UNIVERSITY OF LEEDS

This is a repository copy of *Adaptive Speculation for Efficient Internetware Application Execution in Clouds*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/118714/

Version: Accepted Version

Article:
Ouyang, X, Garraghan, P, Primas, B et al. (3 more authors) (2018) Adaptive Speculation for Efficient Internetware Application Execution in Clouds. ACM Transactions on Internet Technology, 18 (2). 15. ISSN 1533-5399

https://doi.org/10.1145/3093896

© 2018, ACM. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in ACM Transactions on Internet Technology, vol. 18, issue 2, Feb 2018.
https://doi.org/10.1145/3093896

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

XX

Adaptive Speculation for Efficient Internetware Application Execution in Clouds

Xue Ouyang, University of Leeds, and National University of Defense Technology
Peter Garraghan, Lancaster University
Bernhard Primas, University of Leeds
David McKee, University of Leeds
Paul Townend, University of Leeds
Jie Xu, University of Leeds

Modern Cloud computing systems are massive in scale, featuring environments that can execute highly dynamic Internetware applications with huge numbers of interacting tasks. This has led to a substantial challenge − the straggler problem, whereby a small subset of slow tasks significantly impede parallel job completion. This problem results in longer service responses, degraded system performance, and late timing failures that can easily threaten Quality of Service (QoS) compliance. Speculative execution (or speculation) is the prominent method deployed in Clouds to tolerate stragglers by creating task replicas at runtime. The method detects stragglers by specifying a predefined threshold to calculate the difference between individual tasks and the average task progression within a job. However, such a static threshold debilitates speculation effectiveness as it fails to capture the intrinsic diversity of timing constraints in Internetware applications, as well as dynamic environmental factors such as resource utilization. By considering such characteristics, different levels of strictness for replica creation can be imposed to adaptively achieve specified levels of QoS for different applications. In this paper we present an algorithm to improve the execution efficiency of Internetware applications by dynamically calculating the straggler threshold, considering key parameters including job QoS timing constraints, task execution progress, and optimal system resource utilization. We implement this dynamic straggler threshold into the YARN architecture to evaluate it's effectiveness against existing state-of-the-art solutions. Results demonstrate that the proposed approach is capable of reducing parallel job response times by up to 20% compared to the static threshold, as well as a higher speculation success rate, achieving up to 66.67% against 16.67% in comparison to the static method.

Additional Key Words and Phrases: Stragglers, Replicas, QoS, Adaptive Speculation, Execution Efficiency

1. INTRODUCTION

Modern Cloud applications are typically deployed in datacenters composed of thousands of heterogeneous servers [Buyya et al. 2009]. Parallel execution models have become dominant within such large-scale infrastructure; technologies such as MapReduce [Dean and Ghemawat 2008], Hadoop [Hadoop 2016] and Spark [Zaharia et al. 2010] decompose jobs into multiple tasks that executed in parallel within different machines in order to leverage system resources and significantly accelerate final job completion times.

A Service Level Agreement (SLA) [Patel et al. 2009] is often enforced in such systems, detailing the level of acceptable service delivered. One important element provisioned by the SLA is the Quality of Service (QoS) [Xu et al. 2016] composed by numerous parameters including performance, timing deadlines and security constraints. For example, soft real-time applications typically emphasize a boundary on acceptable service response time, with violations resulting in late timing failures [Avizienis et al. 2004] that hinder software dependability.

Guaranteeing timely application completion becomes increasingly difficult due to the growing scale and complexity of Cloud systems [García-Valls et al. 2014]. For Internetware, the emerging

This work is supported by the China National Key Research and Development Program (2016YFB1000101, 2016YFB1000103) and the University of Leeds and CSC joint scholarship program.
Author's addresses: X. Ouyang (scxo@leeds.ac.uk), B. Primas (scbjp@leeds.ac.uk), D. McKee (D.W.McKee@leeds.ac.uk), P. Townend (P.M.Townend@leeds.ac.uk) and J. Xu (J.Xu@leeds.ac.uk), School of Computing, University of Leeds, UK; X. Ouyang is also a member of Parallel and Distributed Laboratory, National University of Defense Technology, China; P. Garraghan (p.garraghan@lancaster.ac.uk), School of Computing and Communications, Lancaster University, UK.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2017 ACM. 1533-5399/2017/09-ARTXX $15.00
DOI: 0000001.0000001

software paradigm for Internet computing, application quality assurance (including performance, reliability and usability) also becomes one of the most significant challenges [Mei and Liu 2011] due to its dynamic and emergent properties [Mei et al. 2012].

One challenge that hinders application response performance is the straggler problem [Zaharia et al. 2008], where the completion time of a parallel job is significantly impeded due to a small subset of its parallelized tasks experiencing abnormally longer duration. The delayed tasks which perform slower compared to their sibling tasks are defined as stragglers. It has been identified that stragglers are caused by numerous factors including contention of shared resources, node disk failures and imbalanced task workloads [Kumar and Kumar 2014].

In order to tolerate stragglers, methods such as speculative execution (also known as speculation) [Hadoop 2016] [Zaharia et al. 2008] [Kumar and Kumar 2014] have been adopted, creating replicas for slow tasks to shorten job completion under the assumption that the replicated task will complete prior to the straggler. Stragglers are detected by measuring or predicting when an individual task duration is proportionally greater than the average task execution within a job, expressed as a threshold. Current research and industrial practice adopts this threshold as a pre-defined value, typically 50% larger than the average duration [Zaharia et al. 2008]. However, this static approach comes with a significant limitation – it cannot reflect optimal straggler detection in accordance with job diversity and system operation. Capturing these characteristics allows stricter or more relaxed thresholds for task replication while adhering to desired QoS timing constraints. For example, if the system is exhibiting high utilization, the overhead incurred by additional task replicas will reduce system availability, as well as increase straggler occurrence probability due to higher contention. In contrast, low utilization allows for additional leniency towards speculative replica generation to improve job completion, and benefits jobs that emphasize quick response.

This paper proposes an algorithm that dynamically determines the optimal straggler thresholds for parallel applications in Clouds in order to improve job completion times while reducing resource overhead on speculation under high system utilization. Specifically, our approach factors service QoS timing constraints, task execution progress, and the cluster resource utilization to calculate the optimal value for defining straggler tasks in parallel jobs. Our approach is validated through implementing the system based on YARN [Vavilapalli et al. 2013] and conducting real experimentals representing different operational scenarios on top of the OpenNebula platform [OpenNebula 2016]. Results show that the proposed algorithm is capable of improving job completion against static threshold as well as improving speculation success rate.

The paper is structured as follows: Section 2 presents the background, introducing basic concepts relating to the straggler problem; Section 3 proposes the dynamic straggler threshold algorithm design and formulation; Section 4 illustrates the algorithm with two theoretical examples; Section 5 presents the implementation of the straggler tolerant system based on YARN adopting the dynamic threshold; Section 6 details the experiment setup and the evaluation; Section 7 surveys the related work, detailing different types of threshold in production environments; Section 8 discusses the conclusions and the future work.

Table I: Notation Table.

J_i	The i^{th} job	T_{ik}	The k^{th} task in job J_i
M_j	The j^{th} machine in the cluster	t	Time stamp
C_{ik}^t	The estimated completion time for T_{ik} at time t	$\overline{C_i^t}$	The average estimated completion of J_i at time t
$Th_{i,dyn}^t$	The dynamic threshold for J_i at time t	$Th_{i,stat}^t$	The static threshold for J_i at time t
Q_i^t	The QoS adjustor for J_i at time t	P_i^t	The progress adjustor for J_i at time t
R^t	The cluster resource utilization adjustor at time t	$\overline{PS_i^t}$	The average progress over all tasks of J_i at time t
PS_{ik}^t	The progress score of T_{ik} at time t	α	The progress weight parameter
β	The resource utilization weight parameter	μ	The progress standard parameter
ϕ	The CPU utilization standard parameter	ω	The memory utilization standard parameter
Ω_j^t	The memory requirement of machine M_j at time t	Φ_j^t	The CPU requirement of machine M_j at time t

2. BACKGROUND: THE STRAGGLER PROBLEM

Stragglers that hinder timely parallel job response time have become a concern for both Cloud consumers and system providers, especially for Internetware applications that run in the dynamic operational environments. The "HOW-WELL" issue (mainly refers to application performance including timely response) has become the core issue in the practices and pervasiveness of the Internetware paradigm [Mei 2010]. For jobs that have specified QoS timing constraints, stragglers can lead to late timing failure which decreases application reliability [Ouyang et al. 2016b]. The MapReduce framework is introduced in this section, followed by an explanation of the reasons why such parallel computing frameworks frequently encounter difficulties with stragglers. The influence stragglers impose within production systems has been analyzed, and the basic concepts of task progress and straggler threshold are discussed as well. Table I summarizes the notation used in this paper.

2.1. MapReduce Framework

In order to effectively leverage large-scale cluster infrastructure, programming models such as MapReduce have been proposed [Dean and Ghemawat 2008]. The MapReduce framework mainly consists of the following procedures: 1) *Fork*: the MapReduce library splits input files into M pieces of data chunks stored on HDFS; 2) *Assign map/reduce*: the scheduler selects idle machine nodes to assign map / reduce tasks; 3) *Read*: a mapper reads the contents of the input split and passes each key-value pair to the map function; 4) *Local write*: the intermediate data pairs produced by the map function are written to the local disk; 5) *Remote read*: the reducer is notified by the scheduler about the location of the intermediate results, and it uses a remote procedure call (RPC) to read the data from the mappers local disks; 6) *Write*: the reduce worker iterates over the sorted intermediate data, passes the key and the corresponding intermediate value sets to the users' reduce function. The output of the reduce function is appended to a final output file. Following the above steps, MapReduce framework splits jobs into parallelized tasks running on different machines to improve completion performance.

2.2. Straggler Problem and Causes

To present an example of the straggler phenomenon, Figure 1 shows three completed jobs within a Google cluster [V2 2016] [Reiss and Wilkes 2011]. It is observable that although each job exhibits different task size and duration, each are characterized by a tailing shape, with the slowest task taking up to more than 10 times longer compared to average task duration. In [Dean and Barroso 2013], Google further demonstrates how this straggler problem becomes an increasingly common phenomenon in the face of increased growth of system scale.

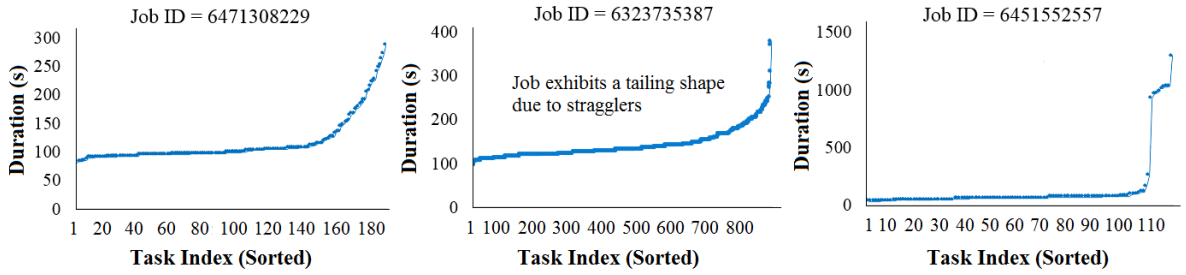

Fig. 1: Task completion pattern for jobs exhibiting straggler phenomena in Google cluster.

Stragglers occur due to multiple causes, ranging from skewed input data size, unbalanced workload, heterogeneous node capacities, shared resource contention, queuing, network congestion [Kumar and Kumar 2014], daemons, maintenance activity, power limits and garbage collection [Dean and Barroso 2013], to fault activation within tasks and servers [Li et al. 2014]. Among these causes, [Wang et al. 2011] categorizes them into internal and external reasons within the context of MapReduce. Internal causes are categorized by behavior which could be addressed by

the MapReduce service provider (e.g. block size and slot number), while external reasons are resultant on user behavior and the system environment (e.g. facility temperature, poor user code, sudden surge in user demand).

2.3. Straggler Influence within Production Systems

Stragglers are not exceptional cases limited to Google, but a common problem in many production systems. We analyzed straggler occurrence from trace data generated by the OpenCloud research cluster at Carnegie Mellon University [OpenCloud 2016]. The cluster is composed of 116 homogeneous nodes, and is used to conduct research in areas such as machine learning, natural language processing and social networking analysis for different schools and departments within the University. After filtering a ten month period tracelog (covering from January to October 2012) running Hadoop, altogether 18,935 jobs and 8,734,974 tasks have been analyzed.

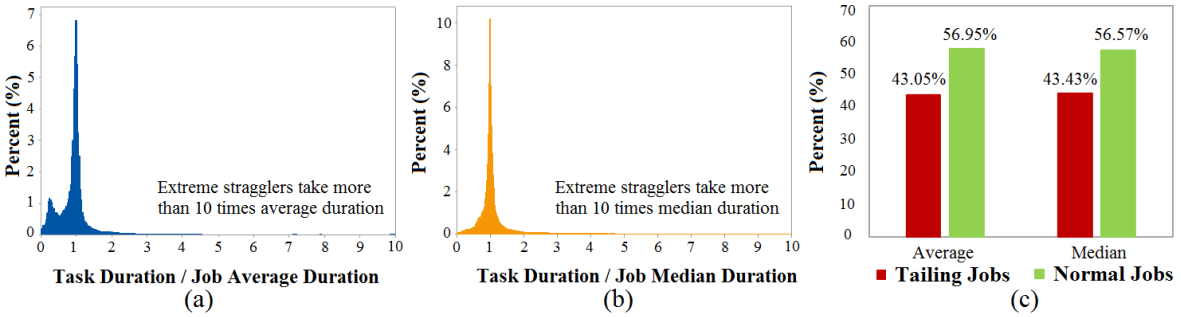

Fig. 2: Straggler statistics in OpenCloud Hadoop cluster compared with (a) average job duration, (b) median job duration, and (c) overall tailing job proportion.

Figure 2 shows straggler occurrence within this OpenCloud cluster and the percentage of jobs that experience stragglers. Figure 2 (a) and (b) show the distribution of individual task duration compared to the mean and median completion of tasks within the same job, respectively. It is observable that tasks exhibit similar percentages of completion around 100%, with a small portion of tasks completing earlier or much later, with the longest being 10 times slower. Approximately, 5% of tasks exhibit straggler behavior. However, because of the cascading effect, when moving to job level, almost half of the parallel jobs are affected (detailed in Figure 2 (c), portraying the percentage of jobs containing stragglers). Such proportions are also identified within other large-scale production Cloud datacenters [Garraghan et al. 2016b], demonstrating that even rare performance abnormalities of stragglers can affect a significant portion of all jobs in Cloud environments.

2.4. Task Progress and Straggler Threshold

In order to identify stragglers, two important concepts are defined: the task progress score (PS) and the straggler threshold. The former measures the execution progress of a task while the latter sets the standard to define to what extent a slow task should be classified as a straggler.

According to [Zaharia et al. 2008], the calculation of PS is given by

$$PS_{ik}^{t} = \begin{cases} L/N & \text{For map tasks} \\ 1/3 * (P + L/N) & \text{For reduce tasks} \end{cases} \quad (1)$$

where PS_{ik}^{t} represents the progress score of task T_{ik} at time t, the k^{th} task of job J_i. The value of PS_{ik}^{t} is bounded between 0 and 1, representing the start and the end point of T_{ik}, respectively. For a map task, PS is the fraction of input data read. In Equation (1), the number of key/value pairs needed to be processed is denoted by N, while L stands for the number of key/value pairs that have already been processed. For a reduce task, the execution is divided into three phases (*copy*, when the task fetches map output; *sort*, when map outputs are sorted by key; and *reduce*, when

the user-defined function is applied to the list of map outputs with each key), and each phase accounts for 1/3 of the final PS (this even weighting can be modified through changing scheduler settings [Chen et al. 2010]). The number of finished phases is represented by P, and within each phase, the score is the fraction of data processed. For example, 0.667 for a map task means that two thirds of key/value pairs have been processed, while 0.667 for a reduce task indicates that it has finished the *copy* and *sort* phases, and will shortly commence the *reduce* phase.

Once the progress score has been collected, a straggler is identified after applying a threshold. If the task's estimated completion time (ECT) is longer than a certain percentage compared to the average value within the same job, it will be identified as a straggler. For example, if the average task completion of a parallel job is estimated to be 100s, a threshold of 200% would result in any tasks that take more than 200s to complete being flagged as stragglers. Changing the threshold value directly impacts the number of stragglers identified. Figure 3 shows how the proportion of stragglers and the corresponding tailing jobs (job containing stragglers) within the OpenCloud cluster are affected by different threshold values ranging from 120% to 260%. Since speculation results in the creation of task replicas upon straggler detection, different straggler numbers will directly impacts the availability of the system and further inflence resource allocation.

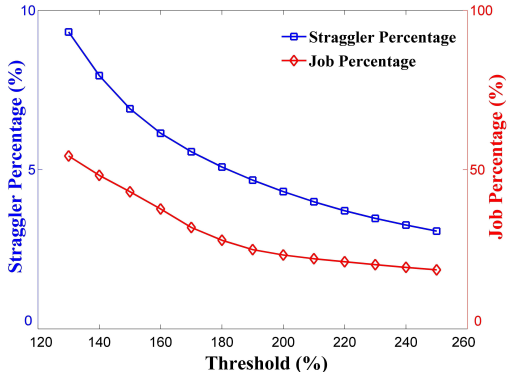

Fig. 3: Relation between threshold setting and straggler/tailing jobs proportion in OpenCloud cluster. Tailing jobs are the ones that containing stragglers.

3. DYNAMIC THRESHOLD CALCULATION MODEL

In order to fill the gap caused by the static threshold and to improve speculation efficiency, we propose an algorithm that calculates a dynamic threshold which can automatically adjust its value according to different operation situations. This algorithm leverages three key factors - job QoS timing constraints, task lifecycle progress, and system resource utilization to indicate when a task replica should be created to tolerate task stragglers.

Figure 4 illustrates the goal of the design and the need for a dynamic threshold. When a job exhibits a strict QoS timing constraint (i.e. where QoS deadline is smaller than 1.5 average completion as shown in Figure 4 (a)), a static threshold will only detect Task A as a straggler to be speculated. However, it is possible for Task C to break QoS constraints, leading to a late timing failure. Therefore, we believe that it would be more effective for the threshold to capture this QoS characteristic in order to create replicas not only for task A but also for task C. Furthermore, if the system load is light, it is possible to also launch additional replicas for Task E due to idle resources available to improve overall application performance. On the other hand, in cases when an application has a lax QoS timing constraint as shown in Figure 4 (b) (with QoS larger than 1.5 average completion), since predicted task completion does not violate the QoS constraint, it is not necessary to create replicas for task A and task C. This is an important consideration when there is already high resource contention within the system. We believe that there is an opportunity to enhance the current threshold approach capable of dynamically capturing these scenarios.

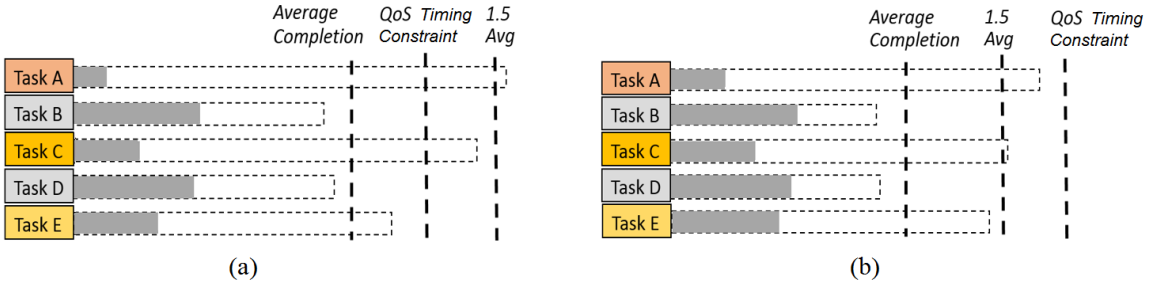

Fig. 4: Dynamic threshold motivation dealing with (a) strict and (b) lax QoS timing constraint.

The dynamic threshold is periodically updated at a certain time interval t in order to allow the algorithm to adapt to the current system environment. The scheduler will label task T_{ik} as straggler if $C^t_{ik} > Th^t_{i,dyn} * \overline{C^t_i}$ is fulfilled. When stragglers are detected, within current Hadoop YARN implementation [Vavilapalli et al. 2013], a duplicate task will then be created and assigned for running using the $addSpeculativeAttempt(taskID)$ function. Equation (2) depicts the calculation of the dynamic threshold value per job at a high level:

$$Th^t_{i,dyn} = Q^t_i + \alpha * P^t_i + \beta * R^t \tag{2}$$

where Q^t_i denotes the threshold baseline determined by job QoS timing constraints. P^t_i is the progress adjustor, altering the value for optimal replica creation based on task lifecycle, and R^t represents the resource adjustor according to the cluster current average resource utilization levels. Weight parameters α and β can be specified by the system administrator to demonstrate a particular emphasize for resource utilization or progress. The weighted sum of Q^t_i, P^t_i and R^t produce the threshold for detecting stragglers.

ALGORITHM 1: DynamicStragglerThreshold

```
Inputs:
        Jobs: the list of all jobs within the cluster
        Tasks: the list of tasks per job
        Interval: threshold calculation interval
        PS: a list of progress score of every task
        alpha, beta: α, β, the adjustor weightings
while Jobs.size > 0 do
    for each Jobs[j] in Jobs do
        Q[j] = TimingConstraintsBaseline;
        P[j] = TaskLifeCycleAdjustor;
        R[j] = UtilizationAjustor;
        Th[j] = Q[j] + alpha*P[j] + beta*R[j];
        for each Tasks[i] in Jobs[j].Tasks do
            PR[i] = PS[i] / (CurrentTime − Jobs[j].startTime);
            ETC[i] = CurrentTime + (1 − PS[i]) / PR[i];
            if (ETC[i] > (Th[j] * average (ETC))) then
                if (Tasks[i].AlreadySpeculated == False) then
                    AddSpecAttempt(Tasks[i]);
                    Tasks[i].AlreadySpeculated = True;
                    Jobs[j].size += 1;
                end
            end
        end
    end
    sleep(Interval);
end
```

ALGORITHM 2: TimingConstraintsBaseline

```
Inputs:
        Job: the parallel job
        Tasks: the list of tasks per job
        PS: a list of progress score of every task
        QoS: QoS timing constraints
Output:
        Q: the timing constraints baseline
for each Tasks[i] in Tasks do
    PR[i] = PS[i] / (CurrentTime − Job.startTime);
    ETC[i] = CurrentTime + (1 − PS[i]) / PR[i];
end
if (QoS ≠ null) then
    Sort(ETC);
    Q = (ETC > QoS).first;
    if (Q ≠ null) then
        Q = Q / average(ETC);
    else
        Q = QoS / average(ETC);
    end
else
    Q = 1.5;
end
```

The pseudo code for this dynamic straggler threshold calculation algorithm is given in Algorithm 1. Every time t, stragglers are identified according to the calculated threshold and their

estimated finish time C_{ik}^t, which is calculated by

$$C_{ik}^t = t + \frac{1 - PS_{ik}^t}{PS_{ik}^t}(t - t_0) \tag{3}$$

where PS_{ik}^t is the task PS recorded at time stamp t for task T_{ik}, while t_0 is the start time of job J_i. A higher value for $Th_{i,dyn}^t$ indicates a stricter straggler threshold enforced at that particular time, in which case fewer task would meet the standard to be defined as stragglers and trigger the speculation, while a lower $Th_{i,dyn}^t$ value allows a more relaxed condition when generating speculated replicas. The values for Q_i^t, P_i^t and R^t are derived by lower levels of calculation.

3.1. QoS Timing Constraints

The QoS timing constraint is an important factor to be considered when deciding how rigorous the time threshold should be based on the nature of the application. For example, a real-time service might emphasize a compulsory response time frame in their QoS. Jobs which fail to complete prior to the specified deadline result in late timing failures and degraded application performance, therefore guaranteeing the rapidness of task execution is more important than saving resources on speculation. In our design, we use this QoS Timing Constraints parameter to set the threshold baseline. This allows for different degrees of strictness for generating replicas when tolerating the impact of task stragglers.

The calculation of the QoS baseline at time t is given in Equation (4)

$$Q_i^t = \begin{cases} QoS/\overline{C_i^t} & \text{if } QoS \geq \max(C_i^t) \\ \min_k\{C_{ik}^t : C_{ik}^t > QoS\}/\overline{C_i^t} & \text{if } QoS < \max(C_i^t) \\ 1.5 & \text{else} \end{cases} \tag{4}$$

where C_{ik}^t represents the ECT for task T_{ik} within the cluster and QoS stands for the time requirement defined as a QoS parameter. To give two examples of how this is calculated, first, assume that a job with QoS timing constraint of 300ms has five tasks with ECTs at time t to be 290ms, 290ms, 300ms, 380ms, 400ms, respectively. In this scenario, the minimal C_{ik}^t greater than QoS is 380ms, and the average ECT is 300ms. Therefore the value for Q_i^t to be used calculating the final threshold $Th_{i,dyn}^t$ is 127% (380 ÷ 300). If all tasks are estimated to complete prior to the specified QoS (change QoS in this example from 300ms to be 450ms, then all C_{ik}^t values will be smaller than QoS), the value for Q_i^t will then change to 150% (450 ÷ 300). This results in no tasks detected as stragglers (if P_i^t and R^t are zero as in this example).

It is worth highlighting that the dynamic straggler threshold algorithm also functions well for applications that do not have an explicit QoS timing request specified. In such an event, a static time proportion value of 150% used in current literatures [Zaharia et al. 2008] [Rosen 2012] [Kwon et al. 2012] can be applied to set the threshold baseline, and the dynamic change for $Th_{i,dyn}^t$ will then depend on P_i^t and R_i^t. Algorithm 2 details the calculation process.

ALGORITHM 3: TaskLifeCycleAdjustor

Inputs:
Job: the parallel job
Tasks: the list of tasks per job
PS: a list of progress score of every task
mu: μ, the progress threshold
Output:
P: the task lifecycle adjustor

```
sumPS = 0;
for each Tasks[i] in Tasks do
    sumPS += PS[i];
end
P = sumPS / Job.size − mu;
```

ALGORITHM 4: UtilizationAdjustor

Inputs:
phi, omega: ϕ, ω, the CPU and the memory threshold
C_Uti, M_Uti: the nodes CPU and memory utilization
Nodes: the machine nodes list within the cluster
Output:
R: the utilization adjustor

```
sumCPU = 0, sumMem = 0;
for each Nodes[n] in Nodes do
    sumCPU += C_Uti[n];
    sumMem += M_Uti[n];
end
C_Adjustor = sumCPU / Nodes.size − phi;
M_Adjustor = sumMem / Nodes.seze − omega;
R = max(CPUadjustor, MEMadjustor);
```

3.2. Task Lifecycle Progress

It is also important to consider the current execution progress completed for effective replica generation. Specifically, a replica should ideally be spawned at an early phase of the task lifecycle when it is likely to complete prior to the task straggler, otherwise the replica will likely result in unnecessary resource consumption with no improvement towards job completion time. For example, when a task experiences slow down in its later phase, the newly created replica has less probability to complete prior to the straggler as it is already too late to catch up. As a result, it is reasonable to increase the threshold in response to late progress to avoid ineffective speculation, and lower the threshold value at early phase within task lifecycle to encourage replica generation, because the replica should have a higher probability to outpace the original task in that case. Adhering to this reasoning, it is important to consider current task execution progress when launching speculative replicas.

The calculation of the progress adjustor at time t is given in Equation (5)

$$P_i^t = \overline{PS_i^t} - \mu \tag{5}$$

where $\overline{PS_i^t}$ is the average progress score for job J_i at time t, representing the current task lifecycle. Progress standard parameter μ is used to denote the specified maximum point within the whole lifecycle suitable for generating a replica. For example, $\mu = 0.5$ represents that any task with a progress score smaller or equal to 0.5 will still be considered as early stage, leading to a smaller $Th_{i,dyn}^t$ by generating a negative P_i^t value, increasing the likelihood of replica generation. And any tasks that progress past half of the entire lifecycle will be treated as in their late stages, resulting in a positive P_i^t value to higher the threshold to limit replica generation. This procedure is detailed in Algorithm 3.

3.3. System Resource Usage

An important consideration for straggler tolerant systems is the overhead incurred by speculations toward different conditions. Creating replicas in a high system resource utilization situation poses a greater threat to system stability and can further increase the likelihood of straggler occurrence [Ouyang et al. 2016b], while low system utilization allows for additional speculation to improve job completion. Furthermore, replicas themselves also have the potential to become stragglers. Observations proposed in [Ananthanarayanan et al. 2010] state that 3% of replica executions still take ten times longer than normal task executions in Bing's production cluster. Considering the fact that replicas will execute with data identical to the original straggler and will be configured with the same resource requests, the expense of tasks should also be considered when deciding whether to perform speculation. If the resource requirement of the original task is high, then generating a corresponding replica can result in a higher resource cost with no substantial improvement towards overall job completion. Based on this reasoning, the dynamic straggler threshold calculation should consider current levels of system resource utilization.

The resource adjustor is represented as parameter R^t in the algorithm to tune the value of $Th_{i,dyn}^t$ dependent on system utilization at time t. This calculation is given as

$$R^t = \max\left(\frac{\sum_{j=1}^{n}\Omega_j^t}{n} - \omega, \frac{\sum_{j=1}^{n}\Phi_j^t}{n} - \phi\right) \tag{6}$$

where n denotes the number of servers within the cluster, while Ω_j^t and Φ_j^t represent the memory and CPU utilization of machine M_j, respectively. If either one of these two parameters hits the optimal utilization specified by the user (memory standard ω and CPU standard ϕ), the equation will increase the threshold by generating a positive R^t value, resulting in stricter requirements for replica generation. Algorithm 4 summarizes this process.

4. THEORETICAL EXAMPLES

In this section, we explain the key idea of the proposed algorithm by giving two theoretical examples, illustrating how the threshold value changes according to different parameters and what impact this change would exhibit upon final job completion. The progress scores are given

one timestamp at a time. One example summarizes the case when the job starts in a low system utilization state while the other is for high utilization.

We apply static and dynamic threshold approaches to both examples and the results are presented in four tables. In both examples, we consider an instance of a single job J consisting of ten tasks $T_1, \ldots, T_{10}$ for illustration simplicity. The progress standard parameter, the memory and CPU standard parameter are set to be $\mu = 0.5, \omega = 0.6$, $\phi = 0.6$, respectively, and values for weighting parameters are $\alpha = 0.5$ and $\beta = 0.5$. This setting represents a common configuration and can be changed according to different interests.

4.1. Example 1: Job Starts in Low Utilization Environment

Table II shows the job PSs and the ECTs at each time interval for the (a) dynamic and (b) static threshold, respectively. The initial system utilization starts from 15% to represent idle cluster conditions. We assume the job does not have an QoS deadline in this example, and the values for $T_1, \ldots, T_{10}$ in the table are ECTs of the initial tasks, while R-T_i stands for corresponding replicas.

Table II: Example 1 job speculative execution with (a) dynamic threshold, (b) static threshold.

Time	PS	Uti	Th	Th*$\overline{ECT}$	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	R-T_3	R-T_5	R-T_7	R-T_{10}
1	0.14	0.15	1.09	8.20	7	7	8	7	8	5	8	8	7	**10**				
2	0.25	0.15	1.15	9.31	7	6	9	6	**10**	6	**13**	7	7	10				8
3	0.31	0.20	1.21	10.58	7	8	**13**	8	10	7	15	7	7	9		8	8	7
4	0.44	0.50	1.42	11.77	7	7	12	8	10	6	16	7	5	9	6	8	7	8
5	0.57	0.55	1.51	12.39	8	7	12	8	9	6	14	6	6	10	7	7	7	8
6	0.71	0.55	1.58	12.74	7	6	11	7	10	6	14	7	7	9	7	8	6	8
7	0.82	0.55	1.63	13.19	7		13	7	10		13	7	7	10	7	7	6	7
8	0.93	0.25	1.54	12.45			12		11		~~14~~			~~10~~	6	7	6	7
9	0.97	0.20	1.54	12.40			12		~~10~~						7	7		
10	1.00	0.20	1.55	12.40			~~11~~								7			

(a)

Time	PS	Uti	Th	Th*$\overline{ECT}$	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	R-T_3	R-T_7
1	0.14	0.15	1.50	11.25	7	7	8	7	8	5	8	8	7	10		
2	0.26	0.15	1.50	12.15	7	6	9	6	10	6	**13**	7	7	10		
3	0.33	0.30	1.50	13.50	7	8	13	8	10	7	15	7	7	9		8
4	0.49	0.30	1.50	12.82	7	7	12	8	10	6	16	7	5	9		7
5	0.61	0.35	1.50	12.68	8	7	12	8	9	6	14	6	6	10		7
6	0.76	0.40	1.50	12.27	7	6	11	7	10	6	14	7	7	9		6
7	0.85	0.35	1.50	12.55	7		**13**	7	10		13	7	7	10		6
8	0.86	0.25	1.50	12.38			12		11		~~14~~			10	6	
9	0.90	0.20	1.50	12.38			12		10					10	7	
10	0.95	0.20	1.50	12.13			11		10					10	6	
11	0.96	0.20	1.50	12.38			12								7	
12	1.00	0.20	1.50	12.38			12								~~7~~	

(b)

Observed from Table II (b), the static threshold identifies two stragglers and the job completes at timestamp 12. Task T_7 is the first straggler identified, being detected at timestamp 2 when its ECT is found to be larger than $1.5 * \overline{ECT}$. Replica R-T_7 is created upon detection, and its ECT is calculated at time 3 when PS is first generated. The other straggler identified is task T_3 with R-T_3 created at timestamp 7. Among the two replicas, R-T_7 completes at time 8 prior to task T_7, however for T_3, due to the late detection, R-T_3 does not have sufficient time to catch up. From this demonstration we see that the static threshold cannot capture slow tasks such as T_3 promptly; even in its early stage, it already shows slow progress (at timestamp 3 when its ECT is 13, however this number does not pass the 1.5 threshold because it is smaller than 13.5).

For the dynamic threshold demonstrated in Table II (a) this gap of late detection is reduced. Altogether four stragglers are identified to shorten the job execution from timestamp 12 to 10. Although the larger replica number imposes a greater overhead, due to the idle system state, it will not cause severe burden towards the system. It is also observable that the threshold value gets larger through job execution. This is due to two reasons: first, the dynamic approach encourages replica creation in early stages of the life cycle by generating smaller threshold values according to Equation 5. Second, as the system utilization is low when the job executes, the usage of the resource adjustor defined in Equation 6 yields even smaller threshold values. This is why tasks T_3, T_5, T_7 and T_{10} are identified in comparison to only T_3 and T_7 using the static approach. The dynamic approach makes use of the low system utilization to create more replicas in a prompt way so that a better response time is obtained.

4.2. Example 2: Job Starts in High Utilization Environment

Table III shows applications that adopt the (a) dynamic and (b) static straggler thresholds for example 2, in which job starts at a high utilization state. This time a job with a QoS constraint

$QoS = 12$ is used to demonstrate the algorithm methodology. Other attributes including task number and the algorithm parameter settings are identical to the previous example.

Table III: Example 2 job speculative execution with (a) dynamic threshold, (b) static threshold.

Time	PS	Uti	Th	Th*$\overline{ECT}$	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	R-T_8
1	0.14	0.80	1.52	11.39	8	7	7	8	7	5	8	8	7	10	
2	0.26	0.80	1.59	12.85	9	6	7	10	6	6	7	**13**	7	10	
3	0.37	0.95	1.68	13.91	13	7	6	10	7	6	6	13	6	9	8
4	0.59	0.90	1.84	13.41	11	6	5	11	5	6	5	12	5	7	7
5	0.66	0.85	1.96	15.65	11	6	6	12	6	6	6	14	6	8	7
6	0.75	0.80	1.88	15.88	11	6	7	13	7	6	7	14	7	9	6
7	0.84	0.80	1.79	15.29	13		7	12	7		7	13	7	10	6
8	0.93	0.50	1.82	15.38	12			11				~~14~~		10	6
9	0.95	0.30	1.48	12.62	12			12						10	
10	0.98	0.35	1.43	11.98	11			11						10	
11	0.98	0.20	1.46	12.36	12			12							
QoS =12	1.00	0.15	1.43	12.21	12			12							

(a)

Time	PS	Uti	Th	Th*$\overline{ECT}$	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	R-T_1	R-T_4	R-T_8
1	0.14	0.80	1.50	11.25	8	7	7	8	7	5	8	8	7	10			
2	0.26	0.80	1.50	12.15	9	6	7	10	6	6	7	**13**	7	10			
3	0.37	0.95	1.50	12.41	**13**	7	6	10	7	6	6	13	6	9			7
4	0.55	0.90	1.50	10.75	11	6	5	**11**	5	6	5	12	5	7	7		7
5	0.59	0.95	1.50	11.77	11	6	6	12	6	6	6	14	6	8	7	7	6
6	0.69	0.90	1.50	12.35	11	6	7	13	7	6	7	14	7	9	7	6	6
7	0.79	0.85	1.50	12.35	13		7	12	7		7	13	7	10	6	5	6
8	0.91	0.70	1.50	12.00	12			11				~~14~~		10	7	6	6
9	0.93	0.65	1.50	12.35	12			12						10	7	7	
10	0.98	0.50	1.50	12.23	~~11~~			11						10	7	7	
11	1.00	0.50	1.50	12.35				~~12~~								7	
QoS =12	1.00	0.00	1.50	12.46													

(b)

As presented in Table III (b), due to the slow progress, the static approach identifies three tasks T_1, T_4 and T_8 as stragglers regardless of the utilization level, followed by the creation of replicas R-T_1, R-T_4 and R-T_8 at times 3, 4 and 2, respectively. These additional replicas further increase utilization, potentially increasing the probability of straggler occurance. In addition, some of them do not generate obvious improvement toward execution performance, for example R-T_1, R-T_4 are both only one step earlier than the original task. For the dynamic method demonstrated in III (a), due to the awareness of the environmental conditions through using the resource adjustor defined in Equation 6, it only identifies the most noticeable straggler T_8. As a result of taking QoS into consideration ($QoS = 12$), the slower ECT of T_1 and T_4 (12) are ignored as a trade off for better resource efficiency while still guarantees acceptable response time. To summarize, the dynamic threshold creates less replicas in the case of a high system utilization to avoid overload of the system while garanteeing the fulfillment of QoS requirement.

5. IMPLEMENTATION

Considering the decentralized nature and its advantage in scaling compared to Hadoop V1, we build our straggler tolerant system on top of Hadoop YARN [Vavilapalli et al. 2013]. The default YARN system consists of three main components shown in Figure 5: Application Master (AM), Resource Manager (RM) and Node Manager (NM). By spliting the original Hadoop V1 JobTracker into AM and RM, YARN is capable of de-coupling resource management functionalities (such as request and release containers) with application management responsibilities (such as job scheduling and monitoring). NM is a per-node slave in charge of launching containers, reporting resource usage back to RM and heartbeat. AM is responsible for handling job execution including MapReduce job creation, request resources from RM, communicate with NM to run containers, monitor job running status and do speculation, etc. In accordance with this implementation, we give our system design as follows.

5.1. System Model

In our design, the Speculator residing within the AM is responsible for straggler detection, and consists of two components: the Progress Monitor which is responsible for recording and reporting task progress and the Threshold Calculator which generates the straggler threshold dynamicly. The Executor is developed aside the scheduler to launch replicas in our design, which is detailed in Figure 5.

5.2. Default Speculator Component

The default speculation component in current YARN 2.5.2 [Hadoop 2016] implementation mainly consists of three key classes: the TaskRuntimeEstimator which is responsible for esti-

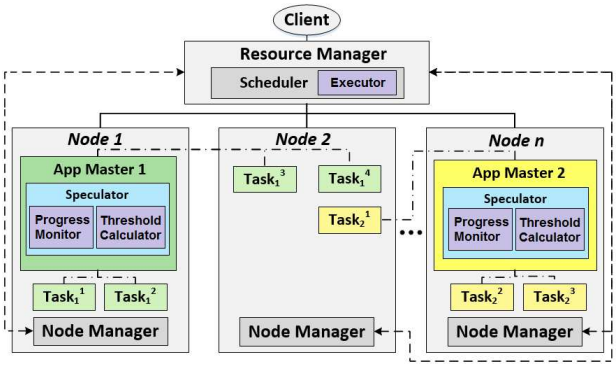

Fig. 5: Hadoop YARN architecture with embedded Straggler Tolerant System which contains Dynamic Threshold Calculator component.

```
<property>
    <name>mapreduce.speculative.progess.para</name>
    <value>0.5</value>
</property>
<property>
    <name>mapreduce.speculative.memory.para</name>
    <value>0.7</value>
</property>
<property>
    <name>mapreduce.speculative.progess.weit</name>
    <value>0.5</value>
</property>
<property>
    <name>mapreduce.speculative.memory.weit</name>
    <value>0.5</value>
</property>
```

Fig. 6: Parameter configuration example.

mating task ECTs; the AppContext in charge of sharing information between objects; and the Speculator. Key methods and parameters are detailed in Figure 7. Every time when the "*speculation*" event is triggered (by default, this time interval is set to be 1 second after no speculation and 15 seconds after speculation), the Speculator will check whether a speculation action should be launched according to conditions given by the parameters including the *minimum_allowed_speculative_tasks* (default value is 10), the *proportion_total_tasks_speculatable* (default value is 0.01) and the *proportion_running_tasks_speculatable* (default value is 0.1). If every condition is fulfilled, the Speculator will calculate the speculation value (SV) as:

$$SV = ECT - ERCT \tag{7}$$

where *ERCT* is short for estimated replacement completion time. This represents the total time it can save from launching a replica for that specific task. Once this value is calculated for each task, the scheduler will create a replica for the task with the largest speculation value by triggering the event of adding the task ID into a waiting queue.

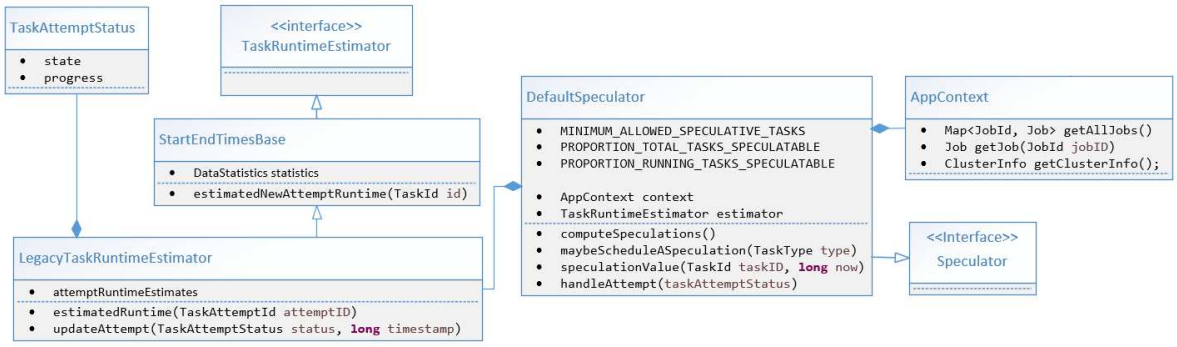

Fig. 7: Class diagram of the speculator component.

5.3. Speculator Modification

Based on the above analysis of the YARN architecture and the default speculator component, it is known that Hadoop already includes several reporting counters such as the progress info for each task attempt in *TaskAttemptHistoryStatistic* class, and provides functions to calculate several key intermediate results such as the estimated task completion. Therefore, our algorithm does not introduce additional monitoring and computation overheads to the system. The only "extra work" performed is at O(1) cost, calculating $Th^t_{i,dyn}$ according to Equation (2) to replace

the "SpeculationValue" function described in the previous section. All external APIs for the Speculator class remain unchanged. For the $Th^t_{i,dyn}$ calculation, P^t_i is straight forward, since all PS values are recorded, it is simple to get the average. As for the resource utilization R^t, two extra attributes are added in the AppContext class (namely "cluster capacity" and "cluster available") in order to record the available CPU and memory resources from *RMContainerAllocator*, which is responsible for communicating with the RM to get the cluster information for AM.

In addition, the parameters used in the calculation can be passed in through modifying the mapred-site.xml file like other YARN related parameters (an example is given in Figure 6). In this way, the algorithm can easily be customized without recompiling the whole YARN platform each time new configurations are added. An automatic log extract tool is developed to locate the precise log position within the cluster and visualize key information such as threshold value, replica number and job execution time.

6. EVALUATION

We developed the straggler tolerant system based on the proposed dynamic threshold algorithm. The system model is illustrated in Figure 5 to evaluate the algorithm effectiveness.

6.1. Experiment Setup

Our experiments were deployed on the OpenNebula platform [OpenNebula 2016] with a typical virtual machine (VM) configuration of 1 GB memory, 1 virtual core with 2.34 GHz capacity and 10GB disk space on potentially shared hard drive. The VM uses KVM virtualization software and runs an Ubuntu 12.04 x86_64 operating system. In all experiments, we configured the HDFS to maintain two data replicas of each chunk. The job types we run include Sort, WordCount, and Hive query (Group By). These three benchmarks were selected as they are frequently used for straggler evaluation, such as in the Google paper [Dean and Ghemawat 2008], in LATE [Zaharia et al. 2008], in Mantri [Ananthanarayanan et al. 2010] as well as in MCP [Chen et al. 2014]. In addition, a number of features of Sort make it a desirable benchmark [Blelloch et al. 1993]. For the Sort and the WordCount, the input size is 10GB and 5GB, respectively, while for the Hive, the Group By query is conducted on a table with more than 10 million rows.

Table IV: Cluster configurations.

Node Num	5	10	30
Default VM	3	15	22
I/O injected	0	1	1
CPU injected	0	1	2
Mem injected	1	1	2
Combined	1	2	3

Table V: Results for different threshold performance.

Job Type	Cluster Size	Response Time (s)					
		Dynamic		Static		No Speculation	
		Avg	Stdev	Avg	Stdev	Avg	Stdev
Word Count	5	103	1.69	110	4.03	107	2.49
	10	96	3.09	96	1.69	98	4.32
	30	63	4.19	75	5.25	88	4.92
Average		87.34		93.67		97.67	
Sort	5	1089	2.16	1164	1.89	1201	3.27
	10	571	1.25	626	2.49	712	4.19
	30	400	3.09	500	2.49	580	4.78
Average		686.67		763.34		831	
Hive Groupby	5	67	9.09	82	0.73	80	0.59
	10	61	2.21	73	0.23	77	1.48
	30	56	1.89	61	1.02	64	0.44
Average		61.33		72		73.67	

For the cluster size, we evaluate the system with 5 data nodes, 10 data nodes and 30 data nodes. We injected faults and extreme resource contention situations into the system to reveal a more realistic environment for the experiments. An I/O intensive program (mainly consisting of the "*dd*" and "*rm*" command to create and delete files that take up most I/O throughput of the machine), a memory intensive tool (which intensively creates arrays to occupy memory) as well as a CPU intensive program (which continuously calculates the π value) are deployed on specific VMs. The machine number settings are listed in Table IV, with "Default VM" representing VMs without injected faults. "I/O injected", "CPU injected" and "Mem injected" refer to VMs with a certain interfere program deployed, while "Combined" indicates VMs with all three (I/O intensive, CPU intensive and memory intensive) applications deployed.

6.2. Experiment Results

We analyze the experiment results from four aspects: the performance improvement which focuses on job response time; the speculation overhead which measures the replica number generated; the speculation success rate which calculates replica number that actually outpace the straggler in the system; and the parameter sensitivity which observes how the threshold value will change along with time and utilization.

6.2.1. Performance Improvement.

Three platforms are deployed in order to compare their efficiencies in job completion with straggler presence: Hadoop YARN with 1) the static and 2) the dynamic straggler threshold, and 3) Hadoop YARN with the speculation mechanism forbidden. For two platforms A and B, we calculate the improvement of job response time I_{AB} for platform A versus platform B by

$$I_{AB} = \frac{JC_B - JC_A}{JC_B}, \tag{8}$$

where JC_A and JC_B denote the average completion time of benchmark jobs running on platform A and B, respectively. Table V shows the results for both workload types. Each experiment case was executed three times in order to determine the average and standard deviation value. We keep the total input size constant throughout different cluster sizes (e.g. for Sort, the files of 10Gb in size have been sorted, therefore in 5 VM cluster, each node holds a 2Gb file while for 10 VM cluster, each node only holds 1Gb). From this result, it is shown that the average job response time for WordCount using dynamic straggler threshold speculation is 87.34s, while a static threshold and no speculation is 93.67s and 97.67s, indicating an improvement of 6.76% and 10.58%, respectively. Figure 8 (a) summarizes the improvement for the dynamic and static thresholds versus no speculation in different cluster sizes. It is observable that in the five node cluster, static speculation performs worse than no speculation, leading to a negative improvement value. This is a result of the system experiencing an extremely high utilization state, with additional replication sometimes further burdening the system instead of causing performance improvements. The dynamic threshold on the contrary captures this information, thus obtaining a better performance. Similar negative improve case is observed for Hive in 5 node cluster as well (static threshold VS no speculation). The improvement is more apparent for the Sort workload in comparison to Wordcount and Hive, achieving 10.04% and 17.37% on average when compared with the static threshold and no speculation framework. The most noticeable improvement is achieved when the system is in an idle state (under cluster size 30, the achievement versus no speculation is 31.03% while 20% compared with static threshold). This is consistent with the algorithm design of creating more replicas to reduce job execution time when utilization is low. In busy states (such as cluster sizes 5 or 10), although the dynamic threshold identifies less replicas, it reduces the chance of tasks becoming stragglers due to contention as well, therefore still performs better than the static method.

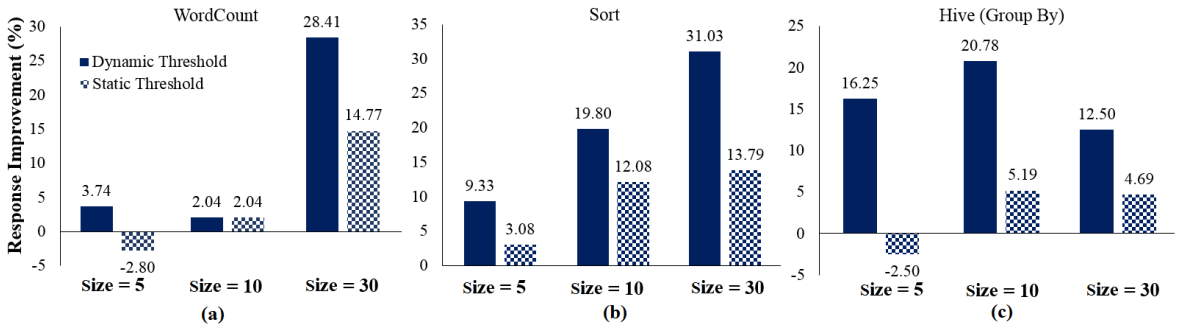

Fig. 8: Job response time improvement of the dynamic threshold and the static threshold comparing to no speculation for jobs (a) WordCount, (b) Sort, and (c) Hive.

Table VI: T test results for response time difference significance.

	wd5		wd10		wd30
ws5	$p = 0.089$	ws10	$p = 0.5000$	ws30	$p = 0.038$
wn5	$p = 0.079$	wn10	$p = 0.289$	wn30	$p = 0.006$
	sd5		sd10		sd30
ss5	$p = 0.000$	ss10	$p = 0.001$	ss30	$p = 0.000$
sn5	$p = 0.000$	sn10	$p = 0.000$	sn30	$p = 0.000$
	hd5		hd10		hd30
hs5	$p = 0.069$	hs10	$p = 0.008$	hs30	$p = 0.026$
hn5	$p = 0.089$	hn10	$p = 0.002$	hn30	$p = 0.013$

In addition, we observe that the execution time performance improvement varies not only with cluster sizes, but with different workload types as well. Table VI details the significance value p after applying independent t-tests between different algorithms. wd5 (ws5, and wn5) in the table represents the Wordcount workload with the Dynamic straggler threshold (Static threshold, and No speculation) running in a 5 node cluster, while sd5 and hd5 represents workload Sort and Hive under the same condition (dynamic threshold, and 5 node cluster). The null hypothesis of the tests are $mean(dynamic) = mean(static)$ and $mean(dynamic) = mean(no\_speculation)$, while the alternative hypothesis are $mean(dynamic) < mean(static)$ and $mean(dynamic) < mean(no\_speculation)$, representing dynamic threshold performs better as it gives a shorter response time. From the results it is observable that, for Sort jobs, the mean difference is quite significant, all tests reject the null hypothesis and admit the fact (with 95% confidence) that the dynamic threshold provides a quicker job response. While for WordCount and Hive jobs, some of the p values are larger than 0.05, indicating a vague improvement. This is due to a limitation of all speculation based techniques which will be analyzed in detail in Section 6.2.3.

6.2.2. Speculation Overhead.

Further comparison regarding the speculation overhead is conducted between static and dynamic threshold methods, primarily measuring the number of replicas generated under each algorithm. The detailed results are listed in Table VII, from which we see that if the cluster size is small, the dynamic threshold can save resources through creating less replicas compared to the static algorithm. In contrast, the dynamic threshold in a larger cluster size will generate more replicas as a result of trading resources for time to achieve better response performance. This auto adjustment is important, especially for jobs with QoS timing constraints.

Table VII: Experiment results for speculation overhead comparison.

Workload Type	Cluster Size	Task Number	Replica Number		Successful Speculation		Speculation Effectiveness	
			Dynamic	Static	Dynamic	Static	Dynamic	Static
Word Count	5	8	3	6	2	1	66.67%	16.67%
	10	14	5	6	3	2	60%	33.33%
	30	36	12	5	5	1	41.67%	20%
Sort	5	89	4	8	2	2	50%	25%
	10	110	16	10	5	2	31.25%	20%
	30	153	23	11	12	3	52.17%	27.27%
Hive Group By	5	8	2	1	1	0	50%	0%
	10	13	3	2	2	1	66.67%	50%
	30	33	5	3	3	1	60%	33.33%

6.2.3. Speculation Effectiveness.

We measure speculation effectiveness by comparing successful speculations (replicas successfully overpace the straggler and contribute to the response time improvement) with total speculations launched using Equation (9).

$$Effectiveness = \frac{SuccessfulSpeculationNumber}{TotalReplicaNumber} \tag{9}$$

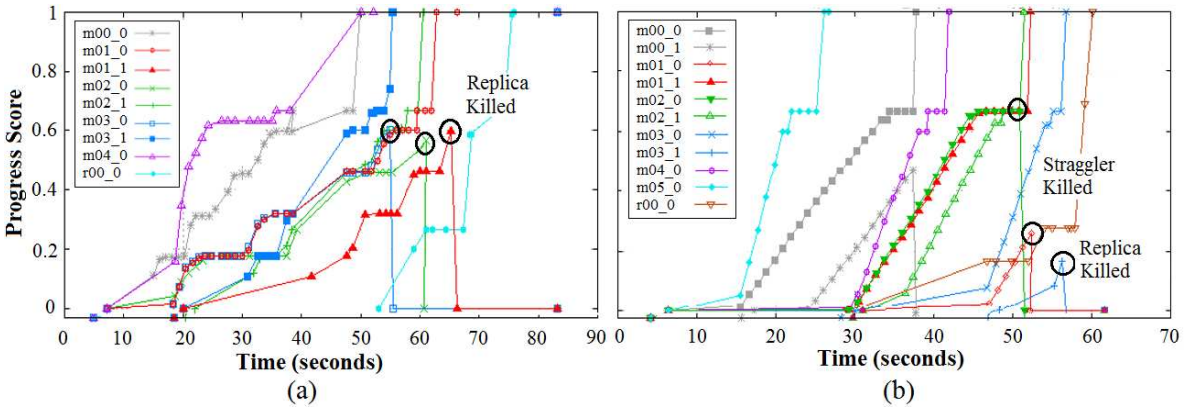

Fig. 9: WordCount task progress in (a) no fault injection, (b) I/O contention fault injection cluster.

We observe in Table VII that the speculation success rate is much higher for Sort jobs compared to WordCount and Hive. This indicates that more replicas outpace the identified stragglers for Sort, and contribute towards improving job execution time. This is due to a limitation of the speculation mechanism; when stragglers are caused by imbalanced workload or uneven input data sizes, speculative copy will still suffer from the same reason and ended up being killed. This can be improved by importing straggler reason-aware mechanisms with mitigating methods such as SkewTune [Kwon et al. 2012] and micro-task technique [Rosen 2012].

The successful replicas for WordCount are found when the testbed has been injected with faults, where stragglers are caused by contention reasons. Figure 9 shows the comparison when both are running the WordCount job using dynamic thresholds in different testbed settings, with Figure 9 (a) contains only default VMs and 9 (b) has three VMs injected with I/O contention. Each color in the figure represents a task attempt, while the ones with suffix zero indicate original tasks and suffix one means it is a speculation of this specific task. In Figure 9 (a), it is observable that all speculative replicas are killed by the system because the stragglers finish first, while in Figure 9 (b), two out of three speculations succeeded in the end overpaceing the stragglers (only task $m03$ has a failed speculation $m03\_1$).

6.2.4. Parameter Setting Sensitivity.

The efficiency of the algorithm is dependent on selecting appropriate values for configurable system parameters. This section studies how the threshold value changes to reflect different system conditions and describes how different parameter settings influence algorithm performance.

Systems have different standards to judge their own “idle” or “busy” state, and different values will lead to differing strictness of creating speculative replicas. The system administrator can also adjust different emphasis toward progress adjustor and resource adjustor. Figure 10 plots the map task threshold changing patterns of two sets of parameter settings for Sort job as an example. The α and β for threshold_1 are both 0.5, representing an equal weighting toward task progress phase and resource utilization level. μ is set with value 0.5, indicating the halfway progress point, and ω is set to be 0.7, meaning any utilization below 70% will be treated as “idle”. In the experiments, ϕ is not used as the RM in YARN 2.5.2 only focuses on memory for now. According to the official website [Hadoop 2016], CPU will be considered in later versions. For threshold_2, α and β are set to values of 0.4 and 0.6, respectively. This reveals more emphasis has been put on resource utilization influence. We decreased the value of ω to be 0.6 for threshold_2 as well, indicating a stricter utilization standard for additional speculation.

From the results we notice that both curves exhibit a similar trend: the threshold values increase in the begining due to tasks starting therefore raising utilization, followed by a relatively flat period as a result of progress adjustor’s influence. We observe a decreasing trend because tasks begin to complete and subsequently release resources. When the job is approaching its completion, at which time the probability of a replica outperforming the straggler is low, the threshold

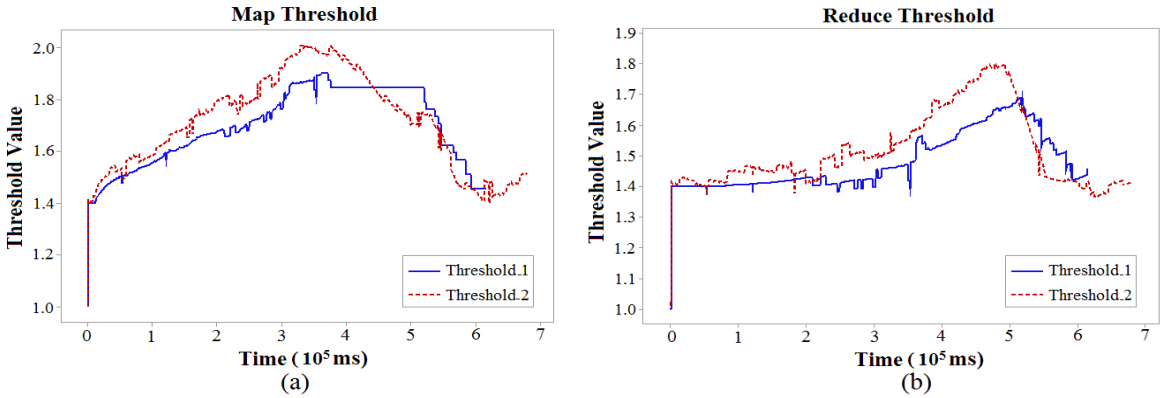

Fig. 10: Parameter configurations (a) $\alpha = 0.5$, $\beta = 0.5$, $\mu = 0.5$, $\omega = 0.7$; (b) $\alpha = 0.4$, $\beta = 0.6$, $\mu = 0.5$, $\omega = 0.6$.

values increase again to avoid needless speculation. Despite the similarity, we also see that the turning point from value increase to decrease for threshold_2 is slightly earlier than threshold_1. This is because the utilization fall caused by task completion generates a larger effect than late phase increase. The highest threshold value for threshold_2 is greater than threshold_1 due to it being more sensitive to utilization.

6.3. Simulation

We conducted a simulation in order to evaluate the advantages of the dynamic threshold algorithm within a larger-scale system. We use SEED - an event-based simulator [Garraghan et al. 2016a] that can simulate Cloud datacenter operations such as the creation of jobs (comprising multiple tasks) onto a set of machines for execution. The design of the simulation also adopts the following five assumptions:

(1) All speculative replicas created are allocated with identical CPU and memory requirements with the straggler task, and start re-execution from the beginning instead of continuing from the detected point [Zaharia et al. 2008];
(2) The scheduler will create replicas immediately after a task has been defined as a straggler, and will schedule the replica as a normal task;
(3) The maximum resource capacity of the cluster remains the same (i.e. no addition/removal of servers) during threshold calculation;
(4) Replicas can potentially become stragglers as well [Ananthanarayanan et al. 2010];
(5) Once a certain level is exceeded, higher resource utilization within a system leads to a higher probability of straggler occurrence [Ouyang et al. 2016b].

When simulating the task progress model, we adopted a simple probabilistic function. We assume the probability of straggler occurrence follows the role given certain system utilization by:

$$P(_{Straggler}) = \begin{cases} 0.1 & \text{if utilization} \in (0,0.6] \\ 0.2 & \text{if utilization} \in (0.6,0.8] \\ 0.3 & \text{if utilization} \in (0.8,0.9] \\ 0.4 & \text{if utilization} \in (0.9,1) \end{cases} \quad (10)$$

Tasks follow a linear progress function with the above straggler probability. For stragglers, the duration will be slowed stochastically by a factor between 120% to 250% compared to the average task duration. This is consistent with the statistics discovered in [Ananthanarayanan et al. 2010]. Other sophisticated progress functions (straggler probability function and straggler tailing duration function) such as the Pareto and the Zipf distributions [Calzarossa et al. 2016] can easily be implemented to replace the linear progress.

We constructed a simulated cluster with 100 servers and 500 tasks, and another with 800 servers and 10,000 tasks. The nodes in the simulations are configured with 4096MB memory capacity, and tasks for the former with 512MB memory requirement while 256MB for the letter. For the dynamic threshold calculation algorithm, we adopted equal weightings to progress and resource adjustor, and we assign 0.5 to both standard parameters. The detailed results are shown in table VIII.

Table VIII: Simulation results for different thresholds.

Threshold Method	Number of Servers	Number of Tasks	Response Time (step)	Replica Number	Successful Speculation	Straggler Percentage	Speculation Effectiveness
Dynamic	100	500	130	72	48	14.4%	66.67%
Static (1.5)	100	500	163	59	18	11.8%	30.5%
Dynamic	800	10,000	162	1,861	1,443	18.61%	77.54%
Static (1.5)	800	10,000	213	1,486	702	14.86%	47.24%

From the results we observe that, at the cost of an additional 2.6% replica numbers, the dynamic threshold can reduce job execution by 20.25%, and among all replicas lauched, 66.67% catch up with the corresponding straggler while only 30.5% effective replicas for static threshold. In the case of 10,000 tasks, the statistics follow the same trend: job execution time has been reduced by 23.94% with 3.75% more replicas when adopting dynamic straggler threshold, with a 30.3% improvement in speculation effectiveness.

7. RELATED WORK

There exist numerous risks for Internetware applications due to the open and dynamic environment they are situated in [Lü et al. 2013]. For example, risks of context inconsistency (environmental information that impacts computation) are discussed in [Xu et al. 2013a]. Environmental contexts encompass time, which stragglers can directly undermine. In addition, a middleware layer is required for Internetware applications to successfully fulfill non-functional requirements including dependability and QoS [Ye et al. 2010]. Since stragglers lead to late-timing failures and potential QoS violation for latency-sensitive services, straggler tolerance should be supported by such middleware platforms.

Current literature predominantly focuses on proposing straggler mitigating techniques, which can be divided into avoidance and tolerance. Avoidance typically occurs within the task scheduling phase [Yadwadkar and Wontae 2012; Xu et al. 2013b]. For example, a MapReduce scheduler will typically assign map tasks to a node that stores the input data in order to reduce unnecessary network transmission [Dean and Ghemawat 2008]. The scheduler may also attempt to avoid scheduling tasks onto known faulty nodes by adopting blacklist techniques [Zhang et al. 2014]. However, such techniques may be insufficient when stragglers are not restricted to a small set of machines [Ananthanarayanan et al. 2013]. As a result, straggler tolerance, which is typically performed at application run-time, becomes the most commonly applied mitigating method, and speculative execution [Dean and Ghemawat 2008] is the most established approach of this kind.

Although various improved versions of speculated execution have been proposed [Dean and Ghemawat 2008; Chen et al. 2014], the basic foundation of task replication remains the same. The scheduler observes the progress of each individual task within the same job. Once a straggler has been identified, the system automatically creates a replica that performs identical work without killing the original task, and uses whichever result completes first. Once a task finishes (either the original straggler or the newly created backup), the scheduler discards the other unfinished one and releases the computing resources. Speculative execution is commonly deployed in production clusters such as Facebook, Google, Bing, and Yahoo [Dean and Ghemawat 2008].

The straggler threshold plays an important role in the straggler detection process, measuring to what extent a slow task should be defined as a straggler. Currently, there exist three categories of thresholds. The Hadoop V1 (version one) default scheduler [Dean and Ghemawat 2008] adopts a PS based threshold, which monitors the PS of each task and identifies the tasks with 20% less

PS compared with average as stragglers. This type of threshold has an unavoidable limitation where tasks that have completed more than 80% progress can never be speculatively executed.

To avoid above problem, LATE [Zaharia et al. 2008] and Mantri [Ananthanarayanan et al. 2010] determine stragglers using a threshold based on the task's estimated completion time (ECT). This type of threshold focuses on the actual remaining time and performs better in improving final response. LATE achieved an improvement by a factor of two compared with Hadoop, and Mantri get a further 32%.

Dolly [Ananthanarayanan et al. 2013] adopts a progress rate (PR) based threshold in their straggler-tolerant system. PR is a metric used to measure the task processing speed, and is calculated by dividing the progress score with the corresponding elapsed time. Dolly classifies a task as a straggler if its PR is less than 50% of the average PR compared to its siblings. This type of threshold comes with its own limitations due to the ignorance of the changing nature of PR and task progress phases. Taking the following scenario as an example, if task A is three times slower in PR than the average yet has a PS of 0.9, while task B is two times slower but is only at 10% of its execution lifecycle, a PR based threshold would detect task A as a straggler due to its slower progress rate than B. However, in reality, it is task B that will significantly impede total job completion time. Based on above reasoning, within this paper, the ECT based threshold is the primary type that we focus on enhancing rather than purely PR.

No matter which type, current work in [Dean and Ghemawat 2008] [Zaharia et al. 2008] [Ananthanarayanan et al. 2013] [Ananthanarayanan et al. 2010] all specify the straggler threshold as a pre-defined value, typically 50% greater than average task execution. However, such a static threshold can debilitate the effectiveness of speculative replica generation. Specifically, it fails to consider the intrinsic diversity of job timing constraints within modern day systems. Furthermore, a static threshold value ignores the resource cost of launching a speculative replica and its negative impact toward the system when the utilization is already high [Xu and Lau 2013], which potentially increases straggler probability due to contention.

Dynamic Replication is a way to solve this problem, and has been widely adopted. Work in [Rabinovich et al. 1999] proposes a dynamic replication algorithm for Internet objects in order to improve requests response. This method attempts to place replicas in the vicinity of a majority of requests while ensuring no servers are overloaded. Work in [Wang et al. 2011] proposes a dynamic slot allocation optimization method for Hadoop clusters to improve job execution, by modifying the pre-configuration of distinct map slots and reduce slots into dynamically assigned slots based on resource utilization in runtime to avoid contention or under-utilization. Work in [Sun et al. 2012] proposes a dynamic data replication algorithm in Cloud environments to improve system availability. This algorithm evaluates and identifies popular data, and triggers a replication operation when the popularity data passes a dynamic threshold. However, most of the current work on dynamic replication focuses on data instead of tasks.

An effective straggler threshold should have the ability to impose different levels of strictness for replica creation to coordinate with specified levels of QoS timing constraints. Our previous work [Ouyang et al. 2016a] proposed the concept of the dynamic straggler threshold; and its performance in reducing job response time is proved to be effective through simulation. In this paper, we further implement the algorithm into YARN platform with three different workloads, revealing a more practical result. Meanwhile, new evaluation metrics of successful speculation rate is discussed which focuses on algorithm efficiency, as well as a larger scale simulation evaluation.

8. CONCLUSION

Tolerating stragglers has become more difficult due to the increasing cluster scale and dynamic operational environments of modern Internetware applications. Current state-of-the-art methods attempt to improve application execution time using speculative execution, which creates replicas for identified straggling tasks. The threshold is a key concept used in defining to what extent a task can be identified as a straggler. In this paper, we propose a threshold calculation method in speculative execution mechanism to mitigate the straggler effects that dynamically adapt to different job types and system conditions to improve the efficiency of Internetware application executions. Our conclusions are as follows:

— *The dynamic threshold is effective in improving job completion times and reducing late timing failures*. While current methods identify stragglers using a static threshold defined as 50% greater than average execution, our approach allows for an adaptive threshold calculation that automatically captures job QoS timing requirements, system resource utilization level, and task progress. Experiment results demonstrate that the dynamic technique can improve job completion by a factor up to 20% compared to static methods, while simulation results indicate the same trend, achieved an improvement up to 23.94% in a large scale environment.
— *Replica number trade-offs for different levels of resource utilizations to cope with the dynamic operational environments*. Improving job execution by speculation and saving resources can be a conflict of interest and require trade-off balancing. Experiments are conducted to compare the dynamic approach against the current static approach under different operational conditions; results demonstrate that our approach creates fewer replicas under high utilization. While under low resource utilizations, the dynamic threshold method proactively generates more replicas to achieve a quicker response time.
— *Enhanced speculation success rate and effective quality assurance*. Not all replicas generated can successfully outpace the identified stragglers; increasing this percentage plays a key role in speculation efficiency. The dynamic threshold will chose the right timing and suitable environment to launch replicas, therefore achieving a higher speculation success rate compared to the static method. Results from experiments and simulations show the largest improvement of 50% (from 16.67% to 66.67%) and 36.17% (from 30.5% to 66.67%), respectively.

Although the dynamic threshold has a better performance compared to the static threshold, we also noticed that both methods have space for improvement, especially towards the speculation success rate. The speculation framework itself meets a bottleneck when stragglers are caused by data skew or inner logic design reasons rather than resource contention or outer environmental impacts. This leaves room for the future improvement of reason-aware straggler mitigating methods. We can then integrate our dynamic threshold approach into such techniques to discover whether substantial gains in job completion can be achieved. Furthermore, there is also an opportunity to extend our approach by exploring other factors through designing a cost function beyond CPU and memory utilization, including disk volume and network speed.

REFERENCES

Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *the 10th USENIX Symposium on Networked Systems Design and Implementation*. (2013), 185–198.

Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI*, Vol. 10. 10 (2010), 24–37.

Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing* 1, 1 (2004), 11–33.

GE Blelloch, L Dagum, SJ Smith, K Thearling, and M Zagha. An evaluation of sorting as a supercomputer benchmark. *International Journal of High Speed Computing* (1993).

Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona 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.

Maria Carla Calzarossa, Luisa Massari, and Daniele Tessera. Workload characterization: a survey revisited. *ACM Computing Surveys (CSUR)* 48, 3 (2016), 48.

Qi Chen, Cheng Liu, and Zhen Xiao. Improving mapreduce performance using smart speculative execution strategy. *IEEE Trans. Comput.* 63, 4 (2014), 954–967.

Quan Chen, Daqiang Zhang, Minyi Guo, Qianni Deng, and Song Guo. Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment. In *Computer and Information Technology (CIT), IEEE 10th International Conference on*. (2010), 2736–2743.

Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

Marisol García-Valls, Tommaso Cucinotta, and Chenyang Lu. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture* 60, 9 (2014), 726–740.

Peter Garraghan, David McKee, Xue Ouyang, David Webster, and Jie Xu. SEED: A Scalable Approach for Cyber-Physical System Simulation. *IEEE Transactions on Services Computing* 9, 2 (2016), 199–212.

Peter Garraghan, Xue Ouyang, Renyu Yang, David McKee, and Jie Xu. Straggler Root-Cause and Impact Analysis for Massive-scale Virtualized Cloud Datacenters. *IEEE Transactions on Services Computing* (2016).

Hadoop. (2016). http://hadoop.apache.org/

Umesh Kumar and Jitendar Kumar. A comprehensive review of straggler handling algorithms for mapreduce framework. *International Journal of Grid and Distributed Computing* 7, 4 (2014), 139–148.

YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, (2012), 25–36.

Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, (2014), 1–14.

Jian Lü, Yu Huang, Chang Xu, and Xiaoxing Ma. Managing environment and adaptation risks for the internetware paradigm. In *Theories of Programming and Formal Methods*. Springer, (2013), 271–284.

Hong Mei. Internetware: Challenges and future direction of software paradigm for Internet as a computer. In *Computer Software and Applications Conference (COMPSAC), IEEE 34th Annual*. (2010), 14–16.

Hong Mei, Gang Huang, and Tao Xie. Internetware: A software paradigm for internet computing. *Computer* 45, 6 (2012), 26–31.

Hong Mei and Xuan-Zhe Liu. Internetware: An emerging software paradigm for Internet computing. *Journal of computer science and technology* 26, 4 (2011), 588–599.

OpenCloud. (2016). http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html

OpenNebula. (2016). http://opennebula.org/

Xue Ouyang, Peter Garraghan, David McKee, Paul Townend, and Jie Xu. Straggler Detection in Parallel Computing Systems through Dynamic Threshold Calculation. In *IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*. (2016), 414–421.

Xue Ouyang, Peter Garraghan, Renyu Yang, Paul Townend, and Jie 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).

Pankesh Patel, Ajith H Ranabahu, and Amit P Sheth. Service level agreement in cloud computing. (2009).

Michael Rabinovich, Irina Rabinovich, Rajmohan Rajaraman, and Amit Aggarwal. A dynamic object replication and migration protocol for an internet hosting service. In *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*. (1999), 101–113.

Charles Reiss and John Wilkes. Google cluster-usage traces: format+ schema. *Google Inc., White paper* (2011), 1–14.

Josh Rosen. Fine-grained micro-tasks for MapReduce skew-handling. *White paper, University of Berkeley* (2012).

Dawei Sun, Guiran Chang, and Xingwei Wang. Modeling a dynamic data replication strategy to increase system availability in cloud computing environments. *Journal of computer science and technology* 27, 2 (2012), 256–272.

Google Cluster Data V2. (2016). https://github.com/google/cluster-data

Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, and others. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, (2013), 5–20.

Kun Wang, Ben Tan, Juwei Shi, and Bo Yang. Automatic task slots assignment in hadoop mapreduce. In *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*. ACM, (2011), 24–29.

Chang Xu, YePang Liu, Shing Chi Cheung, Chun Cao, and Jian Lv. Towards context consistency by concurrent checking for Internetware applications. *Science China Information Sciences* 56, 8 (2013), 1–20.

Huanle Xu and Wing Cheong Lau. Resource optimization for speculative execution in a MapReduce cluster. In *21st IEEE International Conference on Network Protocols (ICNP)*. IEEE, (2013), 1–3.

Jianlong Xu, Zibin Zheng, and Michael R Lyu. Web Service Personalized Quality of Service Prediction via Reputation-Based Matrix Factorization. *IEEE Transactions on Reliability* 65, 1 (2016), 28–37.

Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation*. (2013), 329–341.

Yadwadkar and Wontae. Proactive straggler avoidance using machine learning. *White paper, University of Berkeley* (2012).

Chunyang Ye, Jun Wei, Hua Zhong, and Tao Huang. Middleware support for internetware: a service perspective. In *Proceedings of the Second Asia-Pacific Symposium on Internetware*. ACM, (2010), 4.

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. *HotCloud* 10 (2010), 10–16.

Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, Vol. 8. 8, 4 (2008), 7–20.

Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1393–1404.

Received September 2016; revised March 2017; accepted May 2017