Article:
Ouyang, X, Wang, C and Xu, J orcid.org/0000-0002-4598-167X (2019) Mitigating stragglers to avoid QoS violation for time-critical applications through dynamic server blacklisting. Future Generation Computer Systems, 101. pp. 831-842. ISSN 0167-739X

https://doi.org/10.1016/j.future.2019.07.017

Mitigating Stragglers to Avoid QoS Violation for Time-Critical Applications through Dynamic Server Blacklisting

Xue Ouyang[a,*], Changjian Wang[b], Jie Xu[c]

[a]School of Electronic Sciences, National University of Defense Technology, China
[b]School of Computer, National University of Defense Technology, China
[c]School of Computing, University of Leeds, UK

Abstract

The straggler problem is one of the most challenging issues toward rapid and predictable response time for applications in cluster infrastructures, leading to potential QoS violation and late-timing failure. Straggler tasks occur due to reasons such as resource contention, hardware heterogeneity, etc., and become severe with increased system scale and complexity. Speculative execution and blacklisting are the major two straggler tolerant techniques, but each has its own limitations. The former creates replica task to catch up with the identified straggler, but normally with no selection toward nodes when deciding where to launch the backup. Ignoring server performance hinders the speculation success rate. The latter typically relies on manual configuration, despite the fact that the ability of nodes to effectively execute tasks changes over time. In addition, the misidentification of weak-performance nodes decreases system capacity. Combining these two techniques, we present DSB, a dynamic server blacklisting framework which takes into account both historical and current behavior of a server node to increase straggler mitigation effectiveness. Servers are ranked at each time interval according to their performance in fulfilling jobs instead of their physical capacities, and the worst performed ones got temporarily blacklisted. As a result, no new tasks/replications are assigned to those straggler-prone nodes within the following time window. DSB also provides an alternative API where adjustable top k worst nodes can be blacklisted according to the ranking. The optimal k is investigated as a trade-off between capacity loss and straggler mitigation efficiency. Results show that, the DSB scheme is capable of increasing successful speculation rate up to 89%. In addition, it can improve job completion time by up to 55.43% compared to the default speculator in the YARN platform. This helps to reduce the chance of QoS violation, which is particularly important for time-critical applications.

Keywords:
`Time-critical application`, `Straggler`, `Speculation`, `Blacklist`, `Node performance`

1. Introduction

Clusters often consist of thousands of server nodes with different physical capacities (including CPU, memory, disk, etc.), operational age, and architecture [1]. These heterogeneities along with the dynamic resource utilization and multi-tenancy result in diverse task execution performance for each node [2], which leads to the straggler problem. Stragglers are parallelized tasks which experience abnormally longer duration compared with other sibling tasks within the same job, resulting in degraded service response as well as reduced system availability. For services that emphasize timely behavior, stragglers can cause late-timing failures [3] and Quality of Service (QoS) breakdown [4]. Node execution performance, in this paper, is defined as the measurement of effective task execution within a node in the presence of stragglers. A server exhibits poor execution performance indicating a higher task straggler occurrence possibility.

Two common solutions for mitigating task stragglers are speculative execution and server blacklisting. Speculative execution [5] launches replica copies for identified stragglers in an attempt to outpace the original task. This method is the dominant approach used in industry practice, and are adopted by companies including Google, Facebook, Microsoft, etc. However, the speculation scheme has to wait until the stragglers been identified before it can put efforts in mitigation, therefore its effectiveness can be undermined by late detections. To make things worse, since the speculator does not exert special selection toward nodes, there is still a chance for the replica tasks to be assigned to slow nodes and become stragglers. Statistical analysis conducted based on the OpenCloud dataset indi-

*Corresponding author
Email addresses: `ouyangxue08@nudt.edu.cn` (Xue Ouyang), `c_j_wang@yeah.net` (Changjian Wang), `j.xu@leeds.ac.uk` (Jie Xu)

cates that, as high as 71.22% speculations are actually created in vain on average, ended up being killed by the system because the stragglers are finishing first (Section 3.1). Poor speculation efficiency lead to waste resources. For systems with high utilization rate, these wasted replications may result in a higher contention that further enlarges the chance of straggler occurrence [6] and deteriorate job execution performance.

Server blacklisting is another method used to avoid stragglers [7]. It reduces straggler occurrence though forbidding the usage of weak nodes by configuring a blacklist. The precision of weak node identification is vital in these approaches: the false positive results in unnecessary capacity loss while the false negative hinders straggler avoidance performance. Most methods assume that slow nodes are static [8, 9] (i.e. server execution performance will constantly be poor), however in practice they can be transient. Ignoring such dynamic characteristic limits the scheduler to make smart decisions.

In this paper we propose DSB, a dynamic server blacklisting framework to tolerate stragglers for time-critical applications. It functions through periodically updating the cluster node ranking based on the parallel job execution log, and always launches tasks (both original ones and the speculative replicas) on fast nodes. Our main contributions are summarized as follows:

(1) Analyzing speculative execution effectiveness with real data from a production cluster infrastructure. We demonstrate how node execution performance varies with time, and how straggler behavior is influenced by such fluctuation. Statistics toward successful speculation rate and improvement potential are discussed as well, revealing the state of the art straggler mitigation methods are far from effective.

(2) Proposing an enhanced node execution performance ranking algorithm. This algorithm is based on our previous work that models and ranks node execution performance leveraging historical tracelog data [2]. The previous method is demonstrated to be effective in the presence of stragglers, yet faces a challenge when dealing with failed tasks. In this paper we make improvements toward the data collection and filtering process. The node performance analysis procedure is enhanced as well.

(3) Designing an execution performance aware node blacklisting scheme. Node performance is important in improving speculation efficiency. The proposed DSB framework dynamically reflects the node performance changing trend, which enables enhanced speculation and scheduling when dealing with stragglers. An additional API is provided in DSB to support the customized blacklisting, in which the relationship between the blacklisted node number with performance improvement is analyzed.

The rest of the paper is structured as follows: Section 2 discusses related work of speculation, blacklisting and scheduling that targeting the straggler problem; Section 3 presents the straggler problem background and the speculation limitation; Section 4 illustrates the dynamic server blacklisting framework design; Section 5 discusses the experiment setup as well as the system implementation; Section 6 presents the result and evaluation; Section 7 analyses conclusions and future work.

2. Related Work

Current straggler mitigation techniques can be divided into two main categories: avoidance and tolerance. Avoidance based methods always occur at the task scheduling phase [8, 10]. For example, the MapReduce scheduler often assigns Map tasks to nodes that store the input data in order to reduce unnecessary network transmission [5]. The scheduler may also attempt to avoid scheduling tasks onto known faulty nodes by adopting blacklisting techniques [11, 7], in which tasks will never be assigned to such nodes until list removal.

The effectiveness of these approaches is dependent on correctly detecting faulty nodes for blacklisting, otherwise the system capacity will be degraded due to the false positives. Besides, current blacklisting practice tends to assume that weak nodes are known by the system administrator, which is not true in production environments, and it is infeasible to conduct manual configuration for clusters comprising thousands of nodes. Some works adopt data analytics toward node performance to avoid slow nodes [12], however, such techniques are insufficient when stragglers are not restricted to a small set of machines [13]. In addition, the node execution performance does not remain stable over time, and such static method is unable to accurately capture the most up-to-date performance. There is presently a lack of a comprehensive framework for modeling and ranking node execution performance that can be applied generally to Cloud datacenters.

Another type of avoidance based methods is through efficient virtual infrastructure management, considering orchestrating time-critical applications on Clouds [14]. For instance, a graph based simulation method is applied to evaluate the complexity of the infrastructure network in [15], and a trust model is leveraged to figure out the Cloud

provider with bad reputation [16]. Even the virtual infrastructure can be planned in advance according to the application requirements [17]. Some other works focus on managing the infrastructure at different phases to provide better QoS assurance, such as the provisioning phase [18], the deployment phase [19], and the runtime phase [20]. In addition, several tools are developed to achieve effective virtual infrastructure management [21, 22]. However, all these work try to tackle the timing issue from the Cloud user perspective, which is not the emphasis of this paper. In this paper, straggler avoidance from datacenter infrastructures, or in other words, from the Cloud provider perspective, is considered.

Different with straggler avoidance, tolerance is typically performed at the application run-time. Speculative execution [5] is the dominant approach of this kind. It monitors the progress of each task and launches speculative copy that performs identical work for the identified straggler. The system will adopt whichever result that comes out first and abandon the other instance. Speculation functions with the assumption that the backup copy will complete prior to the straggler, however, due to reasons such as late identification or misplacement of the replication, sometimes the straggler finishes first. Under these cases, the scheduler will discard the speculation and releases the computing resources back to use, leading to a failed speculative attempt.

There exist numerous techniques which extend the default speculative execution method in terms of specified cases, such as for heterogeneous environment [23] and for small job's execution (less than 10 parallel tasks) [13]. 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 replications and ignore the impact of poor node execution performance. It is particularly important to avoid scheduling speculative replicas to the weak nodes, because that practice can lead to a decreased likelihood for the replication to complete prior to the task straggler. Failed speculation results in limited improvement toward job execution as well as increased resource overhead.

In terms of determining the suitable nodes for replica placement, Chen et al. [9] consider both data locality and data skew to develop a cost benefit model based on the cluster load. Yadwadkar et al. [8] develop a system that performs regression using node level statistics based on a production trace. Through periodically produces correlations between node level status and task execution time in the form of a decision tree, this work enhances scheduling policy when determining which node to run the replica in order to minimize overall job execution. While these methods applying data analytics to identify weak nodes, they each assume that poor 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, workload heterogeneity, and user demand.

Beyond the speculation scheme, there exist numerous methods dealing with special types of applications. For example, [24] is designed for distributed matrix multiplication. In addition, coding-theory-inspired approaches such as [25] are applied to mitigate the effect of straggling through embedding redundancy in certain linear computational steps, thus completing the computation without waiting for the stragglers. Optimization method [26] proposes an alternate approach where the redundancy is directly embedded in the data itself, thus allowing the linear computation to proceed completely oblivious to the encoding schemes. Based on these research, some work further examines whether the redundancy should be simple replication or coding [27], and other literature such as [28] analyzes the effect of coding and replication on the tradeoff between cost and latency, quantifies the effect of the tail of task execution times and discusses tail heaviness as a decisive parameter for the

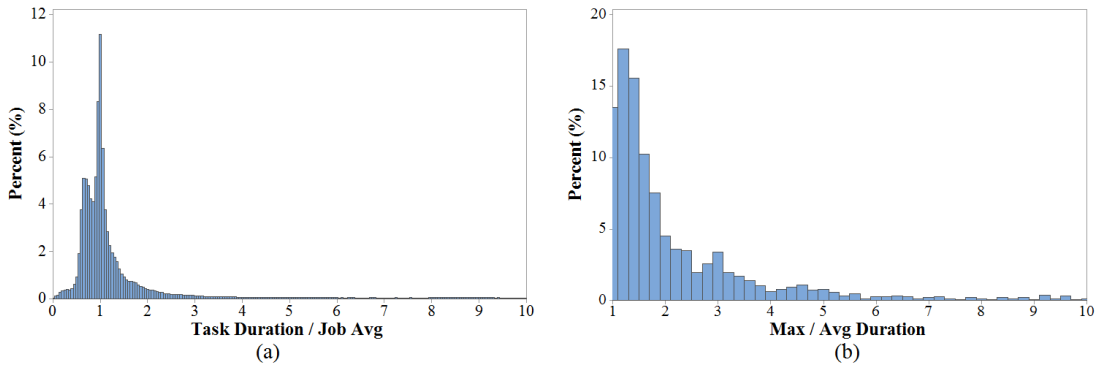


Figure 1: Statistics in the OpenCloud cluster: (a) individual task duration versus job average, (b) job containing stragglers

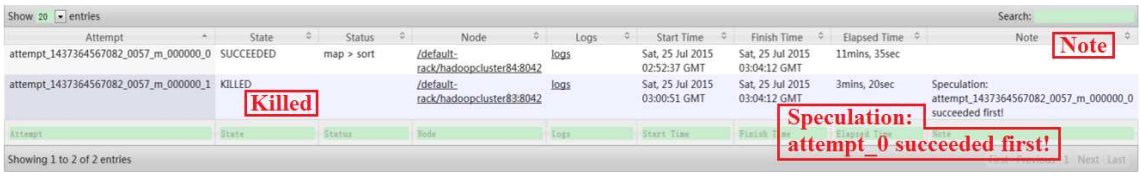


Figure 2: An example of a killed speculation for a Hadoop job

cost and latency of using redundancy.

3. Problem Statement

3.1. Stragglers in Real World Systems

Stragglers are intensively discussed within the MapReduce [5] background as it is the most prominent parallel computing framework for processing large data sets within massive-scale clusters. It mainly consists of seven steps: (1) the MapReduce library forks input file into data chunks; (2) the scheduler assigns map / reduce tasks; (3) the mappers read key-value pairs; (4) the map functions generate intermediate data pairs; (5) the reducers remotely read the intermediate results; (6) the reduce functions generate outputs; and (7) the output files are appended. Following the above steps, the MapReduce model automatically parallelizes and distributes large-scale computation jobs into smaller tasks running on different server nodes.

Typically, a job needs to wait for all the results generated by its parallelized tasks, and the response time is dependent on the last task. Ideally, parallelization will make its best advantage if all tasks generate results at the same time. However in production systems, due to factors such as heterogeneous machine hardware, network latency variation, shared resource contention, and unbalanced input sizes, stragglers will occur [29] which take significantly longer execution time.

Straggler occurrence is analyzed using trace data from production environments to demonstrate its influence. The OpenCloud system at Carnegie Mellon University[1] is a research cluster for applications in areas including machine learning, natural language processing, and social networking analysis, which normally require rapid response. The cluster is composed of 116 homogeneous server nodes, running the Hadoop platform, the de facto standard for open source MapReduce. After applying filters to generate a ten-month-period tracelog, altogether 18,935 jobs and 8,734,974 corresponding subtasks were included in the analysis.

Figure 1 shows the straggler occurrence within the OpenCloud cluster and the percentage of jobs that experience extended durations. Figure 1 (a) is the distribution of individual task durations compared to the mean completion time of all tasks within the same job. It is observable that, most tasks exhibit similar duration around the average, with a small proportion of tasks completing much later: the longest being 10 times slower. Stragglers, with a duration 50% larger than job average, account for approximately 5% of total tasks within the system. Figure 1 (b) portrays the distribution of jobs that contain stragglers: with the slowest task exhibit more than 1.5 times duration compared to its average. From the graph it is seen that, almost half of the parallel jobs are influenced by the straggler behavior. Such phenomenon is also identified in other production systems such as Google [30], demonstrating that even rare performance abnormalities of tasks can affect a significant portion of

[1] http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html

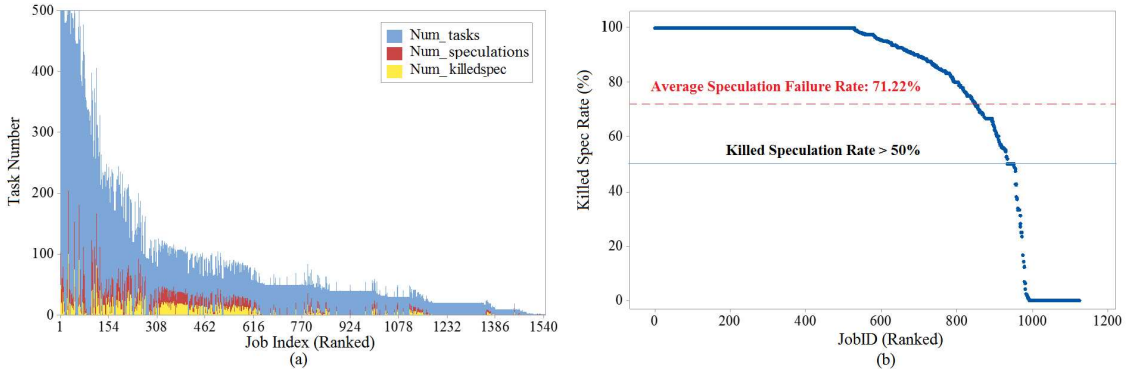


Figure 3: Killed speculations in the OpenCloud cluster (a) numbers (b) statistics

all jobs in large-scale infrastructure.

3.2. Speculation and its Limitations

There are a lot of related works in tolerating negative impacts brought by the stragglers, among them are speculative execution (speculation in short) and blacklisting. Speculative execution [5] is commonly used in industrial clusters such as Facebook, Google, Bing, and Yahoo!, and is been integrated into the default Hadoop/YARN/Spark versions as the mainstream straggler mitigation technique. It observes the progress of each individual task and creates replicas for stragglers. The original straggler will not be killed upon speculation, instead, the system will let the two copies compete with each other, adopting the quicker result to shorten the overall job completion. An example of such replication is given in Figure 2. In Hadoop implementation, original tasks are attempts marked with suffix 0 while speculative copies are with suffix 1. From the "*Note*" column it is observable that, the created speculation got abandoned in the end due to the straggler succeeded first.

Wasted speculation that ended up being killed is not a rare case within the system, on the contrary, they appear quite often. Figure 3 (a) depicts the distribution of total task number (represented in blue) versus speculation number (represented in red) versus killed speculation number (represented in yellow). Different jobs are listed in the x-axis, in the descending order of the task number that the job contains. A clear observation is made from the graph: a large proportion of speculations are actually been killed in the end. Figure 3 (b) further demonstrates the overall statistics of this proportion: the killed speculation rate in the OpenCloud system in average reaches as high as 71.22%, and for almost half of all jobs, the speculative execution is useless at all.

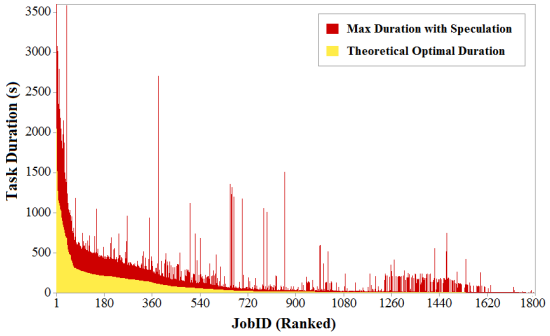


Figure 4: The improvement potential of the speculation in the OpenCloud cluster for jobs with duration less than one hour

There are two approaches that can be used in order to help the speculative copies to catch up with the stragglers and to decrease the speculation failure rate. One is to assign the replicas to fast nodes for a quicker execution, while the other is to predict straggler occurrence according to the node execution performance. Both methods require node performance modeling. In addition, if we assume the

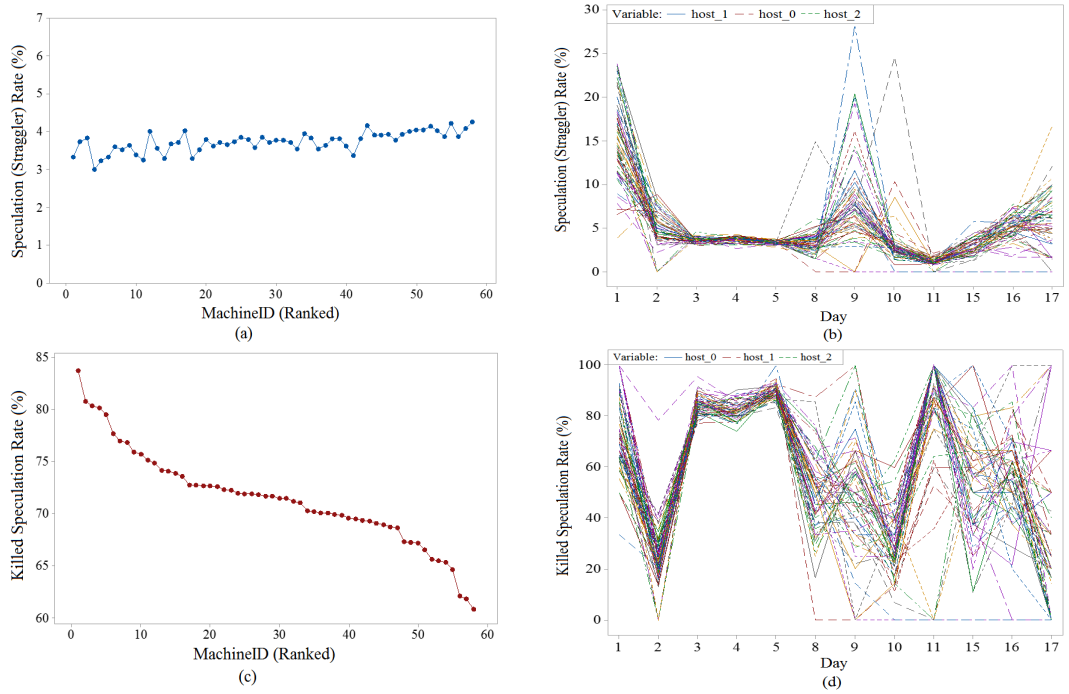


Figure 5: Straggler rate per node (a) in a month time; (b) per day changing trend; and killed speculation rate per node (c) in a month time; (d) per day changing trend. Each line in (b) and (d) represents a node, the legend only gives three examples due to the space

theoretical best case performance speculation can achieve, is to eliminate all stragglers and replace their durations with the average job execution time, we get the speculation performance improvement potential. As shown in Figure 4, the huge gap between the actual execution time for OpenCloud jobs and the theoretical optimal duration indicating another 65.7% performance improvement in average for current speculation mechanism.

3.3. Blacklisting and its Limitations

Besides speculative execution, the other popular straggler tolerant technique is through avoidance. Blacklisting [11] is the representative method of this kind, avoiding scheduling tasks onto known faulty nodes. However, blacklisting may be insufficient when stragglers are not restricted to a small set of machines [13]. In addition, current blacklisting is often through manual configuration (such as to configure the mapred-site.xml file in Hadoop), which requires input from the system administrator. This practice is inflexible especially when the system scale increases.

Another challenge encountered by blacklisting is to capture the most up-to-date node performance when this attribute changes dynamically. Figure 5 (a) shows the data analytics result of the OpenCloud machine behaviors within a 20-day period in October. The speculation rate (or straggler rate, because the speculation will only be triggered if a straggler is identified) for each node is balanced: every one of them faces a 3% to 4% straggler occurrence possibility. However, if we split the performance into daily basis as shown in Figure 5 (b), we observe a quite different trend. On some days such as the 3^{rd} to the 7^{th} day, the straggler rate across different machines are relatively the same (each line in the graph is a machine node in the cluster), however for other days such as the 8^{th} to the 11^{th} day, the performance on each machine varies a lot: the weakest performance is experiencing almost 30% straggler rate while the others remain less than 5%. With a static server blacklist, such characteristic cannot be captured to achieve the best performance. Similar trends for the speculation failure rate are also observed as shown in Figure 5 (c) and (d), which reflect the dynamic state of the speculation efficiency.

3.4. Straggler Root-Cause and Node Performance

It is advantageous to understand the root-causes of stragglers in order to better mitigate them using specifically-designed methods. As discussed in previous sections, stragglers stem from numerous reasons ranging from application related reasons such as unbalanced input data and poorly-designed code, to server related reasons such as heterogeneous machine hardware and shared resource contention [29].

An investigation of correlation is conducted in a previous study of us, using historical data to derive a deep insight into the most important root cause of stragglers [31]. Results show that, server related reasons including high CPU/memory utilization and hardware faults account for more than 60% of straggler occurrence, while data skew type of stragglers only accounts for around 5%. This proportion supports the importance of node execution performance to general straggler mitigation, and motivates the proposition of DSB, the Dynamic Server Blacklisting framework. The idea is trying to avoid weakly performed servers through analyzing task behavior per node statistics.

In addition, the DSB design is an optimization made at the node side, it can cooperate with other methods focusing on the application side to mitigate stragglers together. For example, the ones that taking straggler root cause into consideration. DSB is not contradicted with enhanced speculative execution or specific data skew mitigation methods.

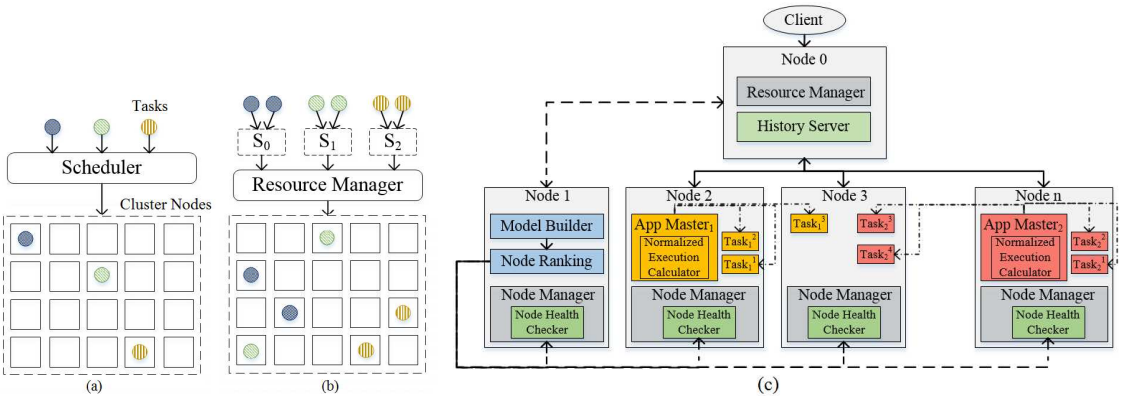


Figure 6: (a) The monolithic and (b) the two-level scheduling architecture; (c) the DSB system model

4. The Dynamic Server Blacklisting Design

4.1. System Model

Currently, both the speculator component and the blacklisting technique have to work with a specific scheduling scheme in order to complete the system design. For the former, once the speculator identifies a straggler and creates a backup, the replication will be submitted to the scheduler as a normal task waiting to be assigned to a node for execution; for the latter, the blacklist provides available resources pool for the scheduler to generate the scheduling decision. There are many scheduler architectures in modern Cloud datacenters such as the monolithic scheduling and the two-level scheduling [32]. In the monolithic architecture shown in Figure 6 (a), all tasks run through the same scheduling logic with a single scheduler. The Borg scheduler [33] and the default scheduler in Hadoop version 1 [5] belong to this kind (e.g., the JobTracker in Hadoop V1). The monolithic scheduling is simple and uniform, however, meets bottleneck handling mixed workloads. This is tricky when most clusters today run heterogeneous types of applications. The two-level architecture shown in Figure 6 (b) has been proposed to address this problem, by separating resource allocation and task placement. The YARN scheduler [34] belongs to this type.

Considering the decentralized nature and its advantage in scaling, we develop our DSB system on top of the YARN architecture. Figure 6 (c) illustrates the system model. The key components of the *Application Master (AM)*, the *Resource Manager (RM)* and the *Node Manager (NM)* are consistent with the default YARN design. AM is in charge of job execution, including MapReduce job creation, request resource from RM, communicate with NM to run containers, monitor job running status and do speculation, etc. RM is responsible for resource management functionalities such as request and release containers, while NM is responsible for launching containers, reporting resource usage to RM and heartbeat. Other modified components related to the DSB implementation including *History Server* that records job/tasks execution logs and *Health Checker* that reports node status and conduct server blacklisting. Additional components such as the *Execution Calculator* is responsible for modeling and ranking the server node execution performance.

The dynamic server blacklisting framework functions through two key steps detailed in Algorithm 1: the node execution performance ranking and the blacklist-based scheduling. For the algorithm inputs, the "*Task*" element is a user defined structure which contains attributes including *tID*, *jID*, *mID*, and *duration*. The notion of *tID* is short for taskID, and is a unique identification string. Similarly, *jID* is short for jobID, indicating which job this certain task belongs to, and *mID* is short for machineID, showing the machine number this task runs on. The notion of *duration* represents the execution time of the task. Other notions used under the DSB background are: a job J_j is decomposed into m tasks. T_j^i represents the i^{th} task in job J_j ($0 < i \leq m$). D_j^i stands for the duration of task T_j^i, while $\overline{D_j}$ is the average duration of job J_j. The cluster is consists of n server nodes, with M_k denoting the k^{th} machine in the cluster ($0 < k \leq n$).

4.2. Framework Description

Key phases of the DSB framework include following procedures: calculating normalized execution value for tasks, building up machine execution performance model, interval-based ranking, and blacklisting. The first step captures key features to represent node execution performance while the second step builds up the distribution model and generates statistical attributes. The third procedure ranks the nodes according to the attributes and the fourth step forbids the weakest nodes from the available resource pool. The following subsections introduce the algorithms developed for each step.

4.2.1. Normalized Task Execution Value

Server nodes in cluster infrastructure are typically composed of different characteristics including resource capacity (CPU, memory, disk, etc.), architecture, and operational age. Besides these physical heterogeneities, dynamic attributes such as utilization and multi-tenancy also result in diverse task execution performance at given time for each node within the system, such as the demonstrated straggler occurrence rate shown in Figure 5. Therefore, it is important to periodically evaluate the node execution performance to avoid straggler occurrence and to improve speculation effectiveness.

The DSB system uses the task execution log to generate the node performance model through calculating the normalized task durations with respect to their own job average using Equation (1).

$$\widetilde{D_j^i} = \frac{D_j^i - \overline{D_j}}{\sigma_j} \tag{1}$$

where σ_j is the standard deviation of all tasks' duration of job J_j. The normalized value of task duration is used in DSB when modeling node performance, because there are always multiple workloads co-exist in clusters, each with a differently designed duration, therefore, raw job response time cannot be compared directly to reflect node execution performance. $\widetilde{D_j^i}$, on the other hand, reveals the

Algorithm 1 The DSB Workflow

Inputs:
{tasks}: A task set with "*Task*" elements
{machines}: A machine set with "*String*" elements

```
1: while True do
2:     set Ω = ∅, set Ψ = ∅, set Γ = ∅
3:     for each task ∈ {tasks} do
4:         μ = NormalizedExecutionValue(task, {tasks})
5:         ω = ⟨ task.tID, task.jID, task.mID, μ ⟩
6:         Ω = Ω ∪ {ω}
7:     end for
8:     for each ω ∈ Ω do
9:         for each mID ∈ {machines} do
10:            if (mID == ω.mID) then
11:                ψ = ⟨ mID, ω.μ⟩
12:            end if
13:        end for
14:        Ψ = Ψ ∪ {ψ}
15:    end for
16:    for each mID ∈ {machines} do
17:        init CI = ⟨ Low, High ⟩
18:        CI = MachineExecutionPerformance(mID, Ψ)
19:        γ = ⟨ mID, CI ⟩
20:        Γ = Γ ∪ {γ}
21:    end for
22:    set Δ = IntervalBasedRank(Γ)
23:    BlackList(Δ)
24:    Sleep(TimeWindow)
25: end while
```

Algorithm 2 Machine Execution Performance

Inputs:
targetMID: The target machine for performance evaluation
Ψ: A set of normalized value tuples {⟨ mID, μ⟩}
Output:
CI: The performance confidence interval for the machine

```
1: set ϒ = ∅
2: AvgNValue = 0, tNum = 0
3: for each ψ ∈ Ψ do
4:     if (targetMID == ψ.mID) then
5:         ϒ ∪ {ψ.μ}
6:         tNum++
7:         AvgNValue += ψ.μ
8:     end if
9: end for
10: AvgNValue /= tNum
11: StDevNValue = 0
12: for each υ ∈ ϒ do
13:     StDevNValue += math.pow((υ− AvgNValue), 2)
14: end for
15: StDevNValue = math.sqrt(StDevNValue / tNum)
16: td = TDistribution(tNum - 1)
17: a = td.inverseCumulativeProbability(0.975)
18: CI.Low = AvgNValue - StDevNValue*a/math.sqrt(tNum)
19: CI.High = AvgNValue + StDevNValue*a/math.sqrt(tNum)
20: return CI
```

relative speed of task T_j^i, makes it possible to compare the performance of tasks running on different nodes irrespective of job heterogeneity.

The *Normalized Execution Calculator* component in *Application Master* as shown in Figure 6 (c) is in charge of the $\widetilde{D_j^i}$ calculation, following Equation (1). A negative $\widetilde{D_j^i}$ value indicates a quick task completion while a positive $\widetilde{D_j^i}$ represents a slower execution because the duration of T_j^i is larger than average. The larger the positive value, the more severe straggler behavior T_j^i exhibits.

4.2.2. Machine Execution Performance Model

For each machine M within the cluster, the normalized execution values for tasks that are assigned to M within a certain time period can be used for analyzing its execution ability during that time. If the majority of tasks assigned are with positive $\widetilde{D_j^i}$s, which indicate slower executions compared with their own average job duration, we say that M encounters a poor execution performance. In contrast, intensive negative $\widetilde{D_j^i}$ values observed from M demonstrate a good execution performance, because tasks assigned to this specific machine tend to always finish quicker than the other sibling tasks from the same job. In the DSB system, every node holds a file consisting of following 6-tuples:

$$< (T_j^i)_s, (T_j^i)_e, ID_{T_j^i}, ID_{J_j}, ID_{M_k}, \widetilde{D_j^i} >$$

where $(T_j^i)_s$, $(T_j^i)_e$ and $\widetilde{D_j^i}$ represent the start time, the end time, and the normalized execution value of task T_j^i that runs on machine ID_{M_k}. This file provides input that enables the analysis of building up the $\widetilde{D_j^i}$ distribution model per node following algorithm 2. The input {⟨ mID, μ⟩} is collected based on the aforementioned file, and the target machine is represented as *targetMID* in the algorithm. The *Model Builder* component demonstrated in Figure 6 (c) is responsible of conducting such analysis.

In the proposed algorithm, once the probabilistic distribution model has been generated, the statistical attributes such as the mean or the quantile values can be used to evaluate node execution performance. To note that, it is reasonable to assume that each node has an equal chance to run the data skew type of slow task, therefore from statistical point of view, when leveraging slow task behavior to measure node execution performance, the influence generated by data skew stragglers to each node are at the same level, therefore can be eliminated. In other words, if a node got blacklisted in our algorithm, it is more due to its contention or temporary unhealthy behavior, in which case migrate tasks running on them is meaningful.

In the current implementation, the 95% confidence interval (CI) is adopted to ascertain the likelihood of straggler occurrence.

4.2.3. Ranking and Server Blacklisting

The node execution performance ranking is generated in this step based on the confidence intervals, and the *Node Ranking* component in Figure 6 (c) is in charge of classifying nodes into different levels that describe the susceptibility of straggler occurrence. The weakest set of nodes in the ranking order will be put on the server blacklist.

In this paper, we modify a graph-based ranking

Algorithm 3 Interval Based Ranking

Inputs:
{machines}: A machine set of "Machine" elements
Output:
{mID}: The machine ID set indicating weakest n machines

```
1:  for each machine ∈ {machines} do
2:      init machine.out = ∅, machine.int = ∅
3:  end for
4:  for each m1 ∈ {machines} do
5:      for each m2 ∈ {machines} do
6:          if (m1.mID != m2.mID) then
7:              if (m1.CI.High ⩽ m2.CI.Low) then
8:                  m1.outEdge = m1.outEdge ∪ {m2.mID}
9:                  m2.inEdge = m2.inEdge ∪ {m1.mID}
10:             end if
11:         end if
12:     end for
13: end for
14: {mID} = ∅
15: for each m ∈ {machines} do
16:     if (m.outEdge == ∅) then
17:         {mID} = {mID} ∪ {m.mID}
18:     end if
19: end for
20: return {mID}
```

algorithm named P-Cores [35] to rank the performance CIs by constructing a directed acyclic graph (DAG). Cluster machines are represented as the nodes in the DAG while the edges indicate the sequence of the CIs. The principle adopted to determine the construction of the sequence edge is given in Algorithm 3: if $[L_1, H_1]$ and $[L_2, H_2]$ represent the performance CIs of machine M_1 and M_2 respectively, with L_i and H_i to be the low and the high bound of the CI, there will be an edge from M_1 to M_2 only when $L_2 > H_1$. CI overlaps will not lead to an edge under this construction.

It can be inferred that, if a machine is with no outward edge, it is the current weakest node because its CI is larger than the others, indicating a frequent straggler occurrence and a severe tailing behavior. It is common that the CIs could overlap with each other, which means the performance difference between the two compared nodes is not that remarkable. Our previous published case study [2] ranks Google nodes using CI as the indicator which classifies Google nodes into 5 performance levels. Results show that, the weakest level 0 nodes accounts for 0.83% of the total population. The majorities are the middle levels (level 1-3 in the case study), accounting 98.73% in total. The nodes in the same level exhibit similar performance due to the fact that their CI is overlapped. This proportion is consistent with the intuitive sense, and our goal is to blacklist the weakest nodes that show obvious performance differentiation compared to others.

As long as the ranking has been generated at each timestamp, the DSB framework can then periodically blacklists the weakest-performed nodes from the cluster function set to improve job execution time as shown in Algorithm 3. The input "*Machine*" element is a user-defined structure that contains attributes of *mID* and its performance CI. The P-Cores algorithm repeatedly records the weakest nodes at each iteration (the ones without any outward edges in the DAG), removes them from the graph with all related inward edges until there are no remaining nodes. The level zero nodes represent the ones that are removed at the first iteration, and are the weakest ones among all. The default server blacklisting policy as demonstrated in Algorithm 3 returns all level zero nodes at the time when the procedure is called, termed as the *P-Cores without number* policy.

In extreme cases that all CIs are overlapped with each other and there is no obvious weak node, we can either blacklist no nodes or specify a certain number to blacklist depending on customized preference, in either case the algorithm still works. Next section discusses the partial blacklisting as the supplement of the default server blacklisting policy, termed as the *P-Cores with number policy*.

4.2.4. Partial Blacklisting as a Supplement

The automatic server blacklisting method can help reduce the administrative burden of setting the predefined blacklist size, guarantees that all weak nodes can be prohibited to reduce the straggler possibility. However, under extreme cases when the cluster size is small, this policy has a risk of hindering system capacity due to no control of the exact server number that been blacklisted. For example, in special cases when all machine CIs are overlapped with each other, making the DAG contains no edges but only scattered points, the *P-Cores without number* policy will rank all nodes as level zero ones and blacklists all available machines.

In order to solve this problem, DSB introduces the *P-Cores with number* blacklisting policy as an alternative complement, which generates the top k weakest nodes. It uses standard deviation and the mean value as two vice indicators to help with the ranking procedure. When the number of level zero nodes surpasses a certain threshold, we will further rank them in descending order of the StDev value and of the mean value, respectively. The top k nodes in the intersection of these two rankings are selected as forbidden servers. For example, if the automatic ranking classifies ten level zero nodes while the system configuration only allows at most six nodes to be removed from the working list considering capacity loss, the *P-Cores with number* policy would rank these ten nodes according to $\widetilde{D^i_j}$ StDev and $\widetilde{D^i_j}$ mean. If the descending order of the StDev is M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8, M_9, M_{10} while the descending order of the mean is M_1, M_2, M_3, M_4, M_5, M_7, M_6, M_8, M_9, M_{10}, the inter-

section of the top six of these two ranking are M_1, M_2, M_3, M_4, M_5. These five nodes will be added to the server blacklist. As for the sixth nodes, the *P-Cores with number* policy would randomly pick either M_6 or M_7 because these two are the sixth nodes from the two ranking. The idea supporting the random choice of the controversial item in the intersection is from Sparrow [36], which adapts the power of two choices load balancing technique [37] to the domain of parallel task scheduling.

The heuristic for this partial server blacklisting is that, nodes with weaker performance will result in a more random task execution behavior, characterized by a larger $\widetilde{D_j^i}$ StDev, and a longer execution time for most tasks, characterized by a larger $\widetilde{D_j^i}$ mean. Through this alternative policy, users can control the blacklisted node number.

The last (but not least) procedure of the DSB design is releasing nodes from the server blacklist. The methodology adopted here is that, at each time period, the ranking result will be re-calculated based on the behavior of all available servers in the system, including the ones in the blacklist. This policy implies a release action: as long as the performance ranking of the blacklisted nodes can surpass some current working nodes, they can then be removed from the blacklist in the upcoming time interval. Refer back to Algorithm 1, at the beginning of each iteration, all related sets, such as the task set Ω and the normalized value set Ψ per machine, are all initialized as empty sets so that the calculation can involve information of the newly generated tasks as well as the servers in the blacklist during the past time interval.

5. Experiments

5.1. Experiments Setup

In order to verify the generality of the proposed framework, two testbed environments are used for our experiments. The first one is a 30 virtual machine (VM) cluster built on top of the OpenNebula platform[2] with typical VM configuration to be 1 GB of memory, 1 virtual core with 2.34 GHz capacity, and 10GB disk space on potentially shared hard drive. The VMs use KVM virtualization software and run the Ubuntu 12.04 x86_64 operating system. Another testbed is a 20 VMs cluster build on top of the ExoGENI infrastructure[3] with 2 XOLarge VM and 18 XOMedium VM (detailed configuration of each VM type can be found in[4]), run the CentOS 6.7 operating system.

In all experiments, we configured the HDFS to maintain two replicas for each data chunk. The job types for the experiments include WordCount and Sort, which are given in the original Hadoop distribution[5]. We use these two workloads because they are the main benchmarks used in Google's MapReduce paper [5] and in the LATE [23] paper for evaluating Hadoop performance. In addition, we configured the container sizes for both map and reduce tasks to be 1GB of memory, and the node capacity to be 2GB. In the ExoGENI cluster, we configured an Ambari[6] system to monitor and to manage the cluster utilization. Figure 7 shows the system utilization of ExoGENI cluster, with Figure 7 (a), (b), and (c) to be the CPU, the memory, and the network usage, respectively, when no user program is

[2]http://opennebula.org/
[3]http://www.exogeni.net/
[4]https://wiki.exogeni.net/doku.php?id=public:experimenters:resource_types:start
[5]http://hadoop.apache.org/
[6]https://ambari.apache.org/

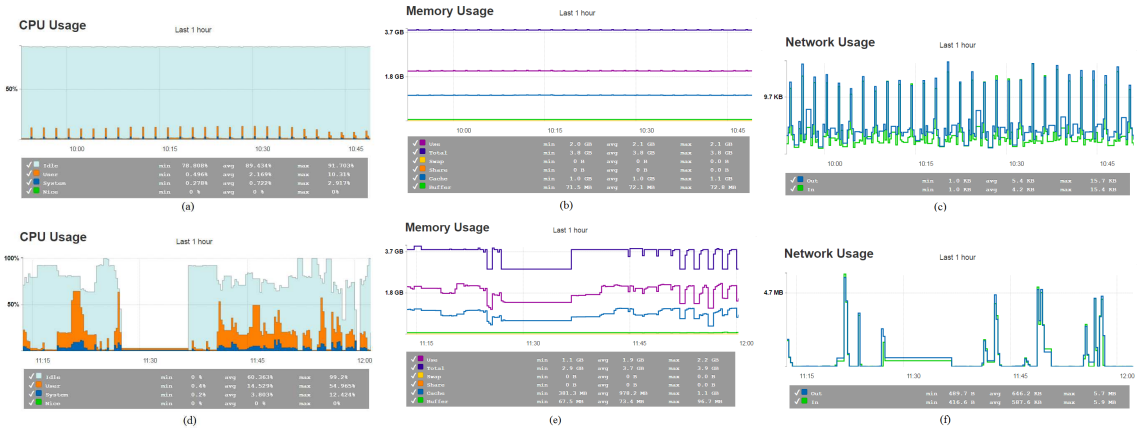


Figure 7: The CPU, memory, and network utilization of the ExoGENI cluster with (a)(b)(c) no user program running, respectively; and with (d)(e)(f) user workload running

Table 1: The VM Configurations for Experiment Clusters

	OpenNebula	ExoGENI	OpenNebula 2
VM Number	30	20	30
Injected CPU Fault	10.1.0.27, 10.1.0.28, 10.1.0.29 10.1.0.30, 10.1.0.31	node1, node2 node3	10.1.0.27 10.1.0.28
Injected Mem Fault	10.1.5.62, 10.1.5.63, 10.1.5.64 10.1.5.65, 10.1.5.66	node4, node5 node6	10.1.5.62 10.1.5.63

running. And Figure 7 (d), (e), and (f) show the utilization changing situation when experiments are conducted, from which we observe the influence of blacklisting and speculation toward system capacity and job execution.

We injected "faults" into the system to simulate a realistic environment with node performance heterogeneity. We tested three cases with different numbers of weak nodes through creating extreme resource contention situations on certain VMs. A CPU intensive program which continuously calculating the π value, and a memory intensive tool which intensively create arrays to occupy memory are developed. The detailed deployment configuration for these three environments is shown in Table 1. In the OpenNebula cluster, VMs are referred with their private IP address while in the ExoGENI cluster, VMs are referred with their host names.

5.2. Implementation

To minimize overhead, we leverage existing interfaces provided by the YARN platform when implementing the DSB framework. Two key components of the DSB system are the *Normalized Execution Calculator* and the *Node Health Checker*. For the former, we use the *History Server Rest API* to collect job execution log details in order to generate the task normalized value within a certain time frame. Attributes of interest include *JobID*, *TaskID*, *AttemptsID* (indicating whether this attempt is a speculation or an original task), *SubmitTime*, *EndTime*, *Status* (success or been killed), *MachineID*, etc. Relevant syntax can be found through the online manual[7] of Apache Hadoop.

For the latter component, the *Node Healthy Checker* mechanism provided by YARN is utilized, which functions through specifying a user-defined script. When the given condition is fulfilled, the script will generate a message with an "ERROR" heading to report the unhealthy status of the node. One example is shown as follows: when the script detects the existence of the flag file, the reported "ERROR" message will be detected by the YARN NM, triggering the built-in mechanism to put this specific node into the blacklist. In other words, no upcoming tasks will be assigned to this node until the next script iteration.

The Weak Node Report Script

```
#!/bin/bash
if [-f machineID.txt] then
echo "ERROR, this node is a weak performance one!"
end if
```

Through this implementation, we make sure that the additional modification toward the default YARN system is minimized. The major overhead comes from three steps: (1) the normalized value calculation; (2) the performance CI calculation; and (3) the P-Cores ranking calculation. The first two calculations are both at linear complexity with task numbers per job and per node. The time window (mapping to the parameter in the REST API) can be adjusted to control the number of history jobs as the input. The last calculation is conducted on the number of nodes, which is small in size compared to the number of tasks (in practice, this number is usually between 100 to 10,000 depending on different cluster scale). Besides, the *Model Builder* and the *Ranking* component can be deployed in other nodes rather than the NameNode to further reduce the overhead, as shown in Figure 6.

We run the Sort job on the same input in the ExoGENI cluster with (1) the default YARN setting, and with (2) the DSB setting configured as no node to be blacklisted, to discuss the DSB overhead. The job execution time result for the default YARN is 299.67s in average, with a StDev of 13.9 (301s, 316s, and 282s), while for the DSB system, duration results are 307s, 283s, and 294s (Avg: 294.67s; StDev: 9.8). This indicates minimal overhead generated by the DSB design.

6. Evaluations

This section evaluates the effectiveness of the proposed DSB framework by measuring three key performance improvements: the node ranking results to show whether the weak nodes are successfully detected; the execution time results to demonstrate the performance in improving job response times; the successful speculation rate results to illustrate how it benefits straggler mitigation. The latter two are chosen because the two common performance metrics for distributed job execution are

[7] http://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/HistoryServerRest.html

Table 2: Job Execution Time Results with DSB and YARN Speculator

	With No Weak Nodes						With Weak Nodes					
	DSB (seconds)			$YARN_{speculator}$ (seconds)			DSB (seconds)			$YARN_{speculator}$ (seconds)		
Execution Time of the Sort Workload	183	160	155	151	154	160	468	451	409	529	531	511
	Avg		166	Avg		155	Avg		443	Avg		524
	StDev		12	StDev		4	StDev		25	StDev		9
	Improvement			***-7.09%***			***Improvement***			***15.47%***		
Execution Time of the WordCount Workload	74	72	71	75	73	76	89	100	109	141	137	106
	Avg		72	Avg		75	Avg		99	Avg		128
	StDev		1	StDev		1	StDev		8	StDev		16
	Improvement			***3.13%***			***Improvement***			***22.40%***		

1) Latency, measuring the execution time, and 2) Cost, measuring the resource usage. In our evaluation, latency is measured in its standard manner while cost is measured in the form of successful speculation rate. Besides these two common indicators, the evaluation toward the node ranking result is needed because this is the key procedure of DSB, and this is also an important measurement for blacklisting based methods.

6.1. Node Ranking Results

Node ranking result is the foundation of the DSB framework which directly influences the system performance. Misidentifying of the weak nodes leads to capacity loss with no benefits toward job execution time and speculation effectiveness. Figure 8 shows the box plot of all tasks' normalized execution value given in Equation (1) per node for the three cluster configurations after an hour, which gives insights toward the node execution ability.

From the results it is observable that, in the first OpenNebula cluster, node 10.1.5.62, 10.1.5.63, 10.1.5.64, 10.1.5.65, 10.1.5.66 are with obvious higher performance CIs (observed from the normalized value distribution, the average and the standard deviation are both larger than the other nodes). This result precisely flagged out all the nodes with memory interference program running on top. The results in the ExoGENI cluster and the OpenNebula 2 cluster exhibit similar trend: for the former, node4, node5, and node6 are identified, while for the latter, the node of 10.1.5.62 and 10.1.5.63 are flagged out. The nodes with injected memory fault are successfully ranked as the worst performed nodes by the DSB system, which is consistent with the fact.

In addition, in Figure 8 (b), it is observable that the nodes with injected CPU fault (namely node1, node2, and node3) are exhibiting the second largest performance CIs, ranked weaker than the rest within the cluster. However in Figure 8 (a) and (c), this observation is not as clear. For example in Figure 8 (a), node 10.1.5.71 performance worse than 10.1.0.28 during the one-hour experiment period. This reveals a fact that, for the YARN system with WordCount and Sort workloads, the contention for memory is the major cause of the straggler behavior rather than the contention for CPU.

6.2. Execution Time Performance

The overall job execution time is dependent on the duration of its last parallel task. When a subset of parallelized tasks is assigned to nodes with poor execution performance, they have a larger chance to become stragglers which lead to an extended job response. For applications that emphasize timing constraints, this response extension may break service QoS and cause late timing failures. After applying the DSB framework in the OpenNebula cluster, we get an improved job response result shown in Table 2.

From the results we see that, for system configuration with no weak nodes (consists of only homogeneous default VMs without any injected faults), the response time improvement is limited: only 3.31% on average for the WordCount job. And for the Sort job, the DSB system even results in deteriorated execution: -7.07% on average. This is because when nodes are exhibiting similar execution performance such as the 3^{rd} to the 5^{th} day in

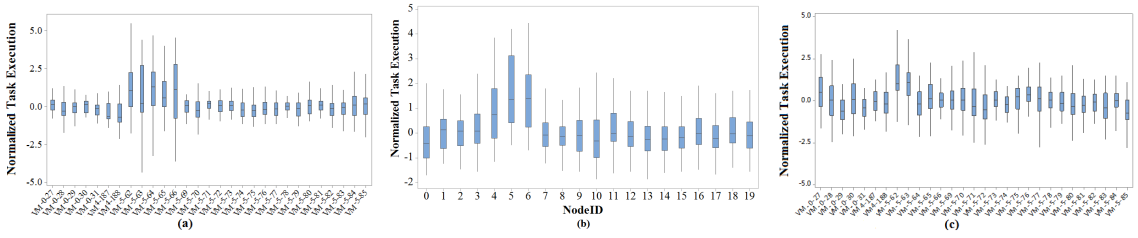


Figure 8: The node performance ranking within (a) the OpenNebula cluster; (b) the ExoGeni cluster; and (c) the OpenNebula 2 cluster

Figure 5, and when cluster size is limited, blacklisting nodes can sometimes hinder system capacity, which is more important for job execution performance compared with the negative impact of stragglers. On the contrary, the DSB framework functions well for the cluster with heterogeneous node execution performance: 15.47% and 22.4% improvement for Sort and WordCount job, respectively, are observed. The improvement value is calculated following Equation (2), with D_{DSB} and $D_{YARN\_speculator}$ stand for the average job duration in the DSB system and in the original YARN system, respectively. The execution average and standard deviation are calculated based on three experiment runs for each test case listed in Table 2.

$$Improvement = \frac{D_{YARN\_speculator} - D_{DSB}}{D_{YARN\_speculator}} \quad (2)$$

The results in Table 2 are generated under the *P-Cores without number* blacklisting policy. That is to say, all level zero nodes identified by the automatic ranking algorithm are blacklisted. During the experiments listed in Table 2, this number is ranging from 2 to 5 in the 30 node OpenNebula cluster, indicating a 7% to 20% weak node percentage. We have tested the additional *P-Cores with number* policy through controlling the parameter of the prohibited node number as well. The job response time in the ExoGENI cluster is evaluated, with the number of blacklisted nodes ranging from 0 to 5. Zero blacklisted node represents the comparison standard of the default YARN performance. The results are detailed in Figure 9. From the results it is observable that, three (15%) is the optimal number of k when determining the number of weak nodes to be blacklisted under this system configuration, with an average improvement for job response time being 55.43%. If we blacklist 5 nodes from this 20-node cluster (in other words, 20%), the execution time will be increased by 35.75%. This is due to the fact that, if the number of the blacklisted nodes is too small, the machine with high straggler occurrence possibility will continue to hinder the job execution, while on the other hand, if this number is too large, it will make the system suffer from the capacity loss. And in this experiment case, the number of three covers all VMs with injected memory fault (weak nodes), which is consistent with the node ranking result.

6.3. Successful Speculation Performance

Another important improvement of the DSB framework is its effectiveness in reducing the killed speculation rate. Bearing the principle of assigning speculative replications to fast nodes to enlarge their chance of surpassing the stragglers, DSB is

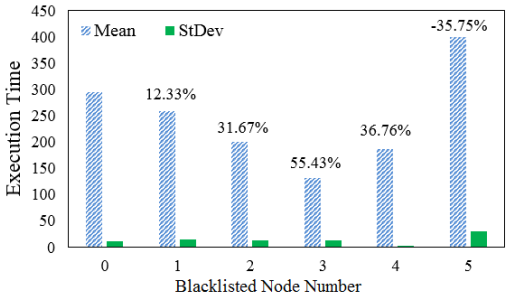


Figure 9: The average job execution time (with standard deviation) with different number of blacklisted nodes in ExoGeni cluster

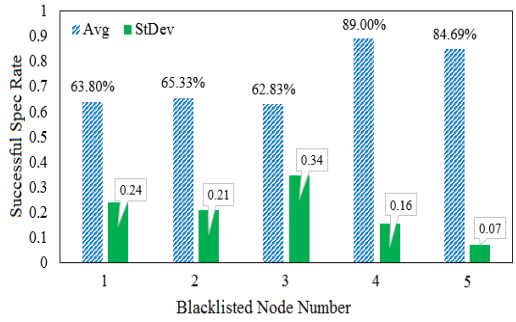


Figure 10: The successful speculation rate with different number of blacklisted nodes in the OpenNebula cluster

effective in improving successful speculation performance. Detailed results are listed in Figure 10. The average successful speculation rate is 63.8%, 65.33%, and 62.83% for 1, 2, and 3 blacklisted nodes (3%, 7%, 10%), respectively. Compared to the current successful speculation rate in real world systems, which is less than 30% as analyzed in previous sections, the performance is doubled.

It is also observable that, after we further blacklist 4 to 5 nodes, the successful speculation rate in DSB system increases to almost 90%. This is because for situations when only a partial of weak nodes are prohibited, there is still a chance for the speculations to be assigned to servers with poor execution performance. And this problem is eliminated after the number of blacklisted nodes covers the majority of the weak ones. All the average and the standard deviation values listed in Figure 10 are calculated based on three execution runs for each case to eliminate randomness. In addition, we did not include evaluation for more than 6 blacklisted nodes in the figure, because the maximum number of weak nodes identified by the DSB framework using the automatic *P-Cores without number policy* is 5, and there is no point to blacklist normal nodes.

7. Conclusion and Future Work

The response time of parallel jobs such as MapReduce can be significantly prolonged by stragglers, due to the fact that the job needs to wait until the last task to finish before it can generate the final outcome. The state-of-the-art straggler mitigation method is speculative execution, which creates task replicas for identified stragglers. However, observations made based on production clusters show that current speculation mechanism has a high failure rate, and the improvement of service response is far from the theoretical optimal.

In this paper we propose DSB, a dynamic server blacklisting framework that avoids scheduling tasks to nodes which exhibiting weak execution performance, through which we manage to decrease susceptibility to straggler occurrence and improve job response time. Core contributions are:

- Demonstrated that current straggler mitigation method is far from effective. Data analytics result based on OpenCloud cluster show that the failed speculation rate reaches as high as 71.22% in average, leads to a dramatic resource waste, and there is still another 65.7% improvement potential for job response times under current speculation scheme. We also demonstrated that machine performance regarding parallel task execution and straggler occurrence are dynamic attributes of server node which chenge over time.
- Proposed a node execution performance modeling and ranking algorithm, which analyze node ability in terms of parallel job execution. Weak nodes influence parallel job execution by enlarging the possibility of the straggler behavior, and can limit speculation efficiency by hindering successful replications. This algorithm helps to identify the weak nodes through dynamically adjust the ranking at each timestamp, and leverages information that collected by the default YARN system to minimise the overhead on additional monitoring.
- Developed a performance-aware dynamic server blacklisting framework. The periodically updated straggler possibility per node analytics accurately reflect the newest system state, and the ranking result enables the enhanced blacklisting as well as the speculation. We have integrated the DSB framework into current YARN system, leveraging the node healthy checker mechanism. Results show that DSB can improve job completion time up to 55.43% compared to the default YARN speculator, and is capable of increasing successful speculation rate up to 89%.

Meanwhile, there remain many challenges, for example, there are too many causes that lead to the straggler problem, and some stragglers cannot be solved purely using speculative execution such as the data skew caused stragglers: because of the replication nature, the newly created speculations would still suffer from the imbalanced input and become a straggler again. We have some initial attempts in identifying root causes for stragglers in [6]. This paper is mainly an optimization for general stragglers that cooperates with the speculative execution scheme, which predicts node performance and avoids assigning tasks/speculations to slow servers. Further improvement could involve developing approaches to find the root cause for each straggler so that DSB could intelligently decide the most appropriate straggler mitigation method to work with, such as the algorithm specifically dealing with reasons such as data skew in MapReduce[38] rather than simply cooperates with speculative execution.

8. Acknowledgement

This work is supported by the China National Key Research and Development Program (No. 2016YFB1000101, No. 2016YFB1000103), and the China National Natural Science Foundation (No.61402514).

References

[1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility, Future Generation Computer Systems 25 (6) (2009) 599–616 (2009).

[2] X. Ouyang, P. Garraghan, C. Wang, P. Townend, J. Xu, An approach for modeling and ranking node-level stragglers in cloud datacenters, in: Services Computing (SCC), IEEE International Conference on, IEEE, 2016, pp. 673–680 (2016).

[3] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE transactions on dependable and secure computing 1 (1) (2004) 11–33 (2004).

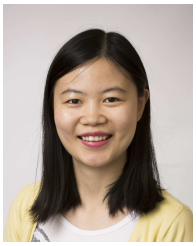
[4] D. McKee, S. Clement, J. Xu, D. Battersby, n-dimensional qos framework for real-time service-oriented architectures, in: 2nd IEEE International Symposium on Real-time Data Processing for Cloud Computing, IEEE Computer Society Press, 2017, pp. 195–202 (2017).

[5] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113 (2008).

[6] X. Ouyang, P. Garraghan, R. Yang, P. Townend, J. Xu, Reducing late-timing failure at scale: Straggler root-cause analysis in cloud datacenters, in: Fast Abstracts in the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN, 2016 (2016).

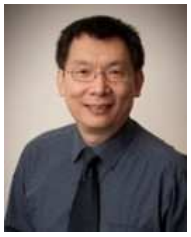
[7] U. Kumar, J. Kumar, A comprehensive review of straggler handling algorithms for mapreduce framework, International Journal of Grid and Distributed Computing 7 (4) (2014) 139–148 (2014).

[8] N. J. Yadwadkar, W. Choi, Proactive straggler avoidance using machine learning, University of Berkeley (2012).
[9] Q. Chen, C. Liu, Z. Xiao, Improving mapreduce performance using smart speculative execution strategy, IEEE Transactions on Computers 63 (4) (2014) 954–967 (2014).
[10] Y. Xu, Z. Musgrave, B. Noble, M. Bailey, Bobtail: Avoiding long tails in the cloud, in: Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), 2013, pp. 329–341 (2013).
[11] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, J. Xu, Fuxi: a fault-tolerant resource management and job scheduling system at internet scale, Proceedings of the VLDB Endowment 7 (13) (2014) 1393–1404 (2014).
[12] J. Tao, J. Kolodziej, R. Ranjan, P. Prakash Jayaraman, R. Buyya, A note on new trends in data-aware scheduling and resource provisioning in modern hpc systems, Future Generation Computer Systems 51 (C) (2015) 45–46 (2015).
[13] G. Ananthanarayanan, A. Ghodsi, S. Shenker, 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 (2013).
[14] K. Jeferry, G. Kousiouris, D. Kyriazis, J. Altmann, A. Ciuffoletti, I. Maglogiannis, P. Nesi, B. Suzic, Z. Zhao, Challenges emerging from future cloud application scenarios, Procedia Computer Science 68 (2015) 227–237 (2015).
[15] W. Jiang, Y. Zhai, Z. Zhuang, P. Martin, Z. Zhao, J.-B. Liu, An efficient method of generating deterministic small-world and scale-free graphs for simulating real-world networks, IEEE Access 6 (2018) 59833–59842 (2018).
[16] X. Wang, J. Su, X. Hu, C. Wu, H. Zhou, Trust model for cloud systems with self variance evaluation, in: Security, Privacy and Trust in Cloud Systems, Springer, 2014, pp. 283–309 (2014).
[17] J. Wang, A. Taal, P. Martin, Y. Hu, H. Zhou, J. Pang, C. de Laat, Z. Zhao, Planning virtual infrastructures for time critical applications with multiple deadline constraints, Future Generation Computer Systems 75 (2017) 365–375 (2017).
[18] H. Zhou, Y. Hu, J. Wang, P. Martin, C. De Laat, Z. Zhao, Fast and dynamic resource provisioning for quality critical cloud applications, in: 2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC), IEEE, 2016, pp. 92–99 (2016).
[19] Y. Hu, J. Wang, H. Zhou, P. Martin, A. Taal, C. De Laat, Z. Zhao, Deadline-aware deployment for time critical applications in clouds, in: European Conference on Parallel Processing, Springer, 2017, pp. 345–357 (2017).
[20] H. Zhou, Y. Hu, J. Su, M. Chi, C. de Laat, Z. Zhao, Empowering dynamic task-based applications with agile virtual infrastructure programmability, in: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), IEEE, 2018, pp. 484–491 (2018).
[21] H. Zhou, Y. Hu, J. Su, C. de Laat, Z. Zhao, Cloudsstorm: An application-driven framework to enhance the programmability and controllability of cloud virtual infrastructures, in: International Conference on Cloud Computing, Springer, 2018, pp. 265–280 (2018).
[22] S. Koulouzis, P. Martin, H. Zhou, Y. Hu, J. Wang, T. Carval, B. Grenier, J. Heikkinen, C. de Laat, Z. Zhao, Time-critical data management in clouds: Challenges and a dynamic real-time infrastructure planner (drip) solution, Concurrency and Computation: Practice and Experience (2019) e5269 (2019).
[23] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, I. Stoica, Improving mapreduce performance in heterogeneous environments., in: OSDI, Vol. 8, 2008, p. 7 (2008).
[24] Q. Yu, M. A. Maddah-Ali, A. S. Avestimehr, Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding, arXiv preprint arXiv:1801.07487 (2018).
[25] S. Dutta, V. Cadambe, P. Grover, Short-dot: Computing large linear transforms distributedly using coded short dot products, in: Advances In Neural Information Processing Systems, 2016, pp. 2100–2108 (2016).
[26] C. Karakus, Y. Sun, S. Diggavi, W. Yin, Straggler mitigation in distributed optimization through data encoding, in: Advances in Neural Information Processing Systems, 2017, pp. 5434–5442 (2017).
[27] M. F. Aktas, P. Peng, E. Soljanin, Effective straggler mitigation: Which clones should attack and when?, ACM SIGMETRICS Performance Evaluation Review 45 (2) (2017) 12–14 (2017).
[28] M. F. Aktas, P. Peng, E. Soljanin, Straggler mitigation by delayed relaunch of tasks, arXiv preprint arXiv:1710.00414 (2017).
[29] J. Dean, L. A. Barroso, The tail at scale, Communications of the ACM 56 (2) (2013) 74–80 (2013).
[30] X. Ouyang, P. Garraghan, D. McKee, P. Townend, J. Xu, Straggler detection in parallel computing systems through dynamic threshold calculation, in: 2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA), IEEE, 2016, pp. 414–421 (2016).
[31] P. Garraghan, X. Ouyang, R. Yang, D. McKee, J. Xu, Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters, IEEE Transactions on Services Computing (2016).
[32] M. Schwarzkopf, Operating system support for warehouse-scale computing, Ph.D. thesis, Citeseer (2015).
[33] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, Large-scale cluster management at google with borg, in: Proceedings of the Tenth European Conference on Computer Systems, ACM, 2015, p. 18 (2015).
[34] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al., Apache hadoop yarn: Yet another resource negotiator, in: Proceedings of the 4th annual Symposium on Cloud Computing, ACM, 2013, p. 5 (2013).
[35] V. Batagelj, M. Zaveršnik, Generalized cores, arXiv preprint cs/0202039 (2002).
[36] K. Ousterhout, P. Wendell, M. Zaharia, I. Stoica, Sparrow: distributed, low latency scheduling, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM, 2013, pp. 69–84 (2013).
[37] A. W. Richa, M. Mitzenmacher, R. Sitaraman, The power of two random choices: A survey of techniques and results, Combinatorial Optimization 9 (2001) 255–304 (2001).
[38] X. Ouyang, H. Zhou, S. Clement, P. Townend, J. Xu, Mitigate data skew caused stragglers through imkp partition in mapreduce, in: Performance Computing and Communications Conference (IPCCC), 2017 IEEE 36th International, IEEE, 2017, pp. 1–8 (2017).



Xue Ouyang received her Ph.D. degree in the School of Computing, University of Leeds. She received her B.Eng. degree in Network Engineering and M.Eng. degree in Software Engineering from National University of Defense Technology (NUDT), China. She is now a lecture in the School of Electronic Sciences, NUDT. Her primary research interest lies in improving performance and efficiency for parallel jobs within large-scale distributed systems, data analytics and machine learning.

Changjian Wang received the B.Sc. degree, the M.S. degree and the Ph.D. degree in computer science from the School of Computer, National University of Defense Technology (NUDT), Changsha, China. He is currently an associate professor in the School of Computer, NUDT. His primary research focus is in the area of Database, Cloud Computing and Machine Learning.



Jie Xu is Chair of Computing at the University of Leeds and Director of the UK EPSRC WRG e-Science Centre. He has industrial experience in building large-scale networked systems and has worked in the field of dependable distributed computing for over 30 years. He is a Steering/Executive Committee member of IEEE SRDS, ISORC, HASE, SOSE, etc. and a co-founder of the IEEE Conference on Cloud Engineering (IC2E). He has led or co-led many research projects to the value of over $30M, and published over 300 research papers.