LASER: A Deep Learning Approach for Speculative Execution and Replication of Deadline-Critical Jobs in Cloud

Maotong Xu, Sultan Alamro, Tian Lan and Suresh Subramaniam
Department of Electrical and Computer Engineering
The George Washington University
{htfy8927, alamro, tlan, suresh}@gwu.edu

***Abstract*—Meeting desired application deadlines is crucial as the nature of cloud applications is becoming increasingly mission-critical and deadline-sensitive. Empirical studies on large-scale clusters reveal that a few slow tasks, known as stragglers, could significantly stretch job execution times. A number of strategies are proposed to mitigate stragglers by launching speculative or clone (task) attempts. These strategies often rely on a model-based approach to optimize key operating parameters and are prone to inaccuracy/incompleteness in the underlying models. In this paper, we present LASER, a deep learning approach for speculative execution and replication of deadline-critical jobs. Machine learning has been successfully used to solve a large variety of classification and prediction problems. In particular, the deep neural network (DNN), consisting of multiple hidden layers of units between input and output layers, can provide more accurate regression (prediction) than traditional machine learning algorithms. We compare LASER with SRQuant, a speculative-resume strategy that is based on quantitative analysis. Both these scheduling algorithms aim to improve Probability of Completion before Deadlines (PoCD), i.e., the probability that MapReduce jobs meet their desired deadlines, and reduce the cost of speculative execution, measured by the total (virtual) machine time. We evaluate and compare the two strategies through testbed experiments. The results show that our two strategies outperform Hadoop without speculation (Hadoop-NS) and Hadoop with speculation (Hadoop-S) by up to 89% in PoCD and 13% in cost.**

***Index Terms*—MapReduce, Straggler, Speculative Strategy, Deep Neural Network**

I. INTRODUCTION

Distributed cloud computing frameworks, such as MapReduce, have been widely employed by companies such as Facebook, Google, and Yahoo due to their ability to exploit inherent parallelism in cloud jobs by breaking them into smaller, parallel tasks. Such frameworks are susceptible to heavy tails in response times, and job execution times could be greatly prolonged by just a few slow tasks, called stragglers. Prior work shows that slow tasks can run up to 8 times slower than the median task [**?**], [**?**], [**?**], [**?**]. These stragglers could significantly impact the overall performance of deadline-sensitive cloud applications and result in the violation of Service Level Agreements (SLAs).

Stragglers could occur due to various reasons. First, the heterogeneity of computing infrastructure in a datacenter causes nodes to perform differently. Second, hardware/software errors in large-scale datacenters lead to node failures, and interrupt the task execution on failed nodes. Third, resource contention among tasks running on virtual machines hosted on the same physical machine can slow down processing rates of tasks [**?**], [**?**], [**?**].

Existing straggler mitigation techniques could be categorized into proactive and reactive. For example, Dolly [**?**] is a proactive cloning approach. It launches extra attempts along with the original attempt for each task, and the task completes when the earliest attempt finishes. Wrangler [**?**] employs a statistical learning technique to detect if a straggler is likely to occur on a node, and schedules tasks on nodes that are likely to not cause stragglers. LATE [**?**] proposes a scheduling algorithm to launch speculative attempts based on the progress score of a task. The progress score equals the fraction of data processed. Mantri [**?**] analyzes reasons for stragglers and presents an algorithm to launch extra speculative attempts based on available resources. These strategies often employ model-based optimization to determine the optimal operating parameters, e.g., the number of speculative attempts, but provide little guarantee on meeting individual application deadlines.

Meeting desired deadlines is crucial as the nature of cloud applications is becoming increasingly mission-critical and deadline-sensitive. Yet, extending existing solutions with deadline-awareness is difficult, since the model-based approach is prone to inaccuracy/incompleteness in the underlying models and analysis, limiting their applicability in real-world cloud platforms. In this paper, we present LASER, a deep learning approach for speculative execution and replication of deadline-critical jobs. Machine learning has been used to solve a large variety of classification and prediction problems. We use a deep neural network (DNN), a specific machine learning tool, to solve the straggler mitigation problem in this paper. Deep neural network, modeled loosely after the human brain, consists of an input layer, hidden layers, and an output layer (see Fig. 1). The input layer takes inputs from a dataset, and passes data to the following hidden layer. Each hidden layer, which is not directly exposed to the input data, consists of multiple neurons. A neuron is a computational unit

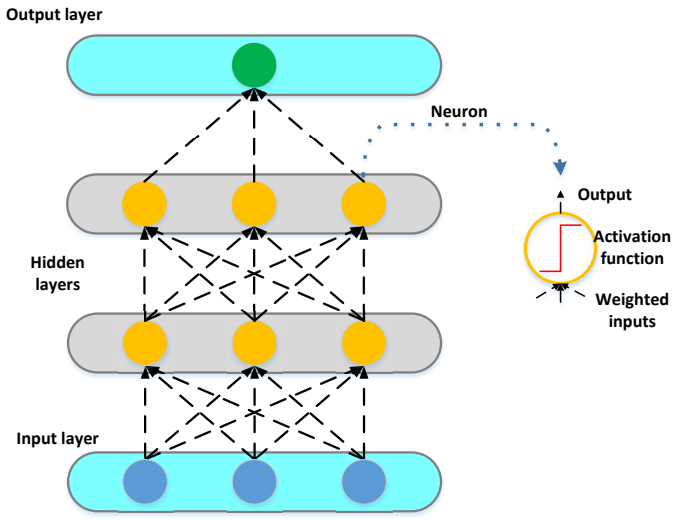


Fig. 1. A deep neural network.

that takes weighted inputs and generates an output using an activation function which is non-linear. In this paper, we use a state-of-the-art activation function, i.e., parametric rectified linear unit (parametric ReLU) [**?**]. The parametric ReLU learns parameters and adjusts them through gradient descent, and has been shown to achieve superhuman performance in some cases [**?**]. The output layer takes results from hidden layers, and produces a value or a vector of values. Mathematically, the output of the lth layer, y_l, is given by:

$$y_l = f_{\text{act}}(W_l \cdot x_l + \beta_l), \tag{1}$$

where f_{act}, W_l, and β_l denote the activation function, the weight of layer l, and the bias of layer l, respectively. Here, x_l is the input of layer l, which is also the output of layer $l-1$, i.e., $x_l = y_{l-1}$.

DNN is known to outperform traditional machine learning algorithms, and is widely used for processing images, videos, speech, etc. [**?**]. To the best of our knowledge, there is no existing work combining DNN with cloud computing framework to mitigate stragglers.

Given a job with a deadline, we propose a metric called the Probability of Completion before Deadline (PoCD) to quantify the likelihood of a straggler mitigation strategy to meet job deadlines. In this paper, we present two scheduling strategies, Speculative-Resume with Quantitative Analysis (*SRQuant*) and Speculative-Resume with DNN (*LASER*) to jointly improve PoCD and reduce the cost resulting from speculative task executions, and find the number of speculative attempts for each straggler. In SRQuant, we provide a quantitative analysis for PoCD and cost. In LASER, we combine DNN with cloud computing framework to predict the task execution time and determine the number of speculative attempts to launch for each straggler based on features collected. Results show that our two strategies outperform Hadoop with no speculation (Hadoop-NS) and Hadoop with default speculative strategy (Hadoop-S) by up to 89% in PoCD and 13% in cost.

The rest of this paper is organized as follows. Section II summarizes related work. Section III presents some background on the Speculative-Resume strategy. Section IV presents SRQuant and the DNN for LASER, and some accompanying analysis. Section V describes the implementation details of the strategies, and Section VI presents results from our experiments. We finally conclude the paper in Section **??**.

II. Related work

While a large variety of techniques have been proposed for improving the performance of MapReduce-like systems, little research has been conducted to mitigate stragglers with the goal of meeting desired deadlines. In this section, we provide a review of closely related works on task scheduling techniques, and straggler mitigation strategies.

To improve the performance of MapReduce-like systems, the design of job schedulers has become an active research topic [**?**], [**?**], [**?**], [**?**], [**?**]. Natjam [**?**] brings a scheduling scheme with support for priorities under deadline constraints. FLEX [**?**] presents a flexible allocation scheme for MapReduce workloads. [**?**] introduces a job scheduler to dynamically determine resource allocation of jobs based on estimations of the job completion time and the future resource availability. [**?**] employs a machine learning algorithm to predict available resources to minimize job deadline misses. However, these works do not consider stragglers which might severely prolong the job execution time.

Both proactive and reactive strategies have been proposed for mitigating stragglers [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. All of these works aim to reduce execution time of jobs without considering job deadlines. Different from these works, we jointly improve the PoCD and reduce the cost resulting from speculative/duplicate task execution, and find the appropriate number of speculative attempts for each straggler.

In addition, a number of works present techniques to avoid re-executing the work done by the original attempts (stragglers) [**?**], [**?**], [**?**], [**?**]. The basic idea is to checkpoint running tasks periodically and make speculative tasks start from the checkpoints. Our proposed Speculative-Resume strategy uses a similar idea. However, SRQuant uses an analytical model to determine the number of extra attempts to launch, and LASER combines DNN with MapReduce to estimate task execution times and determine the number of extra attempts to launch.

Due to the strong power of non-linear classification and regression, DNN has attracted a great deal of attention [**?**], [**?**], [**?**], [**?**], [**?**]. [**?**] and [**?**] present methods for utilizing MapReduce parallel processing framework to accelerate training for neural networks. To the best of our knowledge, this paper is the first to use DNN for straggler mitigation in MapReduce jobs.

III. Background

Consider M jobs submitted to a datacenter, denoted by $i = 1, 2, \ldots, N$. Each job i is associated with a deadline D_i

and consists of N_i independent tasks. Job i successfully meets its deadline if all its N_i tasks are completed before time D_i. The tasks with completion times exceeding D_i are known as the stragglers. Existing approaches mitigating stragglers often involve speculative execution, i.e., launching multiple parallel attempts of each straggler. Let r_i be the number of speculative attempts that are launched for each job i task. Thus, the task is completed if one of its $r_i + 1$ attempts (one original attempt plus r_i speculative attempts) is finished. We denote the execution time of attempt k of job i's task j as $T_{i,j,k}$. Thus, job completion time T_i and task completion time $T_{i,j}$ equal:

$$T_i = \max_{j=1,\ldots,N_i} T_{i,j}, \text{ where } T_{i,j} = \min_{k=1,\ldots,r_i+1} T_{i,j,k}, \ \forall j. \quad (2)$$

To optimize the speculative execution strategies using a model-based approach, Pareto distribution is often selected to model the execution times of tasks [**?**], and is used in [**?**], [**?**], [**?**] to analyze the straggler problem. Following these papers, we consider the execution time $T_{i,j,k}$ of each attempt follows a Pareto distribution with parameters $(t_{\min}, \beta)$, where $t_{\min}$ is the minimum execution time and β is the exponent. Different attempts are assumed to have independent execution time distributions.

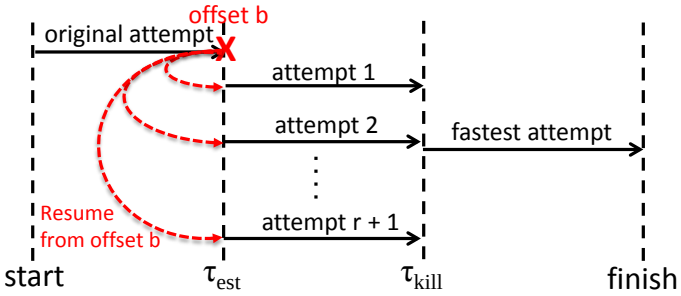


Fig. 2. Speculative-Resume Strategy.

The Speculative-Resume strategy launches one attempt of each task at the beginning. The attempt execution time is estimated at time τ_{est}. If the estimated execution time exceeds D, the original attempt is identified as a straggler. It is then terminated, and $r+1$ new (speculative) attempts are launched to replace the straggler. These speculative attempts resume data processing from the point left by the original attempt and do not repeat the work that has been completed. The progress scores of all attempts are then checked at time τ_{kill}, and the attempt with the smallest estimated completion time (or equivalently, highest progress score) is kept running while other attempts are terminated. Figure 2 illustrates the Speculative-Resume strategy [**?**] when a task is detected as a straggler. The processed byte offset of the original attempt at τ_{est} is b. Thus, speculative attempts are launched at τ_{est} start to continue processing data from byte offset b.

IV. Proposed Strategies

In this section, we present two strategies, Speculative-Resume with Quantitative Analysis (SRQuant), and Speculative-Resume with DNN (LASER).

A. SRQuant

We start with the Speculative-Resume with Quantitative Analysis (SRQuant) strategy. We first define PoCD and quantify PoCD for arbitrary jobs based on our analytical model (and assumptions) for job execution and the Speculative-Resume strategy. Finally, we present an iterative search algorithm for determining the appropriate number of extra attempts to launch for each straggler. In the following, the average progress of the original attempts at time τ_{est} is denoted as φ_{est}.

Definition We define PoCD as the probability that a job completes before its deadline.

We derive PoCD in closed-form in Theorem 1, and present execution time analysis in Theorem 2. We present only the basic ideas of the proofs here, and refer the reader to [**?**] for more details.

Theorem 1. *Under Speculative-Resume strategy, the PoCD for a job with N tasks and r speculative copies per task is given by:*

$$R(r) = \left[1 - \frac{(1-\varphi_{est})^{\beta\cdot(r+1)} \cdot t_{min}^{\beta\cdot(r+2)}}{D^{\beta}\cdot(D-\tau_{est})^{\beta\cdot(r+1)}}\right]^N. \quad (3)$$

Proof. The basic idea is that we first derive the probability of a task completing before the deadline, and then derive PoCD by finding the probability of N tasks finishing before the deadline. □

Theorem 2. *Under Speculative-Resume strategy, the expected execution time of a job is $E_r(T)$, which equals $N\cdot E(T_j)$. The $E(T_j)$ is the expected execution time of a task, and it equals*

$$E(T_j|T_{j,1}\leq D)\cdot P(T_{j,1}\leq D) + E(T_j|T_{j,1}>D)\cdot P(T_{j,1}>D), \quad (4)$$

where

$$P(T_{j,1}>D) = 1 - P(T_{j,1}\leq D) = \left(\frac{t_{min}}{D}\right)^{\beta} \quad (5)$$

$$E(T_j|T_{j,1}\leq D) = \frac{t_{min}\cdot D\cdot\beta\cdot(t_{min}^{\beta-1} - D^{\beta-1})}{(1-\beta)\cdot(D^{\beta} - t_{min}^{\beta})} \quad (6)$$

$$E(T_j|T_{j,1}>D) = \tau_{est} + r\cdot(\tau_{kill} - \tau_{est}) \quad (7)$$

$$+ \frac{t_{min}\cdot(1-\varphi_{est})^{\beta\cdot(r+1)}}{\beta\cdot(r+1)-1} + t_{min}. \quad (8)$$

Proof. The basic idea is that we derive machine running time of a task by considering if the execution time of the original attempt is larger than D or not. If execution time is no more than D, there is no need to launch extra attempt, and the machine running time equals the execution time of the original attempt. If the execution time is larger than D, the task machine time is the summation of the machine running times of extra attempts killed at τ_{kill} and the machine running time of the attempt that successfully finishes. The job's machine running time is obtained by adding the machine running times of N tasks. □

To exploit the tradeoff between R_r and $E_r(T)$, we jointly consider PoCD and machine running time. More precisely, we would like to maximize the following utility function $U(r)$.

$$\max \quad U(r) = \log(R(r) - R_{min}) - \theta \cdot C \cdot E_r(T), \quad (9)$$

$$\text{s.t.} \quad r \geq 0, \quad (10)$$

$$\text{var.} \quad r \in \mathbb{Z}, \quad (11)$$

where C is the usage-based VM price per unit time, and θ is a tradeoff factor to balance the PoCD $R(r)$ and execution cost $C \cdot E_r(T)$. Here, R_{min} is a minimum PoCD that all jobs must achieve.

Algorithm 1 SRQUANT SCHEDULING ALGORITHM

```
U_max = −∞
r_m = 0
for r = 0; r≤r_max; r++ do
  if U(r) > U_max then
    U_max = U(r)
    r_m = r
  end if
end for
for task : Tasks do
  if T_j > D then
    Launch r_m + 1 attempts for task
  end if
end for
```

In the SRQUANT SCHEDULING ALGORITHM, we first iteratively search for r from 0 to $r_{\max}$ to get the maximum value of $U(r)$, where $r_{\max}$ is the maximum number of extra attempts that can be launched. Given $U(r_m) \geq U(r),\ \forall r = 0, ..., r_{\max}$, the algorithm launches $r_m + 1$ attempts for each straggler, and kills the original task (straggler). The pseudocode is shown in Algorithm 1.

B. LASER

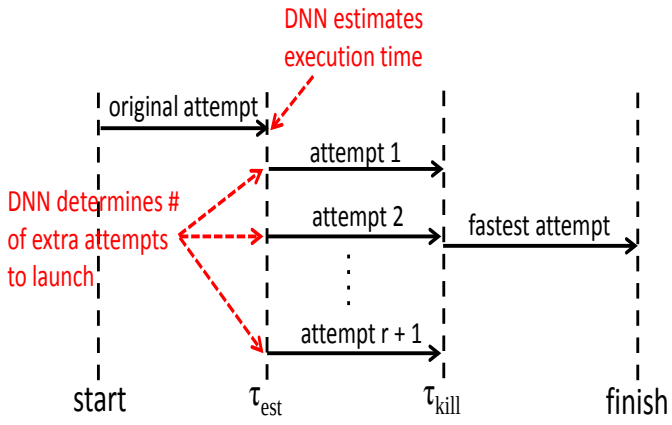


Fig. 3. An illustration figure of LASER.

In this subsection, we present the Speculative-Resume with DNN (LASER) strategy. LASER consists of two deep neural networks, i.e., a deep neural network for the task execution time estimation, and a deep neural network for determining the number of extra attempts to launch for each straggler.

At τ_{est}, instead of using the linear estimation as in SRQuant, LASER estimates the execution time of each task using a DNN. If the task execution time is larger than the job deadline, extra $r + 1$ attempts are launched. LASER determines the number of extra attempts to launch for a straggler through the DNN based on the job execution time, cost, and tradeoff factor (θ) between the job execution time and the cost. Figure 3 illustrates the LASER strategy.

In the LASER SCHEDULING ALGORITHM, we first iteratively search for r. If the estimated job execution time T_i is no more than D, then $U(r) = 1 - \theta \cdot C \cdot M_i$, where the 1 represents the fact that the job is estimated to complete before D, and M_i is the estimated machine running time. If T_i is larger than D, then $U(r) = 0 - \theta \cdot C \cdot M_i$, where the 0 represents the fact that the job is estimated not to complete before D. Given $U(r_m) \geq U(r),\ \forall r = 0, ..., r_{\max}$, the algorithm launches $r_m + 1$ extra attempts for each straggler, and kills the original task (straggler). The pseudocode is shown in Algorithm 2.

Algorithm 2 LASER SCHEDULING ALGORITHM

```
U_max = −∞
r_m = 0
for r = 0; r≤r_max; r++ do
  if T_i≤D then
    U(r) = 1 − θ·C·M_i
  else
    U(r) = 0 − θ·C·M_i
  end if
  if U(r) > U_max then
    U_max = U(r)
    r_m = r
  end if
end for
for task : Tasks do
  if T_j > D then
    Launch r_m + 1 extra attempts for task
  end if
end for
```

V. IMPLEMENTATION

We implement SRQuant and LASER on Hadoop, which consists of a central Resource Manager (RM), Application Masters (AMs) for each application, and Node Manager (NM) for each node in the system. For each application, its AM negotiates resources from the RM and works with the NMs to execute and monitor the application's tasks. Our scheduling algorithms are implemented in the AM to calculate r at time τ_{est}.

A. Implementing SRQuant

At τ_{est}, AM estimates the execution time of each task. The estimation strategy of Hadoop assumes a task starts

TABLE I
DNN WITH DIFFERENT NUMBER OF HIDDEN LAYERS.

	Task execution time			Job execution time			Machine running time		
Number of hidden layers	1	3	5	1	3	5	1	3	5
Mean of test losses (sec)	47	34	33	37	49	50	117	253	249
STD of Test losses	78	66	67	64	65	65	94	197	195
Mean of true values(sec)	226	226	226	249	249	249	1640	1640	1640
Mean of test losses/Mean of true values ($\times 100\%$)	20.8%	15.0%	14.6%	14.8%	19.7%	20.0%	7.1%	15.4%	15.2%
Elapsed time per iteration (msec)	1.3	4.8	7.3	1.2	4.4	6.4	1.1	4.4	6.9

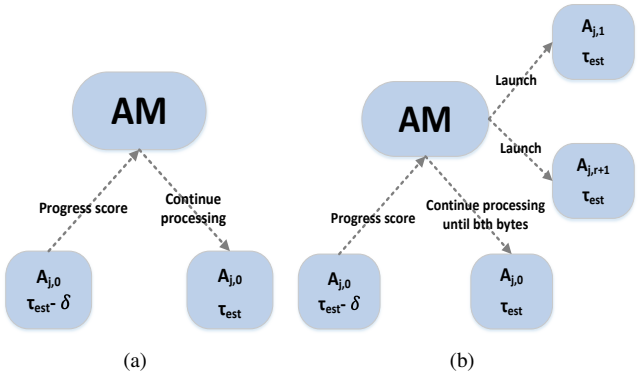


Fig. 4. Implementation of the SRQuant strategy, where $A_{j,r}$ is the rth attempt of the task j.

TABLE II
FEATURES DIRECTLY COLLECTED FROM HADOOP JOBS.

Feature	Description
$location$	Location host
$taskID$	Identifier of task
φ_j	Task progress score
rec_{read}	Amount of records read
rec_{write}	Amount of records written
$Byte_{read}$	Amount of bytes read from HDFS
$Byte_{write}$	Amount of bytes written to HDFS
t_{cpu}	Amount of CPU consumed time
t_{jvm}	Amount of time for launching JVM
t_{gc}	Amount of time for garbage collection
τ_{est}	The time of estimating the task execution time

running right after it is launched, and asserts that the task execution time equals τ_{est} dividing by the progress score (φ_j). However, according to our observations, this is not strictly true, especially in environments with high resource contention wherein JVM startup time is significant and cannot be ignored. We consider the task execution time is the sum of the amount of time used for launching the JVM (t_{jvm}) and the amount of time used for processing data. At τ_{est}, t_{jvm} is known, and the amount of time for processing data is estimated by $(\tau_{est} - t_{jvm})/\varphi_j$.

Considering that a task sends its progress score to the AM every 3 seconds, a task reports the number of bytes processed along with its progress score to the AM at time $\tau_{est} - \delta$, where δ is a user-selectable parameter with a default value of 3 seconds. The AM then estimates the execution time of each task. If the estimated execution time is no more than D, no extra attempt will be launched, as shown in Figure 4(a). Otherwise, the original attempt is killed after processing b bytes, where b is the number of bytes processed during the amount of time required to launch the JVM. $r + 1$ attempts will be launched, and these attempts start to process data from byte offset b, as shown in Figure 4(b).

B. Implementing LASER

As mentioned earlier, LASER uses two DNNs: one for estimating the (remaining) task execution time, and one for estimating the job execution time and the machine running time. In following, we describe details from three steps, i.e., features collected, pre-processing, and combining DNNs with Hadoop.

1) Features collected: To collect features for training the DNN used for estimating the task execution time, we run 2000 jobs, and collect 10 features of each task at τ_{est}. The features collected are shown in Table II.

To estimate the job execution time and machine running time, we run Speculative-Resume with different r values, and collect the 10 features of each task at τ_{est} (shown in Table II). Also, we need three more features, i.e., τ_{kill}, D, and φ_i, where φ_i represents the job progress score.

2) Preprocessing: If the task has not started to process data at τ_{est}, features: φ_j, rec_{read}, rec_{write}, $Byte_{read}$, $Byte_{write}$, t_{cpu}, t_{gc}, are all zeros. In order to avoid NaNs in weights and bias and let imputed values close to 0, we generate random numbers uniformly between 0 and 1 to replace the zeros.

3) Combining DNNs with Hadoop: After training the DNNs, we add DNNs with values for weights and biases, the parameters of the parametric ReLU functions from training, into the AM. As in SRQuant, a task reports the number of bytes processed along with its progress score to AM at time $\tau_{est} - \delta$. Different from SRQuant, LASER computes the r and estimates the execution time of each task using DNNs. If the task execution time is larger than D, the original attempt is killed after processing b bytes. Extra $r + 1$ attempts will be launched, and these start to process data from byte offset b.

VI. EXPERIMENT RESULTS

A. Setup

We prototype the SRQuant and LASER on a cloud testbed consisting of 80 nodes. Each node has 8 vCPUs and 2GB memory. The physical servers are connected to a Gigabit Ethernet switch and the link bandwidth is 1Gbps. We evaluate two strategies by using Map phases of the classic benchmark

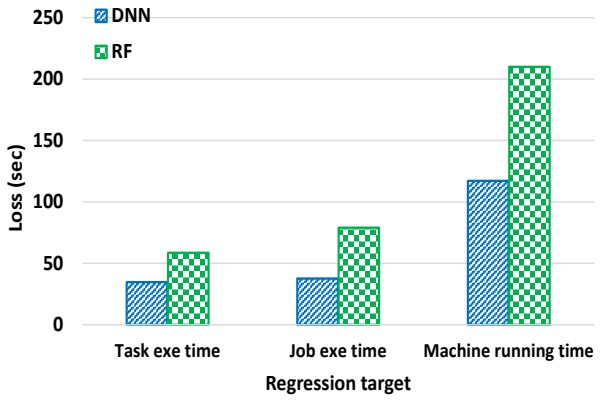


Fig. 5. Comparing DNN with the Random Forest regression algorithm.

WordCount with a 1.2GB workload for WordCount from Wikipedia. To emulate a realistic cloud infrastructure with resource contentions, background applications are run in the host OS of the physical servers; these execute computation tasks at random times and inject background noise. We observe that the task execution time measured on our testbed follows a Pareto distribution with an exponent $\beta = 1.7 < 2$.

B. Tune the DNNs

In this subsection, we show the results of tuning DNNs by changing the number of hidden layers, and present the results of comparing DNNs with a classic regression algorithm called Random Forests (RF) [**?**]. RF constructs a number of decisions trees, with each tree being a weak learner. The final output is the average prediction over all trees. The loss function of DNNs is the least absolute deviations. Table I shows the statistics when employing different numbers of hidden layers for DNNs. The test loss is the absolute difference between the estimated value and the true value. Based on results, the mean of test losses of the task execution time decreases as the number of hidden layers increases, and the means of tests losses of the job execution time and machine running time increase as the number of hidden layers increases. Also, the table shows that the lowest mean of test losses is achieved when the number of hidden layers are 5, 1, and 1 for the task execution time, the job execution time, and the machine running time estimations, respectively. For the task execution time estimation, as the number of hidden layers changes from 3 to 5, the elapsed time per iteration increases by 50%, and the estimation accuracy increases only by 1%. So, in the following experiments, we employ 3 hidden layers for the task execution time, and one hidden layer each for the job execution time and the machine running time estimations, respectively.

Figure **??** shows the comparisons between the DNN and the Random Forests regression algorithm. For RF, we set the number of trees in the forest to be 1000, and the criterion of regression to be the mean squared error. The results show that the DNN outperforms RF by 42%, 52%, and 44% for the task execution time, the job execution time, and the machine running time estimations, respectively.

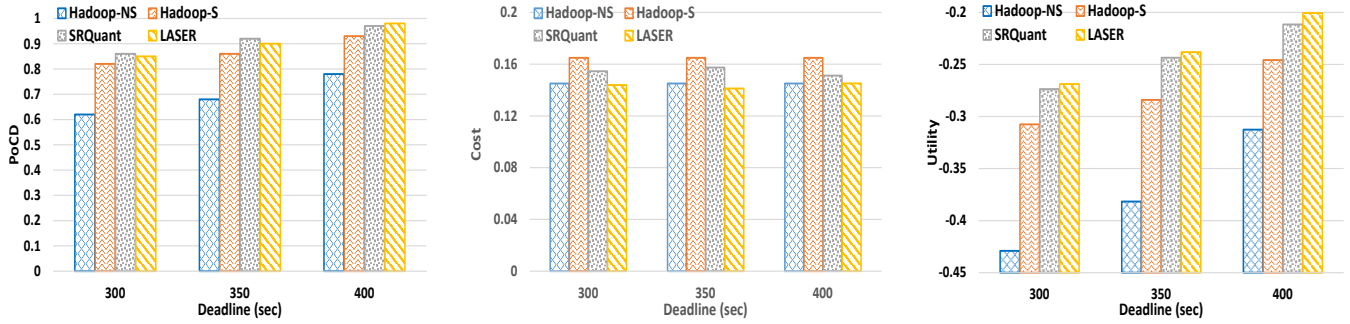
C. Comparison of different strategies

In following, we compare Hadoop without Speculation (Hadoop-NS), Hadoop with Speculation (Hadoop-S), SRQuant, and LASER with respect to PoCD, cost and total utility. We execute 100 MapReduce jobs on our testbed with each job consisting of 10 tasks. The PoCD is measured by calculating the percentage of jobs completed before their deadlines, and the cost by the average job running time (i.e., VM time required), assuming a fixed price per unit VM time that is obtained by Amazon EC2 average spot price (C). In all experiments, we set $\theta \cdot C = 0.0001$.

Figure **??** and Figure **??** show the effect of changing deadlines on PoCD and cost, respectively. We set $\tau_{est} = 120$, $\tau_{kill} = 200$, and $R_{min} = 0.1$. As deadline increases, more jobs can finish before the deadline, and thus PoCDs increase. Also, as deadline increases, the cost of Hadoop-NS and Hadoop-S remain the same, since those strategies are deadline oblivious, in the sense that they are not adaptive to jobs with different deadlines. Our results show that the two strategies considered in this paper have higher PoCD compared with Hadoop-NS and Hadoop-S. In terms of PoCD, LASER can outperform Hadoop-NS and Hadoop-S by up to 40% and 7%, respectively. In terms of cost, LASER can outperform Hadoop-S and SRQuant by up to 18% and 13%, respectively. The cost saving of LASER, relative to SRQuant, results from launching fewer extra attempts. Figure **??** shows the impact of changing deadlines on net utility. Our results show that our two strategies outperform Hadoop-NS and Hadoop-S by up to 43%.

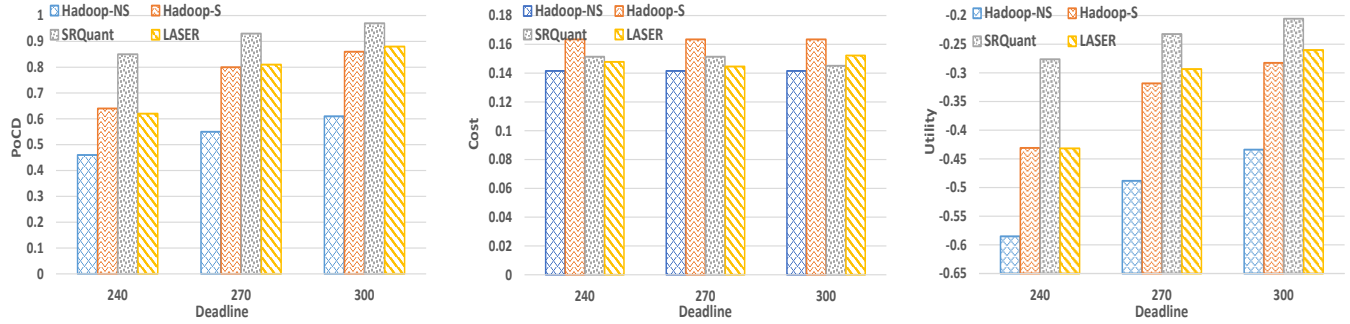
Comparing Figure **??** and Figure **??** shows the effect of decreasing τ_{est}, τ_{kill}, and D on PoCD and cost. To obtain Figure **??**, we set $\tau_{est} = 80$, $\tau_{kill} = 160$, and $R_{min} = 0.1$. A comparison of Figure **??** with Figure **??** shows that SRQuant could achieve higher PoCD by decreasing τ_{est}. Also, Figure **??** shows the PoCD of LASER is lower than SRQuant's. The reason is that the DNN estimation cares little about the outliers, which are the values that are far away from the average, and those outliers are important for highly deadline-critical jobs to meet desired deadlines. Figure **??** uses data from the test set. The test set is a set of data used to assess the strength of the prediction. Figure **??** shows that when the true value is around the mean (249), the difference between the estimated value and the true value is small. As the difference between the mean and the true value increases, the DNN estimation becomes more inaccurate. In terms of PoCD, SRQuant outperforms Hadoop-NS, Hadoop-S, and LASER by up to 89%, 31%, and 30%, respectively. In terms of cost, SRQuant and LASER can outperform Hadoop-NS by up to 13%. Figure **??** shows SRQuant has the largest utility, and it outperforms other strategies by up to 50%.

Remark. Compared to the DNN-based method (LASER), the model-based optimization method (SRQuant) can achieve better performance in mitigating stragglers. This is because DNN is not able to accurately estimate the execution time of a straggler, and the predicted r is smaller than needed. However,



(a) (b) (c)

Fig. 6. Comparisons of different strategies with large τ_{est}, τ_{kill}, and D.



(a) (b) (c)

Fig. 7. Comparisons of different strategies with small τ_{est}, τ_{kill}, and D.

Fig. 8. The comparison between true values and estimated values.

LASER is better than Hadoop-NS and Hadoop-S, in terms of PoCD and Cost, and it can be easily adapted to various cluster environments. LASER does involve training overhead, but such overhead can be mitigated by training with history logs before the application is submitted.

VII. Conclusion

In this paper, we consider a Speculative-Resume strategy for mitigating stragglers and meeting application deadlines in a MapReduce framework. Two distinct strategies, SRQuant and LASER, are proposed to identify stragglers and to optimize the Speculative-Resume strategy. While SRQuant relies on a model-based optimization approach to find optimal operating parameters, LASER provides an refreshing perspective by making use of Deep Learning algorithms for strategy optimization. The two strategies are implemented on Hadoop MapReduce and compared side-by-side in a number of experiments on an 80-node cloud testbed. Our experiment results validate the effectiveness of both strategies in meeting application deadlines in a cost-effective fashion. The results show that our two strategies outperform Hadoop without speculation (Hadoop-NS) and with speculation (Hadoop-S) by up to 89% in PoCD and 13% in cost.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *OSDI*, vol. 8, no. 4, 2008, p. 7.

[3] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri." in *OSDI*, vol. 10, no. 1, 2010, p. 24.

[4] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 185–198.

[5] 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.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015.

[7] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, Aug. 2013.

[8] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin, "Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. ACM, 2013, pp. 6:1–6:17.

[9] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. balmin, "Flex: A slot allocation scheduling optimizer for mapreduce workloads," in *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, ser. Middleware '10. Springer-Verlag, 2010, pp. 1–20.

[10] D. Cheng, J. Rao, C. Jiang, and X. Zhou, "Resource and deadline-aware job scheduling in dynamic hadoop clusters," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 956–965.

[11] S. Alamro, M. Xu, T. Lan, and S. Subramaniam, "Cred: Cloud right-sizing to meet execution deadlines and data locality," in *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*. IEEE, 2016.

[12] D. Cheng, J. Rao, C. Jiang, and X. Zhou, "Resource and deadline-aware job scheduling in dynamic hadoop clusters," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '15. IEEE Computer Society, 2015, pp. 956–965.

[13] 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.

[14] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, and S. Wu, "Maestro: Replica-aware map scheduling for mapreduce," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. IEEE, 2012, pp. 435–442.

[15] J. Rosen and B. Zhao, "Fine-grained micro-tasks for mapreduce skew-handling," *White Paper, University of Berkeley*, 2012.

[16] Y. Wang, W. Lu, R. Lou, and B. Wei, "Improving mapreduce performance with partial speculative execution," *Journal of Grid Computing*, vol. 13, no. 4, pp. 587–604, 2015.

[17] H. Wang, H. Chen, Z. Du, and F. Hu, "Betl: Mapreduce checkpoint tactics beneath the task level," *IEEE Transactions on Services Computing*, vol. 9, no. 1, pp. 84–95, 2016.

[18] J.-A. Quiane-Ruiz, C. Pinkel, J. Schad, and J. Dittrich, "Rafting mapreduce: Fast recovery on the raft," in *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011, pp. 589–600.

[19] Y. Wang, H. Fu, and W. Yu, "Cracking down mapreduce failure amplification through analytics logging and migration," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 261–270.

[20] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, Jul. 2006.

[21] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1915–1929, Aug. 2013.

[22] D. Xia, B. Wang, H. Li, Y. Li, and Z. Zhang, "A distributed spatial-temporal weighted model on mapreduce for short-term traffic flow forecasting," *Neurocomput.*, vol. 179, no. C, pp. 246–263, Feb. 2016.

[23] R. Gu, F. Shen, and Y. Huang, "A parallel computing platform for training large scale neural networks," in *2013 IEEE Conference on Big Data*. IEEE, 2013.

[24] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "Grass: trimming stragglers in approximation analytics," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 289–302.

[25] 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, 2017.

[26] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized speculation-aware cluster scheduling at scale," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 379–392, 2015.

[27] M. Xu, S. Alamro, T. Lan, and S. Subramaniam, "Optimizing speculative execution of deadline-sensitive jobs in cloud," *SIGMETRICS Perform. Eval. Rev.*, 2017.

[28] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001.