Straggler Detection in Parallel Computing Systems through Dynamic Threshold Calculation

Xue Ouyang, Peter Garraghan, David Mckee, Paul Townend, Jie Xu
School of Computing
University of Leeds
Leeds, UK
{scxo, p.m.garraghan, scdwm, p.m.townend, j.xu} @ leeds.ac.uk

Abstract — **Cloud computing systems face the substantial challenge of the Long Tail problem: a small subset of straggling tasks significantly impede parallel jobs completion. This behavior results in longer service response times and degraded system utilization. Speculative execution, which create task replicas at runtime, is a typical method deployed in large-scale distributed systems to tolerate stragglers. This approach defines stragglers by specifying a static threshold value, which calculates the temporal difference between an individual task and the average task progression for a job. However, specifying static threshold debilitates speculation effectiveness as it fails to consider the intrinsic diversity of job timing constraints within modern day Cloud computing systems. Capturing such heterogeneity enables the ability to impose different levels of strictness for replica creation while achieving specified levels of QoS for different application types. Furthermore, a static threshold also fails to consider system environmental constraints in terms of replication overheads and optimal system resource usage. In this paper we present an algorithm for dynamically calculating a threshold value to identify task stragglers, considering key parameters including job QoS timing constraints, task execution characteristics, and optimal system resource utilization. We study and demonstrate the effectiveness of our algorithm through simulating a number of different operational scenarios based on real production cluster data against state-of-the-art solutions. Results demonstrate that our approach is capable of creating 58.62% less replicas under high resource utilization while reducing response time up to 17.86% for idle periods compared to a static threshold.**

***Keywords-* Long Tail Problem; Stragglers; Speculative Execution; Service QoS; Resource Utilization**

I. Introduction

Modern day distributed systems are composed of thousands of heterogeneous servers in order to provide computing services globally. With the rapid growth of data volume and exploitation, parallel computing has become an increasingly common technique for running applications effectively within clusters. Technologies such as MapReduce and Spark achieve this by decomposing a job into multiple tasks which perform a subset of computation and data processing, significantly speeding up job completion time.

In such parallel processing systems, task characteristics are driven by diverse consumer requirements, enforced through a Service Level Agreement (SLA) [2] detailing the level of acceptable service. One element provisioned and enforced by the SLA is the Quality of Service (QoS) that may be composed by numerous parameters including performance, real-time and security constraints of the service. Parameters of interest are dependent on business objectives; for example, soft real-time applications typically emphasize a boundary on acceptable response time, with violations resulting in timing failures [3]. However, given the increasing scale and complexity, there have arisen challenges in tolerating emergent system phenomena that significantly impacts the fulfillment of service QoS [4]. One such challenge is the Long Tail problem, defined as a small proportion of task stragglers experience abnormally long execution in comparison to other sibling tasks, thus incur significant delays to job completion as well as decreased system availability due to committed computing resources wasted on waiting tasks [5]. It has been identified that stragglers are caused by several factors including contention of shared resources, node disk failures, and imbalanced task workloads [17].

Stragglers are detected within systems by measuring or predicting when an individual task completion time is proportionally greater than the average task execution duration within a job, and is expressed as a *threshold*. In order to tolerate the impact of Long Tail problem, methods such as speculative execution creates replicas of task stragglers to shorten job completion by using whichever result completes first. Current research and industrial practice adopts this threshold as a pre-defined value of task execution 50% larger than the average task execution duration [5][11-13]. However, this static approach comes with a significant limitation: as the threshold value does not reflect optimal straggler detection and mitigation strategies in accordance to job diversity and system operation. Capturing these two characteristics allows for stricter or more relaxed time thresholds for straggler detection and task replication while adhering to job QoS. For example, if the system is exhibiting high utilization, the overhead brought by additional replicas will further burden the system, leading to increased straggler occurrence. In contrast, low system utilization allows for more leniencies towards replica generation to improve parallel job completion, and benefit those jobs that emphasize timing constraints for successful execution.

This paper proposes an algorithm that enables dynamic threshold calculation for straggler tolerance in distributed systems to augment state-of-the-art Long Tail identification and mitigation techniques. Specifically, our approach factors service QoS (specifically timing constraints), task execution progress, and the cluster resource usage to calculate the optimal time threshold for defining straggler tasks for replication in parallel jobs. Our approach is validated through simulation of a Cloud datacenter under diverse consumer and operational scenarios, which uses empirical findings to represent real system behavior. Results show that our technique can further improve job completion and reduce timing failure probabilities, as well as reduce replica generation under high resource utilization environment.

The paper is structured as follows: Section 2 presents the background; Section 3 surveys the related work; Section 4 analyzes the problem formulation; Section 5 details the algorithm design; Section 6 presents the experiment set up as well as the evaluation results. Finally, Section 7 discusses conclusions and future work.

II. BACKGROUD

A. *Long Tail Behavior in Distributed Parallel Systems*

Service response time is an important factor in pay-by-the-hour environments such as Amazon EC2, and systems which require rapid response to users. However, such a requirement becomes increasingly challenging in large-scale systems due to Long Tail. In distributed systems, the Long Tail problem is defined as a phenomena that occurs when a distributed job - composed of multiple smaller tasks executing in parallel - incurs significant delays in completion due to a small subset of impeded parallelized tasks [5]. These delayed tasks that perform much slower compared to their sibling tasks are defined as *stragglers*.

To demonstrate such a problem, we have characterized two parallel jobs from the operational tracelog of a Google cluster [6, 7] composed of over 12,500 servers applying the filtering technique described in [8]. Figure 1(a) shows an example of typical job completion, where it can be observed that most tasks complete at approximately 100s with the longest task execution being merely 30% greater compared to the average job completion. On the other hand, Figure 1(b) depicts task completion of a job from Google cluster that exhibits Long Tail phenomena, characterized by a pronounced tailing shape with the longest task taking approximately 300% longer compared to average task completion. There are a number of related works that study the influence of the Long Tail problem. For example, [8] demonstrates that even rare performance abnormalities can affect a significant portion of all requests in distributed systems after analyzing two production system logs, while [9] further shows how Long Tail is an increasingly common phenomenon in the face of increased growth of system scale.

There are two ways to address the Long Tail problem: avoidance and tolerance. Avoidance typically occurs within the task scheduling phase. 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 overhead [1]. The scheduler may also attempt to avoid scheduling tasks onto known faulty nodes by adopting blacklist techniques [16]. However, blacklisting may be insufficient when stragglers are not restricted to a small set of machines [11]. As a result, straggler tolerance, which is typically performed at application run-time, is the most commonly applied method for speculative execution.

B. *Speculative Execution*

First proposed by Dean [1], speculative execution observes the progress of each individual task within the same job. Once a task straggler has been identified, the system automatically creates a replica (or a backup copy that performs identical work) without killing the original task, and uses whichever result that completes first. Once a task finishes (either the original straggler or the newly created replica), the scheduler discards the other unfinished task and releases the computing resources back to use. This method is commonly deployed in many production clusters such as Facebook, Google, Bing, and Yahoo.

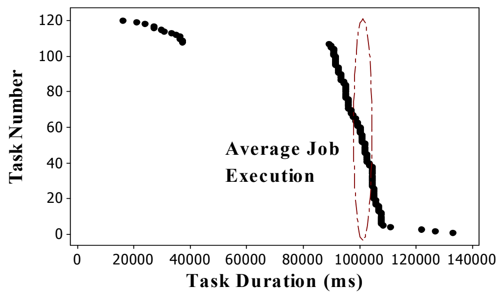
In Hadoop's default speculative mechanism, stragglers are identified by monitoring the matrix of ProgressScore (ranging from 0 to 1) defined in equation (1) to measure the execution progress of a task.

$$PS[i] = \begin{cases} M/N & \textit{For map task} \\ 1/3 * (K + M/N) & \textit{For reduce task} \end{cases} \qquad (1)$$

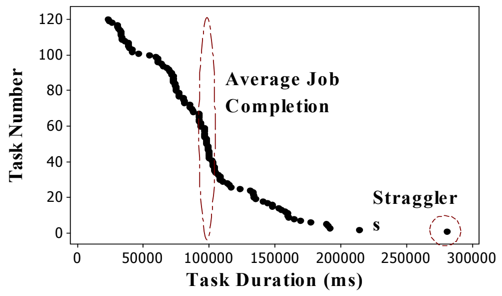
where $PS[i]$ represents the progress score of the i^{th} task. The number of key/value pairs need to be processed in a task is denoted by N, while M stands for the number of key/value pairs that have already been processed in a specific task. The finished phases for a reduce task is represented by K. For a Map, the progress score is the fraction of input data read, while for a Reduce the execution is divided into three phases (copy, sort and reduce) with each account to one third of the final progress score. Such weighting can be modified through changing scheduler settings.

III. RELATED WORK

In order to address the Long Tail problem influence in distributed parallel systems, a straggler-tolerate system is proposed composed by three main components shown in Fig.2: straggler identifier, speculation executor and decentralized agents responsible for recording task progress and reporting. The threshold calculation is a critical component of the straggler identifier in such systems. Threshold is used to identify when a task is



(a)



(b)

Figure 1. Task completion time for jobs exhibiting (a) Typical behavior, (b) Long Tail phenomena.

defined as a straggler, and is expressed as a ratio calculating the time difference between an individual task and the average progresses of all tasks within a job (i.e. If the average task completion is speculated to be 100s, a threshold value of 200% would indicate any tasks that require 200s+ to complete are identified as stragglers).

The direct impact of changing the threshold value is the number of stragglers identified, followed with the resource spent on creating their replicas. Figure 3 shows how the proportion of tailing jobs (ranging from 0-1) within two production Cloud datacenters analyzed in [8] are affected by different time threshold settings ranging from 130% to 200%. Currently, there exist three categories of threshold:

Progress score based threshold identifies a straggler and subsequently launches a speculative replica based on progress score of task execution as shown in equation (2) and (3).

$$PS_{avg} = \sum_{i=1}^{n} PS[i]/n \qquad (2)$$

For task $$Task_i: PS[i] < PS_{avg} * 80\% \qquad (3)$$

where PS_{avg} is the average progress score of a job and n represents the number of tasks that are being executed. This threshold calculation is adopted by the Hadoop default scheduler [1] configured with a default threshold value of 80%. This threshold has an unavoidable limitation where tasks that have completed more than 80% progress can never be speculatively executed.

Progress rate based threshold. Progress rate is a metric used to measure the task progress rate (PR), and is calculated by equation 4:

$$PR[i] = PS[i]/T \qquad (4)$$

where T is the amount of time $Task_i$ has been

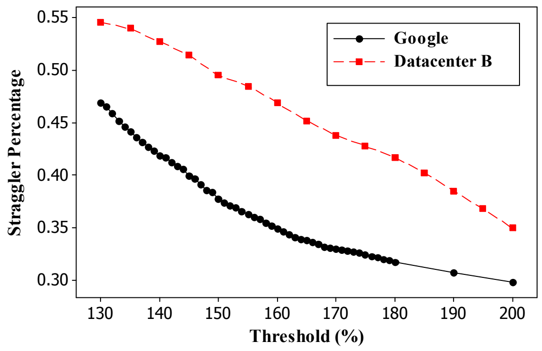


Figure 3. Relation between time threshold setting and tailing jobs.

executing. Dolly [11] adopts this type of threshold in their straggler-tolerant system, and classifies a task as a straggler if its progress rate is less than 50% of the average progress rate compared to its siblings.

Although this type of threshold addresses the limitations of progress score based methods, it still comes with its own limitations. Taking the following scenario as an example, if task A is three times slower than the average task execution yet has a progress score of 0.9, while task B is two times slower but is only at 10% of its execution lifecycle, a progress rate based threshold would detect task A as a straggler due to its slower progress than B. However, in reality, it is task B that will significantly impede total job completion time.

Estimated finish time based threshold (or time threshold) calculates the estimated time to completion given by LATE [5].

$$TTC[i] = (1 - PS[i])/PR[i] \qquad (5)$$

If a task's estimated finish time is longer than a certain percentage compared to the average value within the same job, it will be flagged as a straggler. The current state-of-the-art predominantly use this time threshold [5][11-13]. By focusing on the estimated time remaining rather than

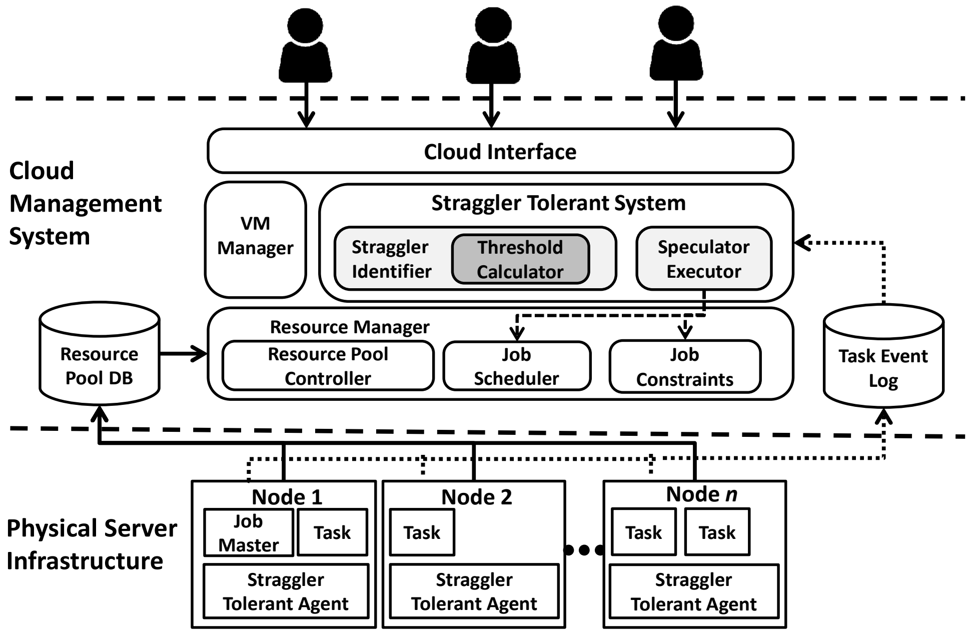


Figure 2. Cloud computing model with integrated straggler-tolerant system

progress rate, this type of threshold only speculatively executes task replicas that will improve job response time. Within this paper, time threshold is the primary type of threshold that we focus on enhancing.

Work in [5][11][12][13] specify the value of time 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. An effective time threshold calculation method should have the ability to impose different levels of strictness for replica creation to coordinate with specified levels of QoS. Furthermore, a static time threshold value ignores the resource cost of launching a speculative replica and its negative impact within the system.

IV. Problem Formulation

While the state-of-the-art defines the time threshold as a static value measured by the time proportion between an individual task and average job execution duration, a more refined strategy is to capture intrinsic system operational conditions and job diversity. To achieve this, it is important to determine key factors that can be leveraged in order to improving job performance and current system resource usage. The key three factors identified are as follows.

QoS timing constraints: This parameter encompasses acceptable levels of QoS response time. Jobs which fail to complete prior to the specified deadline result in timing failures and degraded application performance. Applications within modern day distributed systems typically pursue different objectives and express different levels of acceptable QoS. Such a characteristic allows for different degrees of strictness for generating replicas when attempting to tolerate the impact of task stragglers. Therefore, it is intuitive to consider the QoS response time as an important factor when calculating the time threshold.

Task Lifecycle Progress: It is important to consider the current execution progress completed for a task for effective replica generation. Specifically, a replica should ideally be spawned when it is likely to complete prior to the task straggler, otherwise the replica will result in unnecessary resource consumption with no improvement towards job completion time. For example, when a task experiences slow down at its late phase, it still has a probability to complete prior to its newly generated replica. As a result, it is reasonable to increment the time threshold slightly in response to this late task progress to avoid needless speculation for replica creation. On the other hand, if a straggler is identified at an early phase, the replica should have a higher probability to outpace the original task. This allows the system to lower the threshold value at this point within the task lifecycle to encourage replica generation. Based on above reasoning, it is important to consider current task execution phase for launching speculative replicas.

System Resource Usage: An important consideration for straggler tolerant systems is the impact of replica generation on resource consumption under different operational conditions. Creating replicas at high resource utilization poses a greater threat to system stability and can further increase the likelihood of straggler occurrence, while low system utilization allows for additional replica generation. Furthermore, replicas themselves can potentially become task stragglers too. Observations proposed in [10] state that 3% of replicas still take ten times longer than normal task execution in Bing's production cluster. Since the speculative replicas will execute with data identical to the original task straggler and configured with same resource requests, the expense of tasks should also be considered. If the resource requirement of the original task is high, then generated replicas can result in a high resource cost with no substantial improvement towards overall job completion time. Based on this reasoning, the time threshold calculation should consider current levels of system resource utilization.

V. Algorithm Design

Our proposed algorithm defines a dynamic time threshold which indicates when a task replica should be created for tolerating task stragglers by considering three key factors: QoS timing constraints, task progress and system resource usage. As described in previous sections, time threshold Th_j is a ratio that expresses the difference between the average progress of all tasks within a job against an individual task. In the occurrence of an individual task i exhibiting a time to completion (TTC) value greater than $Th_j \cdot TTC_{avg}$ at time t_j, the identification component (defined in Fig. 2) identifies the task as a straggler. As a result, a replica is created and will commence execution. Th_j is periodically updated at time interval t_j in order to allow the algorithm adapt to the current system environment. Equation 6 depicts the time threshold calculator at a high level:

$$Th_j = Q_j + \alpha * P_j + \beta * S_j \tag{6}$$

where Q denotes differences in task progress with respect to job QoS, P represents optimal replica creation based on task lifecycle, and S indicates the current resource utilization levels of the distributed system. Progress weight parameters α and utilization weight β can be specified by the system administrator to emphasize a particular factor for optimizing the trade-off for replica creation based on specific system operation goals. The sum of Q, P and S produce the threshold for detecting task stragglers. A greater value for Th_j indicates a stricter time threshold enforced at that particular time, resulting in fewer tasks being defined as stragglers, while a lower Th_j value allows a more relaxed condition for generating speculated replicas. These three parameters are respectively composed of lower levels of calculation to derive their values.

A. Calculation of the QoS influence

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, and prioritizes the rapidness of task execution over reduced resource usage for speculation. Q_j calculates the difference between a task's estimated completion time with respect to specified job QoS timing requirement at time interval t_j following equation (7).

$$Q_j = \begin{cases} \dfrac{QoS}{medium(TTC_i)} & if\ QoS \geq max\ (TTC_i) \\ \dfrac{min(TTC_i\ that\ larger\ than\ QoS)}{medium(TTC_i)} & else \end{cases} \quad (7)$$

where TTC_i represents the estimated time to completion for the i^{th} task within the cluster, and QoS stands for the request time requirement defined as a QoS parameter. As an example, assume that the estimated completion times at time interval t_j for five tasks in a job with QoS timing constraint of 300 milliseconds are 290ms, 290ms, 300ms, 380ms and 400ms. The minimal TTC_i larger than QoS under this case is 380ms, meaning that the value for Q_j is 127% (380 ÷ 300). For cases when all tasks are estimated to complete before the specified QoS, for instance, if we change QoS to 450ms instead of 300ms in last example, then all TTC_i values will be smaller than QoS, meaning that the value for Q_j is 150% (450 ÷ 300), and no tasks will be flagged as stragglers.

It is worth highlighting that this algorithm will function for applications which do not contain an explicit QoS time request parameter specified. In such an event, a static time proportion value used in current literatures described in Section 3 is applicable, with the dynamic change for Th_j dependent on P_j and S_j.

B. Calculation of Progress Adjustor

The calculation of progress adjustor at time interval t_j is given as equation 8.

$$P_j = \frac{\sum_{i=1}^{n} PS[i]}{n} - \mu \quad (8)$$

where $PS[i]$ is defined in equation 1 with value bounded between 0 to 1, representing the start and the end of $task_i$ respectively. Progress standard parameter μ represents the specified maximum point within a task's lifecycle suitable for generating a replica. For example, assigning $\mu = 0.5$ represents that any task with a progress score smaller or equal to 0.5 will still be considered as early stage, and will lead to a smaller Th by generating a negative P_j value, increasing the likelihood of replica generation. While any progress score larger than 0.5 we be treated as late stage, resulting in higher threshold to limit replica generation.

C. Calculation of System Environment Adjustor

Parameter S_j affects the value of Th_j dependent on the system utilization at t_j. This calculation is given as

$$S_j = max\ (\frac{\sum_{i=0}^{n} CPUreq_i}{n} - \varphi, \frac{\sum_{i=0}^{n} MEMreq_i}{n} - \omega) \quad (9)$$

where n denotes the number of servers within the cluster system, while $CPUreq_i$ and $MEMreq_i$ represent the CPU and memory requirement of $task_i$ respectively and the sum represents the utilization of the entire system. If either one of these two parameters hits the optimal usage specified by admin (CPU standard threshold φ and memory standard threshold ω), it will increase the time threshold by generating a positive S_j value, resulting in stricter requirements for replica generation.

Algorithm 1 DynamicThresholdSetting

INPUTS:

- QoS - QoS timing constrains
- Phi - φ, CPU threshold
- Omega - ω, Memory threshold
- Mu - μ, Progress threshold
- Alpha, Beta - α, β, weightings

```
WHILE TaskRunningNumber > 0
    FOR EACH task i
        TTC[i] = (1 - PS[i]) / PR[i]  (Eqn.5)
    END FOR
    IF QoS != null
        SORT TTC
        Q = (TTC > QoS).First
        IF Q != null
            Q = Q / AVERAGE(TTC)
        ELSE
            Q = QoS / AVERAGE(TTC)
        END IF
    ELSE
        Q = 1.5
    END IF                          (Eqn.7)
    P = AVERAGE(PS) - mu            (Eqn.8)
    CPUdiff = AVERAGE(CPUreq) - phi
    MEMdiff = AVERAGE(MEMreq) - omega
    S = MAX(CPUdiff, MEMdiff)       (Eqn.9)
    Th = Q + alpha*P + beta*S       (Eqn.6)
    FOR EACH task i
        IF TTC[i] > ( Th*AVERAGE(TTC) )
            i.copy
            TaskEunningNumber ++
        END IF
    END FOR
END WHILE
```

To conclude, the pseudo code for the dynamic threshold setting algorithm is given in algorithm 1. At each time interval, the system will collect the progress score for every active task within the target job and calculate the estimated completion time for each task. The time threshold for time t_j is generated by calculating the QoS influence, progress adjustor and system environment adjustor (Q, P and S, respectively). This will identify the stragglers according to the calculated time threshold and create corresponding replicas. During the next time frame, the scheduler of such a straggler-tolerant system would launch these speculative tasks to tolerate the tailing effect.

VI. PERFORMANCE EVALUATION

In order to evaluate the performance of the dynamic time threshold calculation algorithm in straggler-tolerant systems, a series of simulation experiments have been conducted using distributed system simulator [14]. This section will introduce the simulation and experiment cases set up followed by the performance evaluation.

A. Simulation and Experiment Setup

The simulator SEED [14] can generate jobs with various numbers of tasks running in parallel and simulate Long Tail behavior by assigning task execution times follow the tailing distribution derived from real production data. In our experiment, we use the characteristics concluded in previous research such as [8, 10]. Empirical analysis of a production cluster from Microsoft Bing [10] claims that 80% of task stragglers have a uniform probability of being delayed by between 120% to 250% compared to the average task duration, while 10% of tasks are 10 times average duration. And [8] summarize that task level stragglers have an occurrence rate of 5%. User submission rate models and server configuration are determined from pattern derived from Google production cluster described in [15].

The simulator simulates task behavior based on following five assumptions: 1) All speculative replicas created by the straggler-tolerant system should be allocated with identical CPU and memory requirements, and starts execution from the beginning of the task's phase. 2) The scheduler will create replicas immediately after a task has been defined as a straggler, and will re-schedule the replica as a normal task. 3) The maximum resource capacity of the cluster does not change (i.e. addition/removal of servers) during threshold calculation. 4) Replicas themselves can potentially become task stragglers [10]; this is possible by assigning speculative replicas the same probability of becoming a straggler identical to normal tasks. 5) Higher resource utilization environment leads to a higher probability of straggler occurrence.

We altered system environment parameters, task attribute parameters and algorithm parameters to study their influence in terms of job response time and resource cost of replica generation. All together 17 experiment case pairs were conducted (each repeated five times) for static and dynamic threshold separately, each demonstrating the worst case, the best case and the average results. The value of 1.5 was chosen for static threshold to reflect widely accepted practice in current state-of-the-art.

First 5 case pairs control system environment related parameters, including number of servers, total required CPU and memory for the entire cluster. In our experiment, server number is changing from 20 to 100, while the required capacities are ranging from 135% to 27% (when this number exceeds 100%, which is the maximum available capacity the system could provide, tasks would wait till there are sufficient resource for them to run). The following 7 case pairs are targeting at the change of job attributes including task number per job, task CPU and memory requests. The last 5 case pairs focus on evaluating the sensitivity towards algorithm parameters by changing the weight parameter pair α and β as well as the standard parameter pair φ and ω.

For all experiment runs, the job QoS timing constraint is set to be 1000 milliseconds and the time interval is set to be every 100ms. The average task duration for simulator configuration is 700 milliseconds. When changing one factor of interest, the other two remain the same value. For system environmental factor, this default setting is 50 homogenous servers; for task attribute factor, default job will be set with 90 tasks and each of which require 0.3 CPU cores and memory ability; for the algorithm

TABLE 1. System environmental factor experiments results

Environment			**Threshold**	**Performance**					
Server Number	Total Required Memory (%)	Total Required CPU (%)	Calculation Method	Response Time (time interval 100ms)			Replica Number		
				Worst Case	Best Case	Average	Worst Case	Best Case	Average
20	135	135	Static(1.5)	1200	900	1080	7	5	5.8
20	135	135	Dynamic	1100	900	980	3	2	2.4
40	67.5	67.5	Static(1.5)	1200	1000	1120	6	5	5.4
40	67.5	67.5	Dynamic	1100	800	920	4	2	2.8
60	45	45	Static(1.5)	1100	800	900	4	3	3.4
60	45	45	Dynamic	1000	600	740	4	5	4.2
80	33.75	33.75	Static(1.5)	1000	700	820	4	3	3.4
80	33.75	33.75	Dynamic	900	700	740	5	4	4.6
100	27	27	Static(1.5)	1000	700	800	4	2	3
100	27	27	Dynamic	800	600	680	5	5	5

TABLE 2. Task attribute experiments results

Task			**Threshold**	**Performance**					
Task Number	CPU requirement	Memory requirement	Calculation Method	Response Time (time interval 100ms)			Replica Number		
				Worst Case	Best Case	Average	Worst Case	Best Case	Average
10	0.7	0.7	Static(1.5)	900	700	780	1	0	0.6
10	0.7	0.7	Dynamic	800	600	660	2	1	1.2
50	0.3	0.3	Static(1.5)	1000	700	820	2	1	1.4
50	0.3	0.3	Dynamic	800	700	740	3	2	2.4
50	0.5	0.5	Static(1.5)	1000	700	800	2	1	1.8
50	0.5	0.5	Dynamic	900	700	780	3	2	2.4
50	0.7	0.7	Static(1.5)	1100	800	940	2	2	2
50	0.7	0.7	Dynamic	1100	800	920	2	1	1.8
90	0.3	0.3	Static(1.5)	1100	900	980	4	3	3.8
90	0.3	0.3	Dynamic	1000	800	860	4	3	3.6
90	0.5	0.5	Static(1.5)	1100	900	1000	5	3	4.4
90	0.5	0.5	Dynamic	1100	800	880	4	2	3.4
90	0.7	0.7	Static(1.5)	1200	1000	1080	6	5	5.2
90	0.7	0.7	Dynamic	1100	900	1000	3	1	2.8

TABLE 3. Algorithm parameter experiments results

Algorithm				Threshold	Performance					
Progress Wight α	Utilization Weight β	CPU Standard φ	Memory Standard ω	Calculation Method	Response Time (time interval 100ms) Worst Case	Best Case	Average	Replica Number Worst Case	Best Case	Average
0.5	0.5	0.4	0.4	Static(1.5)	1100	900	960	5	3	4.4
0.5	0.5	0.4	0.4	Dynamic	1000	900	940	4	3	3.2
0.5	0.5	0.5	0.5	Static(1.5)	1100	900	980	4	3	3.8
0.5	0.5	0.5	0.5	Dynamic	1000	800	860	4	3	3.6
0.5	0.5	0.6	0.6	Static(1.5)	1000	800	900	5	3	4.2
0.5	0.5	0.6	0.6	Dynamic	1000	700	840	5	4	4.6
0.3	0.7	0.5	0.5	Static(1.5)	1100	900	980	4	3	3.8
0.3	0.7	0.5	0.5	Dynamic	1000	900	980	3	3	3
0.7	0.3	0.5	0.5	Static(1.5)	1000	900	940	4	3	3.8
0.7	0.3	0.5	0.5	Dynamic	900	900	900	4	3	3.6

parameters, 0.5 is the configured value for *α, β, φ* and *ω*.

B. Evaluation

We demonstrate the effectiveness of our dynamic time threshold calculation in straggler-tolerant systems in two aspects: 1) performance analysis and 2) parameter sensitivity analysis. The first part studies job timing improvement and resource saving improvement, using the job response time and replica number as the primary matrix. The second part studies algorithms effectiveness under different operational and environmental conditions. Our evaluation demonstrates the following:

The dynamic time threshold can improve job completion and decrease timing failure occurrence, as well as saving resource under high utilization scenarios by generating fewer replicas. As shown in Tables 1-3, the dynamic algorithm can reduce response time up to 17.86% on average across all experiment cases compared to a static threshold. The improvement is represented by the speedup of the job execution calculated as follows:

$$Improve = \frac{JobExecution_{Dynamic} - JobExecution_{Static}}{JobExecution_{Static}} \quad (10)$$

Fig.4 presents the changing trend of average response time of dynamic threshold and static ones respectively among different required system capacity (ranging from 14%, the first test case in Table 2 where a job only consists of 10 tasks that each require 0.3 CPU and memory capacity, to 135%, the first test case in Table 1 where the cluster has 20 servers). The required capacity exceeds 100% representing current available resource is less than required, hence tasks would wait till there is sufficient resource for execution. From this figure it is observable that dynamic threshold can improve job response time up to 17.86%, and reduce up to 10.58% timing constrain violations. This improvement is due to the algorithm altering the time threshold in accordance to meeting the specified job QoS.

The algorithm sensitivity is driven by resource utilization level, task progress phase and input variable settings. It is observable from the tables that the replica number created varies dependent on different resource utilizations. When resource utilization is high, the dynamic threshold will produce less task replicas, most notably for case "server number 20" in table 1 and the case "task number 90, requirements 0.7" in table 2. On the other hand, when the system is idle (i.e., low resource usage, resource required below the standard set by *φ* and *ω*), the dynamic method will proactively generate more replicas to pursue higher response time. Through Fig. 5 we see that the dynamic algorithm will decrease replica generation when there is an increase in task resource requirement, while the static method exhibits an opposite trend. This is due to the dynamic method considering the cost of creating speculations, as higher cost leads to stricter time threshold.

In terms of environmental factors, Fig. 6 demonstrates how the time threshold changes over the execution lifecycle of a job. Although different required resource capacity experience different variability in threshold value changes (for example in 6(a) and 6(b), the vibration is much shaper in order to control the number of replicas under high contention situations), there are still some similar trends: in the middle of the job's execution phase, all four experiment cases experience a decrease change, as during this time period tasks begin to complete and release resources, and when the job approaching its late phases, all dynamic values in Fig. 6 (a) to (d) exhibit a similar increase trend to avoid needless speculation when the probability of a replica to outperform the original straggler is low. In addition, when changing the input variables of weight parameters and standard parameters, the algorithm will generate different time threshold value for same conditions.

VII. CONCLUSION

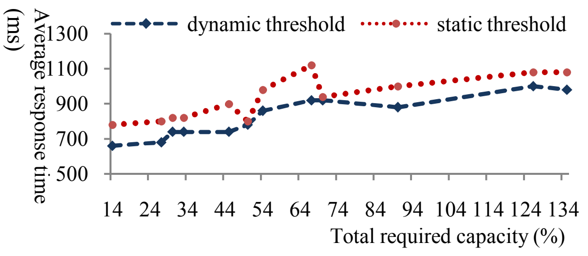


Figure 4. Average job response changing with total required capacity

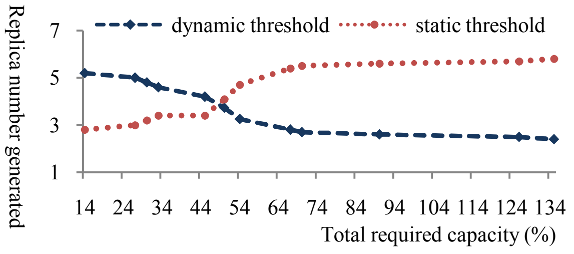


Figure 5. Average replica number changing with total required capacity

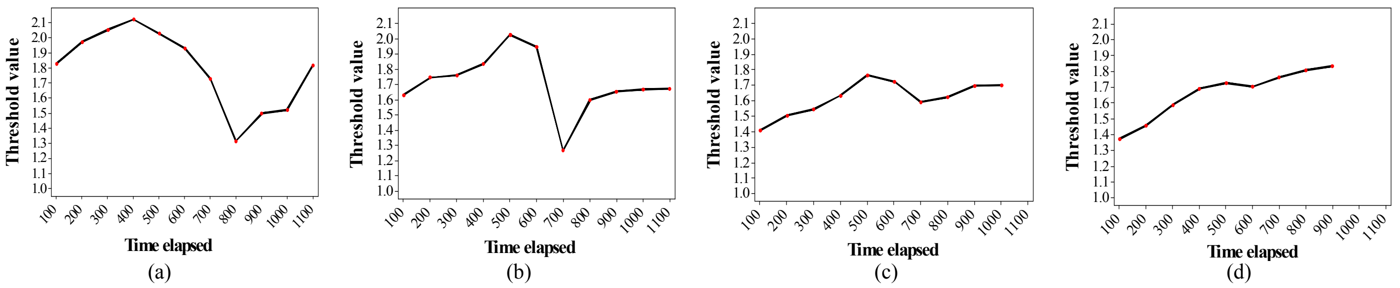


Figure 6. Threshold value change trend under cases with task number = 90, requirement = 0.3 for (a) 20 servers, (b) 40 servers, (c) 60 servicers, (d) 80 servers.

Service providers use SLAs to define the agreed terms of service they should meet whilst service users use it to ensure that the agreed QoS has been fulfilled. However, straggler tasks within a parallel job can significantly prolong service response since all the siblings need to wait for the outcome of the last task before the job finishes. The current state-of-the-art straggler mitigating method is speculative execution, which creates replicas for all identified stragglers. Threshold is the key concept used in defining to what extent a task can be identified as a straggler.

In this paper, we have developed a time threshold calculation method for straggler-tolerant systems that for the first time dynamically adapts to different job types and system conditions in order to improve job performance. Our conclusions are as follows:

- *A dynamic threshold calculation mechanism is an effective means to enhance straggle-tolerant systems that improves job completion*. While current methods identify task stragglers using static time threshold defined as 50% greater than average task execution, our approach allows for an automated dynamic time threshold calculation that captures job QoS, system resource usage level, and task progress. Our findings demonstrate that dynamic technique can further improve job completion by a factor up to 17.86% compared to static methods.
- *Replica number trade-offs exist among different level of resource utilization scenarios.* Improving job execution by speculation and saving resource can be a conflict of interest. Experiments conducted when comparing dynamic approach against current static approaches under different operational conditions demonstrate that our approach creates 58.62% less replicas while still adhering to QoS timing requirements under high utilization. In contrast, under low resource utilization cases, the dynamic threshold calculation method proactively generates more replicas to achieve a much quicker response time.

Future work includes the integration of our approach into other established Long Tail tolerance techniques besides basic speculative speculation to discover whether substantial gains in job completion timeliness and fulfillment of service QoS can be achieved. Furthermore, there is 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.

Acknowledgments

The work is supported in part by the National Basic Research Program of China (973) (No.2011CB302602), and the U.K. EPSRC WRG platform project (No. EP/F057644/1).

References

[1] Dean J, Ghemawat S. "MapReduce: simplified data processing on large clusters", Communications of the ACM, 51(1), 2008, pp. 107-113.

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

[3] Avižienis A, Laprie JC, Randell B, Landwehr C. "Basic concepts and taxonomy of dependable and secure computing", IEEE Transactions on Dependable and Secure Computing, 1(1), 2004, pp. 11-33.

[4] García-Valls M, Cucinotta T, Lu C. "Challenges in real-time virtualization and predictable cloud computing", Journal of Systems Architecture, 60(9), 2014, pp. 726-740.

[5] Zaharia M, Konwinski A, Joseph AD, Katz RH, Stoica I. "Improving MapReduce Performance in Heterogeneous Environments", In OSDI, 2008, vol. 8, no. 4, p. 7.

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

[7] Reiss C, Wilkes J, Hellerstein JL. "Google cluster-usage traces: format+ schema", Google Inc., Technical Report, 2011.

[8] Garraghan P, Ouyang X, Townend P, Xu J. "Timely Long Tail Identification Through Agent Based Monitoring and Analytics", In ISORC, 2015, pp. 19-26.

[9] Dean J, Barroso LA. "The tail at scale", Communications of the ACM, 56(2), 2013, pp. 74-80.

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

[11] Ananthanarayanan G, Ghodsi A, Shenker S, Stoica I. "Effective Straggler Tolerance: Attack of the Clones", In NSDI, 2013, Vol. 13, pp. 185-198.

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

[13] Rosen J, Zhao B. "Fine-Grained Micro-Tasks for MapReduce Skew-Handling", White Paper, University of Berkeley, 2012.

[14] Garraghan P, McKee D, Ouyang X, Webster D, Xu J. "SEED: A Scalable Approach for Cyber-Physical System Simulation", IEEE Transactions on Services Computing, 2015.

[15] Moreno IS, Garraghan P, Townend P, Xu J. "Analysis, modeling and simulation of workload patterns in a large-scale utility cloud", IEEE transaction on Cloud Computing, 2014, vol. 2, no. 2, pp. 208-221.

[16] Zhang Z, Li C, Tao Y, Yang R, Tang H, Xu J. "Fuxi: a Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale", Proceedings of the VLDB Endowment, 7(13), 2014, pp. 1393-1404.

[17] Kumar U, Kumar J. "A Comprehensive Review of Straggler Handling Algorithms for MapReduce Framework", International Journal of Grid and Distributed Computing, 7(4), 2014, pp. 139-148.