5 shows how START is able to provide much lower response times compared to existing detection based algorithms by nearly eliminating the detection time as it predicts early-on the number of tasks that are highly likely to be stragglers. This constitutes the Straggler Mitigation module in Figure 1.

4 Evaluation Setup

4.1 Evaluation Metrics

We use common evaluation metrics [1], [8], [9]. We assume there are n host and q jobs currently in the system.

1) Energy Consumption: The cumulative energy consumed for a given time is given by

$$E = E_{CPU} + E_{Disk} + E_{Memory} + E_{Network} + E_{Misc}; \quad (6)$$

where E_{CPU} is the total energy consumed by all the processors, which includes dynamic energy as CV^2f, short-circuit energy, leakage energy, and idle energy consumption [10]. E_{Disk} is the energy consumed for all read/write operations plus the idle energy consumed by all the disks. E_{Memory} is the energy consumed by all memories (RAM and Cache) in the computational nodes. $E_{Network}$ is sum of energies consumed by network devices which include routers, gateways, LAN cards and switches. E_{Misc} is energy consumed by other components like motherboard and port connectors. However, in simulation it is difficult to find out each energy component separately, so we calculate maximum and minimum energy consumption ($E_{max}; E_{min}$) by hardware profiling as per Equation 6 and using Standard Performance Evaluation Corporation (SPEC) benchmarks https://www.spec.org/cloud_iaas2018/results/ .
We then use Equation 7 to get total energy consumption in CloudSim at time t. Here, U_k^t is the total host resource utilization (sum of all workloads) of host k. This is a common practice [27]. Thus,

$$E_{total}^t = \sum_{k=1}^{n} U_k^t \ (E_{max} - E_{min}) + E_{min}: \quad (7)$$

2) Execution Time: The average execution time is

$$T_{avg}^{exec} = \frac{1}{q}\sum_{i=1}^{q} (T_i^C - T_i^S) + \sum_{i=1}^{q} R_i: \quad (8)$$

This is the total time taken to successfully execute an application, on average, for all tasks. Here T_i^C, T_i^S and R_i are the completion, submission and restart time of task i.

3) Resource Contention: Resource contention occurs when one workload shares the same resource during the execution [20]. This may be due to unavailability of the required number of resources, or because there are a large number of workloads with urgent deadlines. Resource contention is quantified as

$$Con_{total}^{resource} = \sum_{k=1}^{n}\sum_{i=1}^{q_k} Req_{i,k}^{resource} \ \mathbb{1}(resource_i \text{ overloaded}); \quad (9)$$

where q_k is the number of tasks being executed at resource k and $Req_{i,k}^{resource}$ is the resource requirement of i^{th} task at node k. Also, $\mathbb{1}()$ denotes the indicator function.

4) Memory Utilization: The memory utilization of host k in percentage terms is

$$U_k^{memory} = \frac{P_k^{total} - (F_k + B_k + C_k)}{P_k^{total}} \ 100; \quad (10)$$

where $P_k^{total}; F_k; B_k; C_k$ are the total physical, free, buffer and cache memory respectively.

5) Disk Utilization: The disk utilization of host k in percentage terms is

$$U_k^{disk} = \frac{Total\ Used}{Total\ HD\ Size} \ 100: \quad (11)$$

6) Network Utilization: The network utilization of host k in percentage terms is

$$U_k^{network} = \frac{Bits_{total}^{rx} + Bits_{total}^{tx}}{BW_k \quad S_I} \ 100; \quad (12)$$

where $Bits_{total}^{rx}$ and $Bits_{total}^{tx}$ are the total bits received and transmitted in an interval. BW_k is the bandwidth of host k and S_I is the size of the interval.

7) SLA Violation Rate: For q tasks we have q SLAs. Each SLA has a weight (i^{th} SLA having weight w_i). The total SLA violation rate is

$$SLA_{total}^{violation} = \frac{1}{q}\sum_{i=1}^{q} w_i \ \mathbb{1}(SLA_i \text{ is violated}): \quad (13)$$

We also use other metrics including Resource contention, CPU utilization and Completion times as defined in [41].

As per prior work [1], the metric for comparing prediction accuracy is the Mean Average Percentage Error (MAPE) which is defined as the mean percentage error of the predicted value (number of straggler tasks for each job) from the actual value and given by Equation 14. To obtain the actual value, we only perform straggler prediction and compare MAPE of START, IGRU-SD and RPPS [23] as other baselines do not perform straggler prediction. We use this to calculate the number of straggler tasks using maximum-likelihood estimation (see Equation 4). Thus,

$$MAPE = \frac{100\%}{n}\sum_{t=1}^{n} \frac{y_t \quad y_t^0}{y_t}; \quad (14)$$

where y_t and y_t^0 are the actual and predicted number of straggler tasks and n is the number of scheduling intervals for the complete simulation.

4.2 Workload Model

Our evaluation uses CloudSim toolkit and real-time workload traces are derived from PlanetLab systems [42]. This dataset contains traces of CPU, RAM, disk, and network bandwidth requirements from over 1000 PlanetLab tasks collected during 10 random days. These traces are collected

Table 3
Con guration Details of simulated Physical machines

CPU	RAM and Storage	Core count	Operating System	Number of Virtual Nodes
Intel Core 2 Duo - 2.4 GHz	6 GB RAM and 320 GB HDD	2	Windows	12
Intel Core i5-2310- 2.9GHz	4 GB RAM and 160 GB HDD	4	Linux	6
Intel XEON E 52407-2.2 GHz	2 GB RAM and 160 GB HDD	4	Linux	2

using a scheduling interval size of 300 seconds. The virtual machines are located at more than 500 places across the globe. The data was collected on 2880 intervals each, thus each trace was of this size [43][3]. In this dataset, 50% of the traces are deadline driven and 50% are not. We get similar results on other distributions. A collection of 2 to 10 tasks is de ned as a job. We use data for 800 tasks as our training set and 100 tasks' data as the test set. As in prior work [32], a Poisson Distribution $Poisson(\ )$, with $= 1:2$ jobs, is selected for the number of jobs to be created periodically. This is because all the workloads/tasks of different jobs are independent of each other. The requests submitted by users are considered as cloudlets, which have three speci c requirements (CPU, memory and task length).

4.3 CloudSim Simulation Environment

We evaluate the performance of START using a simulated cloud environment. We implement our straggler detection and mitigation technique by introducing the different kinds of faults using an event-driven module. The neural network and back-propagation through time code were implemented using PyTorch library in Python. As in prior work [44], we have used a Weibull Distribution to model failure characteristics. The failure distribution is given by

$$f(x;k;\ ) = \frac{k}{\ }\frac{x}{\ }^{k\ 1}; \qquad (15)$$

where x is the time-to-failure. We assign the parameters $k = 1:5;\ = 2$ as in [44], [45]. The introduced fault types are (1) host faults (memory faults and faults in the processing elements), (2) Cloudlet faults (due to network faults) and (3) VM creation faults. We consider task faults where the underpinning applications need to rerun due to task breakdown. For host failure, all tasks running in that host need to restart. We consider only ephemeral host faults, i.e., our hosts are of ine for a short duration of time (up to 4 intervals in our experiments) instead of being permanently down. Other faults considered in the system include unavailability of memory space, disk page faults and network packet drops that increase the response time of running tasks. Every change in the states of VMs and hosts should be realized by the cloud datacenter through the cloud broker. Further, the broker uses a cloudlet speci cation to request the creation of VM and scheduling of cloudlets. We have designed a Fault Injection Module to create a fault injector thread by simulating the cloudlet faults, host faults and VM creation faults. A failed node can return to service only after a downtime as de ned in [44].

3. The traces from the PlanetLab systems can be downloaded from https://www.planet-lab.org/planetlablogs .

Table 4
Simulation Parameters for experiments

Parameter	Value
Number of VMs (n)	400
Number of Cloudlets (Workloads)	5000
Host Bandwidth	1 -2 KB/S
CPU IPS (in millions)	2000
Cloud Workload size	10000 3000 MB
Cloud Workload cost	3 - 5 C$
Memory Size	2-12 GB
Input File size	300 120 MB
Output File size	300 150 MB
Power Consumption (KW)	108 - 273 KW
Latency of hosts	20-90 Seconds
Size of Cache memory	4 - 16 MB
CPU Power Consumption	130 - 240W
RAM Power Consumption	10 - 30W
Disk Power Consumption	3 - 110W
Network Power Consumption	70 - 180W
Power Consumption of other Components	2 - 25W

The Fault injector thread uses a Weibull Distribution and generates events which execute commands such as “sendNow(dataCenter.getId(), FaultEvent-Tags.HOST_FAILURE, host);” [44]. The Fault Injection Module contains three entities such as FaultInjector, FaultEvent and FaultHandlerDatacenter. FaultInjector extends the SimEntity class of CloudSim and start simulation to insert fault events randomly using the Weibull Distribution. FaultEvent extends the SimEvent class of CloudSim, which describes the type of faults such as create VM failure, cloudlet failure and host failure. FaultHandlerDatacenter extends the Datacenter class and processes fault events sent by the FaultGenerator and handles VM migration. In this simulation setup, four Physical Machines (PMs) characteristics (CPU, RAM, Disk and Bandwidth capabilities) are used with a various number of virtual nodes as shown in Table 3. Since straggler tasks are particularly common in resource-constrained devices [1], we use devices with low core count and RAM for our experiments. The test setup is similar to prior work [41] .

Table 4 details the values of the simulation parameters used in the performance evaluation, collected from the existing literature and empirical studies [10], [46]–[48]. We keep the parameters I and T xed as 1 and 5 seconds respectively throughout the simulation. We dynamically change the k value based on empirical results for the data up till the current interval with the initial value as $1:5$ (as described in Section 1).

4.4 Model Training

To train the Encoder-LSTM network, we use the PlanetLab dataset and divide the workloads of 1000 tasks into 80%

training dataset and the rest as the test dataset. For training and test sets too, we keep the 50-50 ratio of tasks that are deadline-driven to those that are not. Further, we use a scheduler that selects tasks at random and schedules them randomly to any host using a uniform distribution. The random scheduler allows us to obtain diverse host and task characteristics for model training, which is crucial to prevent under- tting of the neural network. The response time histogram was generated and compared against the (;) output of the Encoder-LSTM network. The model was trained using Mean-Square-Error Loss between the values based on the predicted distribution and the actual data. We used a learning rate of 10^{-5} and the Adam optimizer to train the network [49].

4.5 VM Scheduling Policy

We use the A3C-R2N2 policy which schedules workloads using a policy gradient based reinforcement learning strategy which tries to optimize an actor-critic pair of agents [32]. This approach uses Residual Recurrent Neural Networks (R2N2) to predict the expected reward for each action (i.e scheduling decision) and tries to optimize the cumulative reward signal. The A3C-R2N2 policy has been shown to outperform other policies in terms of response time and SLA violations [32]; hence, it is our choice of scheduling method for comparing straggler mitigation techniques.

4.6 Baseline Algorithms

We have selected six baseline techniquesNearestFit, Dolly, GRASS, SGC, Wranglerand IGRU-SD which are the most recent among prior works (see Section 2 for details). We have chosen recent and relevant techniques from the literature to validate our technique against state-of-the-art techniques.

1) NearestFit: uses a statistical curve tting approach to detect stragglers. The function $a + b\ x^c$ is tted with x as the size of the input le for a task [6]. However, vanilla NearestFit is not able to mitigate the detected stragglers, so we use speculation on the detected tasks.
2) Dolly: is a straggler mitigation technique that forks tasks into multiple clones which are executed in parallel within their speci ed budget. The number of clones are calculated based on the Upper-Con dence-Bound as in [20] using the CPU utilization of tasks.
3) GRASS: is straggler mitigation framework, which uses the concept of speculation to mitigate stragglers reactively. It is implemented using two algorithms, one for greedy speculation and the other for resource-aware scheduling.
4) SGC: is an approach using distributed gradient calculation to utilize a pair-wise balancing scheme for running clones of tasks.
5) Wrangler: is a proactive straggler mitigation technique, which uses linear modelling approach to reduce the utilization of excess resources by delaying the start of tasks predicted as straggler.
6) IGRU-SD: is a GRU neural network based resource requirement prediction technique which uses detection mechanisms on the predicted future characteristics [22]. As it only predicts straggler tasks and does not mitigate them, we use the same re-run and speculation strategy (based on deadline requirements) for fair comparison.

5 Performance Evaluation

5.1 Experimental Observations

As in prior work [1], [30], we used QoS parameters to evaluate the performance of START as compared to the existing techniques. We run our experiments for 24 hours, i.e., 288 scheduling intervals. We average over 5 runs and use diverse workload types to ensure statistical signi cance.

5.1.1 Variation of Resource Utilization

We consider 4 types of reserved utilization for CPU, disk, memory and network, where utilization is blocked intentionally (20%, 40%, 60% and 80%) to test the performance of the proposed technique. Figure 6 shows the comparison of QoS parameters such as Execution Time, Energy, Resource Contention and SVR with different values of CPU, disk, network and memory utilization.

Figure 6(a) shows the value of execution time for different straggler management techniques with variation in the value of CPU, disk, network and memory utilization. The value of execution time increases with the increase in the value of reserved utilization, but START performs better than the existing techniques because it tracks the states of the resources dynamically for ef cient decisions. The value of execution time in START is 11.47-17.4% less than the baseline methods. Figure 6(b) shows the variation of resource contention with different values of utilization. The value of resource contention increases as the value of utilization increases. The value of resource contention in START is 12.34-15.19% less than the baseline methods. This is due to the execution time variation across various tasks and resources due to the ltered resource list obtained from the resource provisioning unit (see Section 2).

Figure 6(c) shows the energy consumption for different values of utilization and we observe that energy consumption increases with the utilization for all straggler management techniques. However, START performs better than the prior art as it avoids over or under-utilization of resources during scheduling. The value of energy consumption in START is between 18.55% and 22.43% less than the baseline methods. Figure 6(d) shows the variation of SLA violation rate with different values of utilization and value of SLA violation rate is increasing as the value of utilization increases. The value of SLA violation rate in START is between 21.34% and 26.77% less than the baseline methods. This occurs because START uses admission control and a reservation mechanism for execution of workloads in advance.

5.1.2 Variation of Number of Workloads

In this section we evaluate the value of various performance parameters as we increase the number of workloads.

Figure 7(a) shows the variation of execution time with different numbers of workloads. The value of execution time in START is 19.74-23.84% less than the baseline methods. The interpretation of resource contention for different numbers of workloads is shown in Figure 7(b) which shows the value of resource contention increases with the increase in the number of workloads. START performs better than existing techniques; the average value of resource contention in START is 19.12-24.84% less than the baseline methods. Figure 7(c) shows the variation of energy consumption

Figure 6. Comparison of QoS parameters with different value of CPU, disk, network and memory Utilization: a) Execution Time, b) Resource Contention, c) Energy Consumption and d) SLA Violation Rate

Figure 7. Comparison of performance parameters with different value of workloads: a) Execution Time, b) Resource Contention, c) Energy Consumption, d) SLA Violation Rate, e) Network Utilization, f) CPU Utilization, g) Disk Utilization and h) Memory Utilization

with different numbers of workloads and the value of energy consumption in START is 13.71-18.01% less than the baseline methods. The variation of SLA violation rate for different number of workloads is shown in Figure 7(d) and the value of SLA violation rate is increasing with the increase in number of workloads but START performs better than existing techniques. The average value of resource contention in START is 9.26-12.92% less than the baseline methods. The reduced execution times (and hence energy consumption and SLA violations) are due to ef cient and proactive mitigation of stragglers by START. Further, using the Pareto distribution allows START to identify stragglers prior to their completion, which reduces resource usage and hence contention.

Figure 7(e) shows that the variation of network utilization with a different number of workloads for START and the baseline methods. All the utilization metrics presented in the gure are averaged across the completed tasks. The experimental results show that the average value of network utilization in START is between 18.6% and 25.67% more than the baseline methods. The variation of CPU utilization with different numbers of workloads is shown in Figure 7(f) and it shows the value of CPU utilization is decreasing with the increase in the number of workloads but START performs better than existing techniques. The value of CPU utilization in START is between 16.61% and 17.29% more than the baseline methods. Figure 7(g) shows the variation of disk utilization with a different number of workloads for all methods. The experimental result show that the average value of disk utilization in START is 13.25-15.34% more than the baseline methods. The variation of memory utilization with a different number of workloads is shown in Figure 7(h) and indicates that the value of memory utilization is decreasing with the increase in the number of workloads but START performs better than existing techniques. The value of memory utilization in START is 7.92-17.54% more than the baseline methods. The reduction in usage of resources in case of START is because of the conservative execution of tasks based on straggler prediction. Instead of running/speculating straggler tasks in advance, START waits for the completion of q b E_sc (refer Algorithm 1). Thus, if the predicted straggler tasks do complete earlier than expected, they are not cloned, avoiding resource wastage.

5.2 Straggler Analysis

Figure 8 shows the variation of completion time of different workloads for different straggler management techniques with different utilization percentages of CPU, disk, memory and network. The line plots show the completion time across

(a) (b) (c) (d)

Figure 8. Comparison of performance based on execution time for different utilization: a) utilization limit = 20%, b) utilization limit = 40%, c) utilization limit = 60% and d) utilization limit = 80%

the workloads sorted by their creation time and the bar plots show the variation in the completion time. A higher variance of completion time implies a higher number of tasks that cause a delay in job completion. Thus, a simple measure for comparison is the variance of execution times across different tasks. Figures 8(a), 8(b), 8(c) and 8(d) show the comparison of START with existing straggler management techniques for 20%, 40%, 60% and 80% reserved utilization respectively. The observed improvement occurs because START is very effective in the detection and mitigation of stragglers at run-time. It is also identi ed that the completion time is increasing with the increase in utilization limit from 20% to 80%. Figure 8(d) shows that START has more variation in job completion time with an 80% utilization limit, but START performs better than existing techniques while detecting and mitigating stragglers more ef ciently.

5.3 Prediction Accuracy Comparison

To demonstrate the ef cacy of the prediction model, we show that the prediction error is minimized in our model. To evaluate prediction error, we use the same environment as before with diverse task requirements and heterogeneous hosts with host failures. We use the MAPE metric for this. For ease of comparison, we consider only 2 physical host types with processors: i5 and Xeon as given in Table 3. We keep a total 200 VMs out of which the number of VMs on the Xeon host are changed with time (the variation is not smooth due to injected VM failures in the model). As shown in Figure 9(d), as the number of VMs on the Xeon host change, the percentage prediction error is higher for RPPS and IGRU-SD than START. This is because these models do not consider the heterogeneity of VM resource capabilities. Clearly, when the number of VMs in the Xeon host change, the heterogeneity changes dynamically, leading to different probabilities of tasks becoming stragglers. Thus, the models in IGRU-SD and RPPS are unable to predict straggler tasks accurately. In contrast, START is able to analyze host resource capabilities with the task allocation to correctly predict straggler tasks.

5.4 Overhead Comparison

Figure 10 shows a comparison of run-times of the START and baseline approaches (including scheduling of re-run or speculated tasks) amortized over the average task execution times. As can be seen, the methods proposed in the prior art are faster at detecting straggler tasks. However, as seen earlier, they do not perform well. START has a slightly higher (0:09%) run-time than the best approach among the prior work (IGRU-SD).

(a) (c) (d) (b)

Figure 9. Comparison of prediction accuracy of START with IGRU-ISD and RPPS. (a) Number of VMs in Xeon host out of total 400 VMs, (b) Comparison of percentage prediction error, (c) MAPE values for modi ed environment with changing host resources (d) MAPE values for initial setup described in Section 5.

Figure 10. Overhead comparison

6 Conclusions and Future Work

We proposed a novel straggler prediction and mitigation technique using an Encoder-LSTM Model for large-scale cloud computing environments. This technique allows us to reduce response time and provide better results with fewer SLA violations compared to prior works. Thanks to the prediction models based on maximum likelihood estimation from a Pareto distribution and recurrent encoder network,

our model is able to predict straggler tasks beforehand and mitigate them early on using speculation and re-run methods. Unlike prior prediction based approaches, START is able to analyze tasks with host characteristics and utilize the underlying Pareto distribution for more accurate prediction and mitigation leading to higher performance than state-of-the-art mechanisms. It is clear that for different workload levels, START performs better giving lower execution time, resource contentions, energy consumption and SLA violation rate. When compared with different levels of workload on the cloud system, again START outperforms the baseline approaches. START has higher CPU, network, RAM and disk utilization. This is because many jobs, and hence, tasks complete quickly which leads to more tasks being nished in a period of time compared to other approaches. This implies that START is able to leverage resources in a more ef cient manner leading to faster job completion and hence also saving energy, even with slightly higher resource utilization for the same number of tasks.

As part of future work, we plan to implement START in real-life settings using fog frameworks such as PRISM [12] or COSCO [26]. This will help in making the model more robust to task and workload stochasticity in real scenarios. Moreover, we can also ne-tune our neural network models and Pareto distribution parameters using a larger dataset which includes diverse fog and cloud applications.

ACKNOWLEDGEMENTS

S.T. is grateful to the Imperial College London for funding his Ph.D. through the President's Ph.D. Scholarship scheme. P.G. and S.S.G are supported by the Engineering and Physical Sciences Research Council (EPSRC) (EP/P031617/1). R.B. is supported by Melbourne-Chindia Cloud Computing (MC3) Research Network and Australian Research Council. The work of G.C. has been partly funded by the EU's Horizon 2020 program under grant agreement No 825040.

REFERENCES

[1] S. S. Gill, X. Ouyang, and P. Garraghan, "Tails in the cloud: a survey and taxonomy of straggler management within large-scale cloud data centres," The Journal of Supercomputing, pp. 1–40, 2020.

[2] H. Xu and W. C. Lau, "Optimization for speculative execution in big data processing clusters," IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 2, pp. 530–545, 2016.

[3] M. Liaqat, A. Naveed, R. L. Ali, J. Shuja, and K.-M. Ko, "Characterizing dynamic load balancing in cloud environments using virtual machine deployment models," IEEE Access, vol. 7, pp. 145 767–145 776, 2019.

[4] S. Mustafa, K. Sattar, J. Shuja, S. Sarwar, T. Maqsood, S. A. Madani, and S. Guizani, "Sla-aware best t decreasing techniques for workload consolidation in clouds," IEEE Access, vol. 7, pp. 135 256–135 267, 2019.

[5] D. Wang, G. Joshi, and G. Wornell, "Using straggler replication to reduce latency in large-scale parallel computing," ACM SIGMETRICS Performance Evaluation Review, vol. 43, no. 3, pp. 7–11, 2015.

[6] E. Coppa and I. Finocchi, "On data skewness, stragglers, and mapreduce progress indicators," in Proceedings of the Sixth ACM Symposium on Cloud Computing. ACM, 2015, pp. 139–152.

[7] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in 2015 IEEE 31st international conference on Data Engineering. IEEE, 2015, pp. 1352–1363.

[8] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "Grass: Trimming stragglers in approximation analytics," in Networked Systems Design and Implementation (NSDI). 2014, pp. 289–302.

[9] R. Bitar, M. Wootters, and S. El Rouayheb, "Stochastic gradient coding for straggler mitigation in distributed learning," IEEE Journal on Selected Areas in Information Theory, 2020.

[10] S. S. Gill, P. Garraghan, V. Stankovski, G. Casale, R. K. Thulasiram, S. K. Ghosh, K. Ramamohanarao, and R. Buyya, "Holistic resource management for sustainable and reliable cloud computing: An innovative solution to global challenge," Journal of Systems and Software, 2019.

[11] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code of oading," in 2012 proceedings IEEE Infocom. IEEE, 2012, pp. 945–953.

[12] D. Lindsay, S. S. Gill, and P. Garraghan, "Prism: An experiment framework for straggler analytics in containerized clusters," in Proceedings of the 5th International Workshop on Container Technologies and Container Clouds. ACM, 2019, pp. 13–18.

[13] P. Garraghan, R. Yang, Z. Wen, A. Romanovsky, J. Xu, R. Buyya, and R. Ranjan, "Emergent failures: Rethinking cloud reliability at scale," IEEE Cloud Computing, vol. 5, no. 5, pp. 12–21, 2018.

[14] D. Wang, G. Joshi, and G. Wornell, "Ef cient task replication for fast response times in parallel computation," in ACM SIGMETRICS Performance Evaluation Review, vol. 42, no. 1. ACM, 2014, pp. 599–600.

[15] U. Kumar and J. Kumar, "A comprehensive review of straggler handling algorithms for mapreduce framework," International Journal of Grid and Distributed Computing, vol. 7, no. 4, pp. 139–148, 2014.

[16] M. F. Aktas, P. Peng, and E. Soljanin, "Effective straggler mitigation: Which clones should attack and when?" ACM SIGMETRICS Performance Evaluation Review, vol. 45, no. 2, pp. 12–14, 2017.

[17] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz, "Wrangler: Predictable and faster jobs using fewer resources," in Proceedings of the ACM Symposium on Cloud Computing. ACM, 2014, pp. 1–14.

[18] F. Farhat, "Stochastic modeling and optimization of stragglers in mapreduce framework," 2015.

[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in Networked Systems Design and Implementation (NSDI). 2012, pp. 2–2.

[20] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in Networked Systems Design and Implementation (NSDI), 2013, pp. 185–198.

[21] S. S. Gill, S. Tuli, M. Xu, I. Singh, K. V. Singh, D. Lindsay, S. Tuli, D. Smirnova, M. Singh, U. Jain et al., "Transformative effects of iot, blockchain and arti cial intelligence on cloud computing: Evolution, vision, trends and open challenges," Internet of Things, p. 100118, 2019.

[22] Y. Lu, L. Liu, J. Panneerselvam, B. Yuan, J. Gu, and N. Antonopoulos, "A gru-based prediction framework for intelligent resource management at cloud data centres in the age of 5g," IEEE Transactions on Cognitive Communications and Networking, 2019.

[23] W. Fang, Z. Lu, J. Wu, and Z. Cao, "Rpps: A novel resource prediction and provisioning scheme in cloud data center," in 2012 IEEE Ninth International Conference on Services Computing. IEEE, 2012, pp. 609–616.

[24] S. Tuli, N. Basumatary, S. S. Gill, M. Kahani, R. C. Arya, G. S. Wander, and R. Buyya, "Healthfog: An ensemble deep learning based smart healthcare system for automatic diagnosis of heart diseases in integrated iot and fog computing environments," Future Gerenation Computer Systems, vol. 104, pp. 187–200, 2020.

[25] S. S. Gill, S. Tuli, A. N. Toosi, F. Cuadrado, P. Garraghan, R. Bahsoon, H. Lut yya, R. Sakellariou, O. Rana, S. Dustdar et al., "Thermosim: Deep learning based framework for modeling and simulation of thermal-aware resource management for cloud computing environments," Journal of Systems and Software, p. 110596, 2020.

[26] S. Tuli, S. Poojara, S. N. Srirama, G. Casale, and N. Jennings, "COSCO: Container Orchestration using Co-Simulation and Gradient Based Optimization for Fog Computing Environments," IEEE Transactions on Parallel and Distributed Systems, 2021.

[27] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," Software: Practice and experience, vol. 41, no. 1, pp. 23–50, 2011.

[28] X. Ouyang, P. Garraghan, D. McKee, P. Townend, and J. Xu, "Straggler detection in parallel computing systems through dynamic threshold calculation," in 2016 IEEE 30th International Conference on