An Approach for Modeling and Ranking Node-level Stragglers in Cloud Datacenters

Xue Ouyang[1,2], Peter Garraghan[1], Changjian Wang[2], Paul Townend[1], Jie Xu[1]

School of Computing[1]
University of Leeds
Leeds, UK
{scxo,p.m.garraghan,p.m.townend,j.xu}@leeds.ac.uk

Parallel and Distributed Laboratory[2]
National University of Defense Technology
Changsha, China
c_j_wang@yeah.net

Abstract— **The ability of servers to effectively execute tasks within Cloud datacenters varies due to heterogeneous CPU and memory capacities, resource contention situations, network configurations and operational age. Unexpectedly slow server nodes (node-level stragglers) result in assigned tasks becoming task-level stragglers, which dramatically impede parallel job execution. However, it is currently unknown how slow nodes directly correlate to task straggler manifestation. To address this knowledge gap, we propose a method for node performance modeling and ranking in Cloud datacenters based on analyzing parallel job execution tracelog data. By using a production Cloud system as a case study, we demonstrate how node execution performance is driven by temporal changes in node operation as opposed to node hardware capacity. Different sample sets have been filtered in order to evaluate the generality of our framework, and the analytic results demonstrate that node abilities of executing parallel tasks tend to follow a 3-parameter-loglogistic distribution. Further statistical attribute values such as confidence interval, quantile value, extreme case possibility, etc. can also be used for ranking and identifying potential straggler nodes within the cluster. We exploit a graph-based algorithm for partitioning server nodes into five levels, with 0.83% of node-level stragglers identified. Our work lays the foundation towards enhancing scheduling algorithms by avoiding slow nodes, reducing task straggler occurrence, and improving parallel job performance.**

Keywords- Stragglers; Node Performance; Clusters; Tracelog Data Analysis; Modeling; Ranking

I. INTRODUCTION

Modern day Cloud datacenters are composed of thousands of heterogeneous server nodes to provide computing services globally [1]. These nodes are composed of different characteristics including resource capacity (CPU, memory, disk, etc.), architecture, and operational age. These physical heterogeneities combined with changing resource utilization patterns and multi-tenancy [2] result in diverse task execution performance at given time for each node within the system.

The response time of services is critical for both Cloud providers and users. For providers, jobs which take longer to complete result in system performance degradation and increased resource waste, causing financial loss and decreased user satisfaction. For users, additional response time results in potential Quality of Service (QoS) [3] violations with respect to timing constraints. Parallel computing models such as MapReduce [4] and Hadoop [5] divide jobs into multiple tasks for performing a subset of computation. Although these models attempt to evenly split computation based on input data size to achieve similar completion times across all tasks, in practice tasks will exhibit different execution durations as a result of being assigned to different nodes. This is important within the context of *task-level stragglers,* defined as parallel tasks which experience abnormally longer execution in comparison to sibling ones, and *node-level stragglers* defined as server nodes that exhibit poor performance regarding task execution, leading to higher task straggler occurrence and susceptibility. In this paper, *node execution performance* is defined as the measurement of effective task execution within a node in the presence of task stragglers.

This straggler problem for parallel jobs increasingly manifests itself within distributed systems as scale increases [6], significantly impedes overall parallel job completion, and debilitates service response time, causing economic loss and degraded system performance and availability. There have been concentrated efforts toward addressing these issues, ranging from task straggler analysis [6][16] to mitigation [9-15][18]. Effective approaches for the latter include server blacklisting [17], which attempts to assign tasks to nodes that do not exhibit poor performance in order to reduce task straggler occurrence, and speculative execution [5] which launches replica copies for task stragglers on different nodes in an attempt to outpace the original task. However, these approaches assume that slow nodes are static [9][13] (i.e. server execution performance will constantly be poor), while in practice server execution performance can be transient due to diverse resource utilization driven by user demand. Such a characteristic is important to consider in order to inform the scheduler concerning alterations to node performance at run-time. Furthermore, there is presently a lack of comprehensive methods for modeling slow nodes that are capable of capturing node execution performance that can be applied generally to Cloud datacenters.

This paper proposes a method for modeling and ranking node performance to determine the likelihood of straggler manifestation within Cloud datacenters. This is achieved through analyzing historical data within a configurable time window. The outputs of such work can be further integrated into straggler mitigation techniques such as intelligent speculation [10]. Our contributions are summarized as follows:

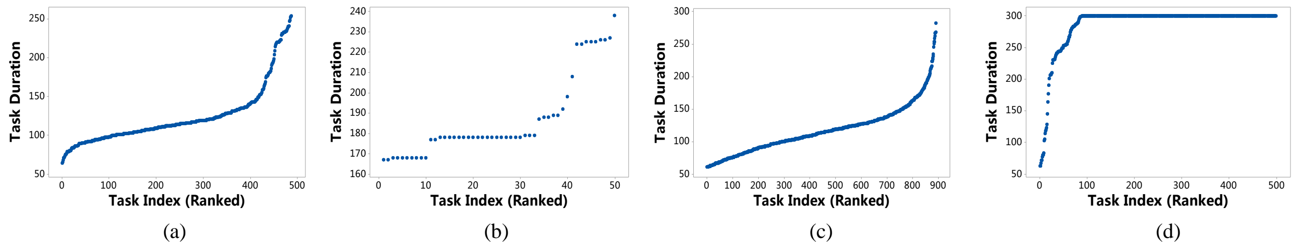


Figure 1. Task duration pattern of (a) job 6336907144, (b) job 6336143870, (c) job 6335538471, (d) job 6301092750 from Google trace

- *A method for modeling node execution performance.* We describe in detail a model framework that can analyze nodes' task execution performance in the presence of stragglers, including execution data extraction and normalization, distribution fitting, indicator selection and ranking.
- *An approach for node-level straggler detection.* Through ranking, our framework can classify server nodes into different performance levels and identify the weakest node-level stragglers within a system. This is important for improving blacklisting and speculation efficiency when dealing with stragglers.
- *New insights into node-level straggler manifestation within production Cloud datacenters.* We have evaluated our method using production data from a Google datacenter [7][8]. From our results, we are able to gain new insight demonstrating the transient nature of execution performance within datacenter nodes.

The paper is structured as follows: Section 2 presents the straggler problem background and related work; Section 3 illustrates the node performance modeling and ranking framework; Section 4 discusses the case study data and the methods of extracting MapReduce task information, while Section 5 presents the analysis findings using the model; Section 6 discusses conclusions and future work.

II. Related Work

Frameworks such as MapReduce decompose jobs into multiple tasks which are executed across numerous nodes in order to achieve better performance through parallelization. However, these frameworks still face challenging problems toward effective execution within large-scale clusters. The Long Tail problem [6] is one example of such a challenge, characterized as a small subset of parallel tasks performing much slower (defined as task-level stragglers) in comparison with other tasks in the same job, therefore incurring a significant delay towards final job completion. Figure 1 depicts task execution patterns within jobs of varying sizes from a Google production cluster [7][8]. It is observable that the Long Tail problem manifests within different sizes of jobs. For example, in Fig. 1(a-c) the slowest task is approximately 2.5 slower to complete in comparison with the average. Fig. 1(d) demonstrates job containers with no stragglers, with the majority of tasks completing in approximately the same time frame. Task-level stragglers can stem from numerous causes including hardware heterogeneity [10], resource contention [11], background network traffic [12], I/O discord [13], data skew [14] and OS or application-level related sources [15].

There are numerous works that analyze the impact of task-level stragglers on system performance from the application perspective: for example, Jeffrey et al. [6] demonstrate that the slowest 5% of completed requests are responsible for half of the total 99th percentile latency. That work also discusses the positive correlation between straggler probability and cluster size, concluding that the probability of longer latency increases within larger system scales.

Ananthanarayanan et al. [12] show that 80% of task stragglers have a uniform probability of delay between 150%-250% compared to the median task duration, with 10% exhibiting a delay 1000% greater than median task duration.

Garraghan et al. [16] analyze two production Cloud datacenters and demonstrate that less than 5% and 3% of task stragglers negatively impact between 35% and 59% of total batch jobs within two production Cloud datacenters. From these results they propose a task straggler identification system that combines both historical and online analytics.

There exist primarily two methods for mitigating task-level stragglers: server blacklisting [17] and speculative execution [5]. Server blacklisting is performed by the scheduler maintaining a blacklist of nodes within the cluster (such as detailed in Fuxi [23] or by modifying the mapred-site.xml for Hadoop). This results in tasks never assigned to such nodes until list removal. The effectiveness of this approach is dependent on correctly detecting faulty nodes for blacklisting, otherwise, the system capacity will be degraded due to false positives. Furthermore, current practice assumes that weak nodes are known by the system administrator, and that server performance will remain stable. This is not always the case, making it increasingly infeasible to conduct such manual configuration for clusters comprising thousands of nodes, and such static configuration is unable to capture accurate node performance.

Speculative execution [5] monitors the execution of each task and will launch speculative replicas for identified task stragglers with the assumption that it will complete prior to the original straggler. There exist numerous techniques which extend this method in terms of specified cases such as a heterogeneous nodes environment [10] and small jobs execution (less than 10 parallel tasks)[18]. While these works are effective in minimizing the impact of task stragglers within the system, they are mainly focused on selecting the best task candidates to make replicas and ignore the impact of poor node-level straggler performance. This is particularly important in avoiding the scheduling of speculative replicas onto these straggler nodes, leading to decreased likelihood of the replica completing prior to the task straggler. In this case,

speculation results in limited improvement in job performance as well as increased resource overhead.

In terms of determining the effective placement of replicas on nodes, Chen et al. [13] consider both data locality and data skew to develop a cost benefit model based on the cluster load. Yadwadkar et al. [9] develop a system to proactively avoid stragglers by analyzing a production trace and perform regression using node level statistics. This system periodically produces correlations between node level status and task execution times in the form of decision trees, enhancing scheduler policy making when determining which node to run the replica in order to minimize overall job execution.

While these methods leverage the concept of applying data analytics to identify weak nodes effectively, they each assume that poor node performance is a static characteristic that remains constant within the cluster, and is determined by node capacity. In reality, the execution performance of a node fluctuates over time due to factors such as resource contention level, workload heterogeneity and user demand. This has been studied in [2] demonstrating that there exists dynamicity in system conditions within clusters driven by diverse users and tasks execution profiles.

III. Framework for Modeling and Ranking Node Execution Performance

Before introducing our framework, it is necessary to define key concepts in our research. The *scheduler* is responsible for assigning *jobs* into the Cloud datacenter which comprise M server nodes. Job j contains N subtasks running in parallel on multiple nodes where $task_j^i$ represents the i_{th} task belonging to job j ($i \leq N$). It is assumed that subtasks exhibiting non-straggler behavior share similar response times (for example, all map tasks of a certain MapReduce job have similar execution lengths due to identical block size).

The duration of $task_j^i$ is defined as the time between scheduling and completion, represented as t_j^i. Task stragglers refers to those with $t_j^i > Th * \overline{t_j}$, where $\overline{t_j}$=$Avg(t_j^1, t_j^2, \ldots, t_j^n) = \sum_{i=1}^{n} t_j^i / n$. *Th* represents a specific threshold for task straggler identification. In most literature pertaining to stragglers, this threshold is typically configured to a value of 1.5 (i.e. tasks whose execution is 50% greater than the average execution of tasks within the same job) [10][14][18], while [19] proposes a dynamic threshold calculation algorithm to define task stragglers in accordance to workload type and cluster resource usage.

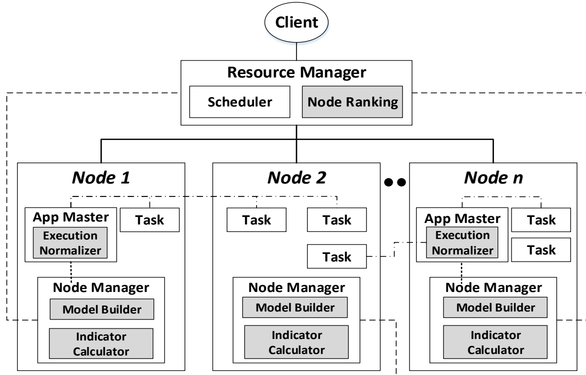


Figure 2. Framework architecture

Algorithm 1. Operation of Model Framework

Inputs:

Quintuple $e = < t_s,\ t_e, ID_{task}, ID_{job}, ID_{machine} >$

- t_s , t_e – task start / end time
- ID_{task} , ID_{job} , $ID_{machine}$ – task ID, job ID this task belongs to, and machine ID this task runs on

1. **For each** e from filtered trace data
2. μ = ***NormalizedExecutionValue*** (e)
3. $\omega = < ID_{task}, ID_{job}, ID_{machine}, \mu >$
4. **For each** ω
5. $\varphi = < ID_{machine}, \mu >$
6. **For each** $ID_{machine}$
7. f = ***BestFitDistribution*** (φ)
8. I = ***TargetAbilityIndicator*** (f)
9. ***Rank*** (I)

A. Overall Framework

The framework consists of four steps as shown in Algorithm 1 and Figure 2: (i) *Execution Normalizer* within *Application Master* calculates normalized task duration respective to job execution, (ii) *Model Builder* within *Node Manager* is responsible for calculating the probability distribution models of execution performance for each node, (iii) *Indicator Calculator* is responsible for selecting statistical properties to indicate node ability in turns of parallel job execution, and (iv) the *Node Ranking* component will produce a rank order list of all system nodes to determine susceptibility to stragglers. This framework can be readily integrated into current 2-tier Cloud resource managers such as Yarn [24] as shown in Figure 2, leveraging data typically recorded in system operation, including task start and completion timestamps, job and task identifiers, and server node ID. Therefore, it results in minimal burden upon the system due to no need for additional monitoring and data extraction. The following sections introduce each of the phases in detail.

B. Task Duration Normalization

Node-level straggler detection is performed by studying task execution performance on a per node basis. However, as Cloud datacenters are composed of heterogeneous workload characteristics [2], it is challenging to directly compare job (by extension task) performance across the entire system. On the other hand, the relative task durations (compared to its job average) can be used to study a tasks' progress in comparison to its own siblings. As a result, the first step of the framework is to use z-score normalization across all tasks duration t_j^i to generate $\widetilde{t_j^i}$ by using equation (1).

$$\widetilde{t_j^i} = \frac{t_j^i - \overline{t_j}}{\sigma_j} \quad (1)$$

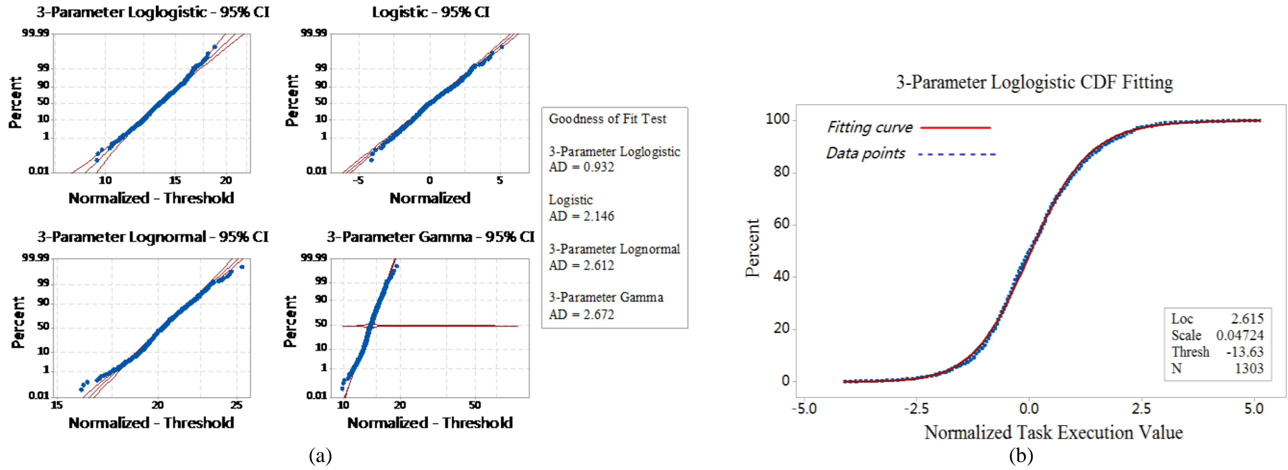


Figure 3. (a) Top four best fitting distribution for node 4820223869 (b) 3-parameter loglogistic CDF fitting

where $\sigma_j = stDev(t_j^1, t_j^2, \dots, t_j^n)$ is the standard deviation of t_j. The normalized value $\widetilde{t_j^i}$ reflects the relative speed of $task_j^i$ compared to other sibling tasks within job j. $\widetilde{t_j^i} > 0$ signifies t_j^i is larger than the average job execution time, therefore $task_j^i$ is a slow task within this job. A larger value of $\widetilde{t_j^i}$ represents more severe straggler behavior in $task_j^i$. This makes it possible to compare the performance of tasks irrespective of job heterogeneity.

For every node M, the normalized execution value for assigned tasks can be used to analyze its performance in terms of task execution ability. For example, if the majority of tasks assigned onto a node are always the slower ones (with a positive normalized value indicating longer execution than their own average) for different jobs, this reflects poor node execution performance compared to other nodes within the system. In contrast, if most of the normalized values assigned onto the node are negative, this represents that tasks assigned tend to execute faster, demonstrating strong node execution performance. Therefore, it is necessary to form a data file consisting of the following sextuples for each node in order to analyze its performance model:

$$< t_s,\ t_e, ID_{task}, ID_{job}, ID_{machine}, Normalized_t >$$

This records start time of a task, end time of a task, task ID, job ID, machine ID and task normalized value as the input to enable the analysis.

C. Best Fit Distribution for Node Execution Performance

The normalized task duration is used in order to construct probabilistic models of execution performance for tasks within each server node in order to ascertain the likelihood of task straggler occurrence.

There exist numerous Goodness of Fit (GoF) tests designed for different data characteristics including chi-square, Kolmogorov-Smirnov (KS) and Anderson-Darling (AD) [20, 21]. For our analysis, AD test is adopted as it places greater emphasis towards tailing data distributions. Figure 3 shows an example result of distribution fit for the node with ID 4820223869 in Google. In this case, 9 different distributions have been tested, including 3-parameter Lognormal, Normal, 2-parameter Exponential, 3-parameter Weibull, Smallest Extreme Value, Largest Extreme Value, 3-parameter Gamma, Loglogistic and 3-parameter Loglogistic. Figure 3 (a) lists the top four best fits, among which 3-parameter Loglogistic distribution represents the best accuracy due to a lower AD value (CDF fitting is given in Figure 3 (b)).

D. Target Indicator of Node Execution Performance

The distribution function constructed for each node is used to derive the statistical properties of node performance and susceptibility to task straggler behavior. The indicator used to capture this behavior includes the mean value, standard deviation, confidence interval, quantile points, and extreme value possibility, with each reflects different analysis objective. For example, standard deviation describes the stability of the node execution performance, while extreme value possibility represents the task straggler occurrence probability. Table 1 lists representative attributes and their corresponding meanings for reference.

Under cases when the confidence interval is chosen as the indicator of interest, it provides information that, for all tasks assigned onto this node, there is a confidence (e.g. 95%) to believe their normalized durations will fall within a specified interval. This is necessary to determine the optimal placement of tasks into nodes under the presence of stragglers; therefore in the following analysis, the indicator confidence interval will be adopted.

Table 1 Indicator Candidates and Corresponding Meanings

Indicator	Meaning
Mean Value	The possible normalized execution value for tasks assigned onto this node
Standard Deviation	The normalized task execution value on this specific machine is stable or random
Confidence Interval	The possible normalized execution value assigned will between a certain interval
Extreme Value Possibility	The task straggler possibility for this machine.
Quantile Value	Describes the normalized value for most tasks been assigned onto that specific node

E. Ranking

The final step of the framework is to determine a rank order of the indicator values to classify node performance in order to categorize and identify node-level stragglers. There exist numerous ranking algorithm candidates which can leverage indicators presented in Table 1. If value-type indicators have been chosen (i.e. mean, deviation, quantile points and extreme value possibility), the ranking is relatively straight forward. If an interval-type indicator has been chosen, such as confidence interval, a graph based ranking algorithm can be adopted.

In this paper, we use the P-Cores algorithm [25] to deal with confidence interval indicator by constructing a directed acyclic graph (DAG). The nodes in the DAG represent servers within the cluster. If [A1, B1] represents the confidence interval execution performance of Node 1 and [A2, B2] represents the confidence interval of execution performance of Node 2, there will be an edge from Node 1 to Node 2 only when the condition A2>B1 stands (as shown in Figure 4). The next step is to remove nodes that do not contain an outward edge (as they represent the weakest nodes within the graph) until there are no nodes remaining. The time on which the node is been deleted is subscribed as the level this machine should be classified to. In other words, level zero nodes represent the worst execution performance as they contain the largest confidence interval value, and were removed within the first iteration. P-cores only demonstrates one such ranking algorithm within this framework, and can be substituted by other means if required.

This framework can be easily integrated into current implementations such as Hadoop. As long as the ranking result has been generated, the only modification required for system implementation is to re-modify the node health checker condition. This allows for the core system scheduler code to remain unattached and the node performance can be easily adopted by existing scheduling algorithms.

IV. Case Study based on Production Tracelog Data

The proposed framework was applied to a Google datacenter publicly available at [7][8] to demonstrate its effectiveness. As different providers typically have bespoke methods for collecting and structuring produced system data, we will first detail the semantics, data formats and schema of the case study. We will then describe the filtering and sampling process.

The tracelog of the case study system comprises 29 days operation detailing job/task behavior within a Google cluster consisting of 12,583 server nodes that share a common cluster management system. Work arrives at this cluster in the form of jobs that comprises several tasks, and a task is represented as a Linux program that executes on a single node.

There are four tables within the tracelog that relate to our research objectives. The *machine events table* details server status (i.e. whether it has been added, removed, or modified within the cluster). The *job events* and *task events* tables record information pertaining to job/task status (un-submitted, pending, running, dead) expressed through recorded events

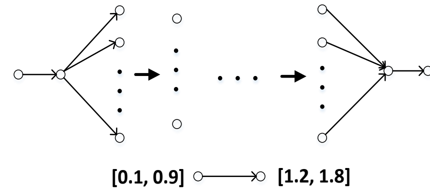


Figure 4. DAG edge example

(submit, schedule, kill, evict) at specific timestamps. The *task usage table* gives information of the start/end time of each individual task as well as the specific placement onto servers. Usually, all tasks within a job execute exactly the same binary code with the same options and resource request. The dataset is voluminous (approximately 400GB in size unzipped), and contains traces of 672,074 jobs composed of 25,228,174 tasks. As a result, it is important to filter out noisy information and properly decide suitable target jobs to conduct the analysis.

A. Data Pre-processing

It has been identified that the cluster contains numerous application types including batch, latency sensitive, gratis and system monitoring jobs. In our analysis, we focus on MapReduce jobs - a representative job type that containing subtasks which exhibit similar completion times. For example in Hadoop system, the default data block size settings is 64MB, and map tasks are automatically generated based on input data size; they will, therefore, have similar completion times. However, due to commercial confidentiality, Google does not reveal precise information concerning specific job types. In order to extract MapReduce job data, three filter conditions have been applied.

1) Identify parallel jobs

The first condition is to identify jobs which execute tasks in parallel. Task number is used to design this filter, and the ones that have more than two tasks submitted at the same time are been extracted out. This is possible by studying the timestamps of job and task submissions and completions, as well as using the right SQL query to select corresponding jobID with multiple taskIDs.

2) Determine production jobs

Tasks within the cluster are assigned priorities ranging between 0 and 11 for lowest and highest scheduling priority, respectively, indicated in job events table. According to documentation [8], production tasks including batch job processing and latency sensitive tasks are with priority from 2 to 9, monitoring are 10 to 11, and gratis tasks are 0 to 1.

3) Extract MapReduce jobs

Unlike the former two conditions that have explicit relating attributes, filtering out jobs that exhibit MapReduce characteristics used two additional hypotheses in our work. First, according to [8], the attributes "job name" and "logical job name" within the job events table, both of which are opaque base 64-encoded strings that have been hashed to hide sensitive information, can be used. Unique job names are generated by automated tools to avoid conflicts, however, the job names generated by different executions of the same program will usually have the same logical name. [8] points

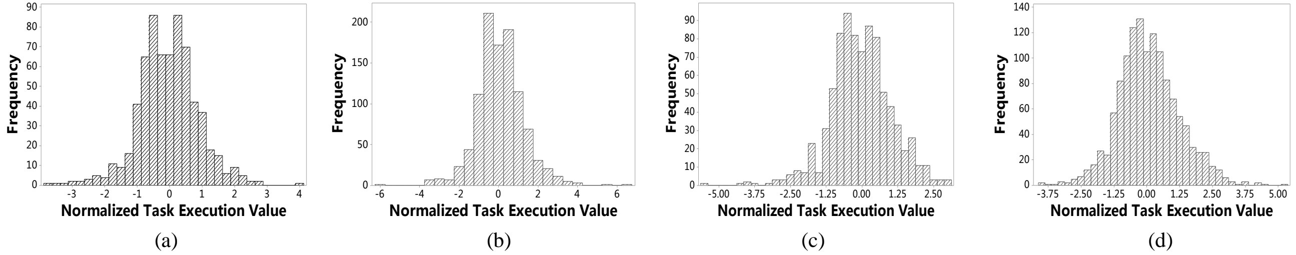


Figure 5. Normalized value frequency for machine (a) 672206, (b) 554297904, (c) 257336015, (d) 4820223869 from filtered trace data

out that MapReduce is an example of this kind of application that frequently generates unique job names with identical logical names. Second, the attribute named "username" can also assist towards identifying MapReduce jobs. Usernames in this trace represent services run on top of the Google Cloud cluster, and jobs executing under the same username are likely to be part of the same external or internal service. When a single program runs multiple jobs, such as master job and worker job spawned by the same MapReduce, those jobs will almost always run as the same user.

After filtering, the total target job size is reduced to 92,848 with 10,894,461 tasks for analysis. Importantly, since no biased selections have been conducted towards node type, the influence brought by eliminating additional tasks is applied equally to all nodes, therefore, there should be no imbalance that will lead to unreliable node performance result.

B. *Node Sampling*

As there are over 12,500 server nodes within the cluster, it is beneficial to perform sampling in order to conduct in-depth analysis of node execution performance that accurately reflects the general characteristics of the whole cluster. There exist four primary types of server nodes within this cluster, with each type reflecting different physical capacities in dimensions of CPU cores and RAM size as shown in Table 2. We conduct systematic sampling based on the number of tasks assigned to the nodes. The minimum value of the sample set that retains a 5% margin of error to the whole population is 132 after applying Minitab (statistical software similar to SPSS) sample size calculation function. By making a random selection from each server type to generate the corresponding number of nodes, we generate the final target set which consists nodes that remain the same server type proportion. Furthermore, in order to minimize the error brought by sampling, two different sets have been selected to perform both modeling and validation.

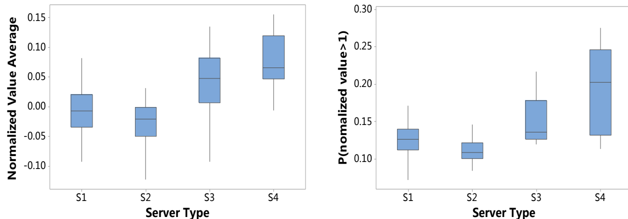
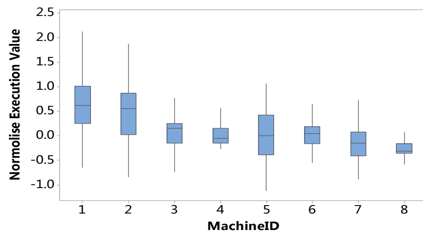


Figure 6. Box plot of (a) mean normalized value result (b) extreme value possibility result for each group in Google cluster



(c) Box plot of normalized execution value for each VM in experiment

V. MODEL EVALUATION

This section evaluates the framework's effectiveness for determining a node's performance and its susceptibility to straggler behavior, performed through experiments and data analytics.

A. *Distribution Modeling*

Figure 5 illustrates four examples of node performance distributions of individual server nodes when applying the proposed framework. Within the time the data covered, it is observable that the normalized execution data on nodes with ID (a) 672206, (b) 554297904, and (d) 4820223869 follow a relatively normal distribution, with approximately same positive and negative values, indicating an average node performance regarding to its ability of executing tasks, while node (c) 257336015 has more negative values, representing a better performance.

When modeling the distribution of nodes execution performance within the derived samples described in Section 4, it is shown that among the 132 machines, 112 and 117 have 3-parameter loglogistic as their best fit (84.85% and 88.64%), and all of them include this distribution in their top three.

B. *Node Execution Performance*

1) *Relationship with physical capacity*

In order to explore whether the assumption adopted by most current literature claiming that higher node capacity always results in better execution performance is valid, we conduct several analyzes. The first sample set is applied as the input for the framework, with mean value selected as the indicator of interest to analyze nodes performance regarding tasks execution. We group the nodes' execution performance results in accordance with four server types described in Table 2. When expressed as a boxplot as shown in Figure 6(a), it is observable that server 4 nodes tend to exhibit larger normalized execution values, while values for server type 2 nodes all fall below zero, signifying most tasks run on this server category execute quicker than their own average. Figure 6(b) illustrates the result that takes the second sample set as the input for the framework with extreme value possibility as the indicator, and nodes still categorized

depending on the four server types to summarize their performance. This analysis indicates similar results with the first sample set, that the ranking of the overall nodes execution performance is $S_2 > S_1 > S_3 > S_4$. However, the server node capacities actually have an opposite ranking, with capacity(S_2) < capacity(S_1) < capacity(S_3) < capacity(S_4).

To test whether this negative correlation exists within other environments, we executed MapReduce jobs on top of a test cluster consisting of 9 VMs (one Namenode and eight Datanodes), using Hadoop 2.5.2 integrated with Hive 1.0.1 [22]. The CPU and memory capacities configured for workers are detailed in Table 3. The Hive queries are automatically transformed into MapReduce jobs that commence execution. For our experiment, four types of queries have been tested on different data sizes, including “select”, “count”, “group by”, and “load data”. By collecting these newly generated job completion logs from the VMs and making them inputs for our framework, we ascertain the node execution performance per machine ID shown in Figure 6(c). We observe that the relationship between node capacity and its execution performance exhibits a positive correlation, differing from the Google analysis result.

From above analysis, we demonstrate that node capacity is not the only factor that decides node performance regarding task execution. In some situations, a larger capacity can actually have worse execution performance for reasons such as different architecture, load, utilization or even year of purchase. Therefore, the assumption of simple proportional correlation between a node’s capacity and its execution performance requires restructuring. Our framework uses task execution data itself to analyze node performance, and is capable of accurately identifying weak nodes.

2) Temporal patterns of node execution performance

Another assumption adopted by most straggler related literature is that node execution performance is a static attribute of servers that will remain constant for long periods of time. However, this assumption breaks down in real systems due to reasons including utilization rate, resource contention, workload type, latent server faults, etc. Figure 7 illustrates the analysis result of how node performance changes over time after applying Google data into our framework, with the average value the indicator of interest to represent performance. Five server nodes have been chosen based on a random selection. From this daily performance trend we observe that node performance in executing tasks varies along with time, demonstrating that node-level

Table 2 Google server capacity characteristics

Server Type	CPU Capacity	Memory Capacity	Proportion
S1	0.5	0.4995	53.50%
S2	0.5	0.2493	30.70%
S3	0.5	0.749	7.96%
S4	1	1	6.32%

Table 3 VM capacity characteristics

VM IP	CPU Cores	Memory	VM IP	CPU Cores	Memory
10.1.0.2	1	1 GB	10.1.0.6	2	1 GB
10.1.0.3	1	2 GB	10.1.0.7	2	2 GB
10.1.0.4	1	3 GB	10.1.0.8	2	3 GB
10.1.0.5	1	4 GB	10.1.0.9	2	4 GB

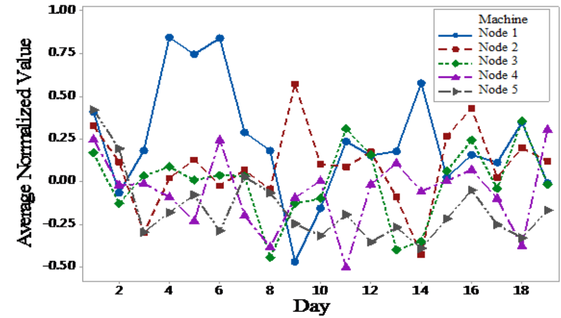


Figure 7. Node execution performance daily changing trend

stragglers identification should take time interval into consideration; simply configuring a static blacklist does not meet this requirement.

Although node performance fluctuates over time, it is still observable that node one performs weaker than the other nodes, with a larger normalized execution value. The next section shows how our framework is capable of ranking nodes according to their performance within the system, and identifying node-level stragglers.

C. Node-level Straggler Identification

Our framework classifies Google nodes into five levels depending on their performance of executing tasks, with level 0 servers representing the slowest node-level stragglers (as such nodes were removed in the first iteration of the ranking procedure detailed in Section 3) and level 4 nodes the fastest. We observe that, 105 out of 12583 nodes within the Google cluster are identified as node-level stragglers after analyzing one month of MapReduce execution data. This information can be integrated into further enhanced scheduling algorithms to improve blacklist or speculation efficiency. Furthermore, when the system is fed with further new executions, the newly generated trace can be used to dynamically adjust the ranking, making it accurately reflect newest system state.

To evaluate the generality of the ranking, another analysis is conducted using batch job execution data. Batch jobs are derived following the method described in [16], which considers the characteristics of job priority, job start and completion time in relation to task submission and completion. Through such filtering criteria, 3,043 jobs comprised of 252,950 tasks within Google trace have been identified. The node performance ranking result generated by this new workload execution log is detailed in Table 4. Although it only classifies the servers into four levels instead of five (with more level 0 nodes), all node-level stragglers that have been identified using MapReduce data are included in this batch job result, indicating the same small set of weakest nodes have been successfully detected.

Table 4 Ranking Results According to Confidence Interval

	Batch Job Result		MapReduce Job Result	
	Node Number	Proportion	Node Number	Proportion
Level 0	416	3.31%	105	0.83%
Level 1	4031	32.04%	1772	14.08%
Level 2	7505	59.64%	7265	57.74%
Level 3	631	5.01%	3386	26.91%
Level 4	--	--	55	0.44%

VI. CONCLUSION AND FUTURE WORK

In this paper, we propose a node performance modeling and ranking framework to analyze node execution ability and susceptibility to straggler occurrence. This can be used to avoid scheduling tasks onto node-level stragglers that exhibiting weak performance. Our core contributions are as following:

- *Propose a node execution performance modeling framework, describing and ranking node ability in terms of parallel job execution*. Slow nodes will influence parallel job execution by enlarging the possibility of the long tail problem, and can limit straggler mitigation technique efficiency by influence speculation execution. Our framework enables the ability to identify node-level stragglers within a cluster in a lightweight manner, and can be easily integrated into systems that already adopted node health check mechanisms such as Hadoop.
- *A ranking of node execution performance within the cluster, enabling further enhanced straggler-aware scheduling*. By using the Google Cloud production data as a case study, our framework generated a node performance ranking which identified the 105 weakest nodes among 12,500+ servers within the datacenter (0.83%) over a period of a month. If the system generates new tracelog data detailing new executions, it can be used to dynamically adjust the ranking, making it accurately reflect the newest system state.
- *Demonstrate that node execution performance is not purely dependent on server capacity*. Current literature assumes that larger CPU or memory capacity of a node means better performance; however, after analyzing real production task execution data and cross-matching it with different server types, we see that those assumptions do not always stand in the presence of stragglers.

Future work includes the development of a node execution performance aware scheduling algorithm that can better improve straggler mitigating efficiency, as well as integrate additional parameters such as current node load into the framework, improve node-level straggler identification into node-level straggler prediction.

ACKNOWLEDGMENT

The work is supported by the National Basic Research Program of China (973) (No. 2011CB302602, No. 2014CB340303), as well as the University of Leeds and CSC joint scholarship program.

REFERENCES

[1] Buyya R, Yeo CS, Venugopal S. Market-oriented cloud computing: Vision, hype, and reality for delivering IT services as computing utilities. In 10th IEEE International Conference on High Performance Computing and Communications, HPCC 2008. pp. 5-13.

[2] Moreno IS, Garraghan P, Townend P, Xu J. Analysis, modeling and simulation of workload patterns in a large-scale utility cloud. Cloud Computing, IEEE Transactions on. 2014; 2(2):208-21.

[3] Patel P, Ranabahu AH, Sheth AP. Service Level Agreement in Cloud Computing, Proceedings of Cloud Workshops at OOPSLA, 2009.

[4] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Communications of the ACM 51.1 (2008): 107-113.

[5] Shvachko K, Kuang H, Radia S, Chansler R. The hadoop distributed file system. In Mass Storage Systems and Technologies (MSST), IEEE 26th Symposium on, 2010, pp. 1-10.

[6] Dean J, Barroso LA. The tail at scale. Communications of the ACM. 2013 1;56(2):74-80.

[7] Google. Google Cluster Data V2. Available: https://github.com/google/cluster-data.

[8] Reiss C, Wilkes J, Hellerstein JL. Google cluster-usage traces: format+ schema. Google Inc., White Paper (2011).

[9] Yadwadkar NJ, Choi W. Proactive Straggler Avoidance using Machine Learning. White paper, University of Berkeley, 2012.

[10] Zaharia M, Konwinski A, Joseph AD, Katz RH, Stoica I. Improving MapReduce Performance in Heterogeneous Environments. in Proceedings of the 8th USENIX conference on Operating systems design and implementation (ODSI), 2008, pp 29-42.

[11] Bortnikov E, Frank A, Hillel E, Rao S. Predicting Execution Bottlenecks in Map-reduce Clusters. USENIX conference on Hot Topics in Cloud Computing, 2012, pp. 18-18.

[12] Ananthanarayanan G, Kandula S, Greenberg AG, Stoica I, Lu Y, Saha B, Harris E. "Reining in the Outliers in Map-Reduce Clusters using Mantri." OSDI, 2010, vol. 10, no. 1, pp. 24.

[13] Chen Q, Liu C, Xiao Z. Improving Mapreduce Performance using Smart Speculative Execution Strategy. IEEE Transactions on Computers, 2014, no. 4, pp. 954-967.

[14] Kwon Y, Balazinska M, Howe B, Rolia J. Skewtune: mitigating skew in mapreduce applications. In Proceedings of the ACM SIGMOD International Conference on Management of Data, 2012, pp. 25-36.

[15] Li J, Sharma NK, Ports DR, Gribble SD. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. in Proceedings of ACM Symposium on Cloud Computing, 2014, pp. 1-14.

[16] Garraghan P, Ouyang X, Townend P, Xu J. Timely Long Tail Identification Through Agent Based Monitoring and Analytics. IEEE symposium on real-time computing ISORC 2015, pp. 19-26.

[17] Kumar U, Kumar J. A. A Comprehensive Review of Straggler Handling Algorithms for MapReduce Framework. International Journal of Grid and Distributed Computing 7, no. 4 (2014): 139-148.

[18] Ananthanarayanan G, Ghodsi A, Shenker S, Stoica I. Effective Straggler Tolerance: Attack of the Clones. In Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation NSDI, 2013, Vol. 13., pp. 185-198.

[19] Ouyang X, Garraghan P, Mckee D, Townend P, Xu J. Straggler Detection in Parallel Computing Systems through Dynamic Threshold Calculation. In Proceedings of the 30th IEEE Conference on Advanced Information Networking and Applications (AINA-2016).

[20] Massey Jr FJ. The Kolmogorov-Smirnov test for goodness of fit. Journal of the American statistical Association. 1951; 46(253):68-78.

[21] Razali NM, Wah YB. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. Journal of Statistical Modeling and Analytics. 2011, 2(1):21-33.

[22] Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R. Hive: a warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment 2, no. 2 (2009): 1626-1629

[23] Zhang Z, Li C, Tao Y, Yang R, Tang H, Xu J. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. Proceedings of the VLDB Endowment, 2014: 1393-1404.

[24] Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B. Apache hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th annual Symposium on Cloud Computing 2013 (p. 5).

[25] Batagelj V, Zaveršnik M. Generalized cores. arXiv preprint cs/0202039. 2002 Feb 28. http://arxiv.org/abs/cs/0202039.