# Intelligent Straggler Mitigation in Massive-Scale Computing Systems

**by**

*Xue Ouyang*

**Submitted in accordance with the requirements for the degree of Doctor of Philosophy.**

UNIVERSITY OF LEEDS

**The University of Leeds**
**School of Computing**

**March 2018**

**The candidate confirms that the work submitted is her own, and the appropriate credit has been given where reference has been made to the work of others.**

# Intellectual Property and Publication Statements

The candidate confirms that the work submitted is her own, and the works formed part of jointly authored publications have been included. The contribution of the candidate and the other authors to this work has been explicitly indicated. The candidate confirms that, the appropriate credit has been given within the thesis where reference has been made to the work of others. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgment.

The jointly authored publications are organized according to the main contents. The papers that provide the basis for Chapter 3, the straggler quantitative analysis, are:

- *P. Garraghan, X. Ouyang, R. Yang, D. McKee, J. Xu, "Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters", IEEE Transactions on Services Computing, 2016.*

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

- *P. Garraghan, X. Ouyang, P. Townend, J. Xu, "Timely long tail identification through agent based monitoring and analytics" in the proceedings of the 18th IEEE International Symposium on Real-Time Distributed Computing (ISORC), pp. 19-26, 2015.*

In the ISORC paper, the candidate was responsible for data analytics. This includes the straggler problem severance analysis using the Google and Ali datasets, revealing the core observation that, limited task stragglers can lead to significant consequence on total job completion. P. Garraghan was responsible for conducting the hive query tests on the mini cluster built for experiments. P. Townend and J. Xu offered help in reviewing and improving the paper writing. For the DSN fast abstract, the candidate was responsible for conducting the correlation analysis towards the straggler occurrence and the system conditions. R. Yang helped in collecting and providing the data. P. Garraghan, P. Townend, and J. Xu offered help in editing the paper. The TSC paper is a journal extension mainly based on the ISORC paper, but combining the DSN observations. Therefore the contents that directly attributable to the candidate and to the other authors are similar to the two previous conference papers.

The papers that provide the basis for Chapter 4, the adaptive straggler threshold, are:

- *X. Ouyang, P. Garraghan, B. Primas, D. McKee, P. Townend, J. Xu, "Adaptive Speculation for Efficient Internetware Application Execution in Clouds", ACM Transactions on Internet Technology, 18(2): 15, 2018.*

- *X. Ouyang, P. Garraghan, D. McKee, P. Townend, J. Xu, "Straggler detection in parallel computing systems through dynamic threshold calculation" in the proceedings of the 30th IEEE International Conference on Advanced Information Networking and Applications (AINA), pp. 414-421, 2016.*

- *P. Garraghan, D. McKee, X. Ouyang, D.Webster, J. Xu, "Seed: A scalable approach for cyber-physical system simulation", IEEE Transactions on Services Computing, 9(2): 199-212, 2016.*

In the AINA paper, an adaptive straggler threshold algorithm is proposed in order to properly evaluate the most suitable task stragglers for mitigation under different system conditions. The candidate proposes the method design. P. Garraghan joined the discussion in order to improve the algorithm design and to chose the suitable parameter settings. D. McKee helped setting up the simulation using SEED. P. Townend and J. Xu offered help in reviewing and revising the paper. The ToIT paper is the journal extension based on the AINA one. The major extension is in respect of implementation and evaluation. I implemented the adaptive threshold algorithm into the YARN architecture for experiment evaluation, B. Primas helped with the theoretical example chapter. The role for other authors is similar to the previous conference paper. As for the TSC paper, the content is not directly linked to the straggler problem, instead, it is an introduction to the self-developed simulator, of which D. McKee is the main developer. The candidate uses the SEED simulator when evaluating the performance of her own algorithm. For the paper, the candidate mainly contributes to the literature review and the "practicality of SEED" chapter, in which the Google Cloud behavior was simulated as a use case of datacenter simulation. The candidate helped with the implementation of the job and the task characteristics, including average length and completion times, average resource requirements for CPU and memory, user submission rate, etc.

The papers that provide the basis for Chapter 5, the node execution performance modeling and prediction, are:

- *X. Ouyang, P. Garraghan, C. Wang, P. Townend, J. Xu, "An Approach for Modeling and Ranking Node-level Stragglers in Cloud Datacenters" in the proceedings of the 13th IEEE International Conference on Service Computing (SCC), pp. 673-680, 2016.*

- *X. Ouyang, C. Wang, R. Yang, G. Yang, P. Townend, J. Xu, "ML-NA: A Machine Learning based Node Performance Analyzer Utilizing Straggler Statistics" in the proceedings of the 24th IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2017.*

- *X. Ouyang, C. Wang, David McKee, P. Townend, J. Xu, "NEPAB: Node Execution Performance Aware Blacklisting for Straggler Mitigation within Cloud Environments".* [This manuscript is currently ready for submission]

In the SCC paper, a node execution performance modeling framework was proposed, and it was mainly based on statistical analysis. The idea of leveraging normalized task duration as a metrics was proposed by C. Wang. The candidate designed the modeling framework using this metrics, and demonstrated a case study through conducting the analysis on the Google data. P. Garraghan helped in the distribution fitting process as well as the sampling process. P. Townend and J. Xu joined the discussion and helped with the paper review process. The candidate then proposed a machine learning based node performance prediction framework in the ICPADS paper, implemented the prediction using the OpenCloud data. In this paper, C. Wang suggested the XGboost classifier, and advised on how to handle the timing attributes. R. Yang and G. Yang helped with the feature selection. P. Townend and J. Xu helped in reviewing the content. The upcoming journal paper is about adopting the result of the node execution performance prediction into the blacklisting mechanism of the YARN platform. The candidate was the main person who proposed the dynamic blacklisting design and implemented it for evaluation. C. Wang, David McKee, P. Townend, and J. Xu helped with the algorithm revision and the paper writing.

The papers that provide the basis for Chapter 6, the data skew straggler mitigation, is:

- *X. Ouyang, H. Zhou, P. Townend, J. Xu, "Mitigate Data Skew Caused Stragglers through ImKP Partition in MapReduce", in the proceedings of the 36th IEEE International Performance Computing and Communications Conference (IPCCC), 2017.*

By introducing a pre-processing layer in front of the normal Map phase, this IPCCC paper proposes an even partition algorithm in order to deal with the data skew type of stragglers for Reduce tasks, acting as a complement mechanism for the traditional speculative execution technique. The candidate proposes the algorithm, implements it based on the YARN system, and launches a series of experiments for evaluation. H. Zhou helped with the literature review process for other Reduce skew mitigating methods and helped with the workload implementation. P. Townend and J. Xu helped with the paper review.

Other papers that I have involved with are (the content of which are not included in the thesis):

- *P. Garraghan, S. Perks, X. Ouyang, D. McKee, I.S. Moreno, "Tolerating transient late-timing faults in cloud-based real-time stream processing" in the proceedings of the 19th IEEE International Symposium on Real-Time Distributed Computing (ISORC), pp. 108-115, 2016.*

- *D. McKee, S. Clement, X. Ouyang, J. Xu, R. Romanoy, J. Davies, "The internet of simulation, a specialization of the internet of things with simulation and workflow as a service (sim/wfaas)" in the proceedings of the 11th IEEE Symposium on Service Oriented System Engineering (SOSE), pp. 47-56, 2017.*

- *D. McKee, X. Ouyang, J. Xu, "Facilitating Dynamic RT-QoS for Massive-Scale Autonomous Cyber-Physical Systems", IEICE Transactions on Communications, 2018*

- *R. Yang, X. Ouyang, Y. Chen, P. Townend, J. Xu, "Intelligent Resource Scheduling at Scale: a Machine Learning Perspective", the 12th IEEE Symposium on Service Oriented System Engineering (SOSE), 2018.*

# Abstract

In order to satisfy increasing demands for Cloud services, modern computing systems are often massive in scale, typically consisting of hundreds to thousands of heterogeneous machine nodes. Parallel computing frameworks such as MapReduce are widely deployed over such cluster infrastructure to provide reliable yet prompt services to customers. However, complex characteristics of Cloud workloads, including multi-dimensional resource requirements and highly changeable system environments, e.g. dynamic node performance, are introducing new challenges to service providers in terms of both customer experience and system efficiency. One primary challenge is the straggler problem, whereby a small subset of the parallelized tasks take abnormally longer execution time in comparison with the siblings, leading to extended job response and potential late-timing failure.

The state-of-the-art approach to straggler mitigation is speculative execution. Although it has been deployed in several real-world systems with a variety of implementation optimizations, the analysis from this thesis has shown that speculative execution is often inefficient. According to various production tracelogs of data centers, the failure rate of speculative execution could be as high as 71%. Straggler mitigation is a complicated problem in its own nature: 1) stragglers may lead to different consequences to parallel job execution, possibly with different degrees of severity, 2) whether a task should be regarded as a straggler is highly subjective, depending upon different application and system conditions, 3) the efficiency of speculative execution would be improved if dynamic node performance could be modelled and predicted appropriately, and 4) there are other types of stragglers, e.g. those caused by data skews, that are beyond the capability of speculative execution.

This thesis starts with a quantitative and rigorous analysis of issues with stragglers, including their root-causes and impacts, the execution environment running them, and the limitations to their mitigation. Scientific principles of straggler mitigation are investigated and new algorithms are developed. An intelligent system for straggler mitigation is then designed and developed, being compatible with the majority of current parallel computing frameworks. Combined with historical data analysis and online adaptation, the system is capable of mitigating stragglers intelligently, dynamically judging a task as a straggler and handling it, avoiding current weak nodes, and dealing with data skew, a special type of straggler, with a dedicated method. Comprehensive analysis and evaluation of the system show that it is able to reduce job response time by up to 55%, as compared with the speculator used in the default YARN system, while the optimal improvement a speculative-based method may achieve is around 66% in theory. The system also achieves a much higher success rate of speculation than other production systems, up to 89%.

# Declarations

Some parts of the work presented in this thesis have been published in the following articles:

**X. Ouyang**, P. Garraghan, B. Primas, D. McKee, P. Townend, J. Xu, “Adaptive Speculation for Efficient Internetware Application Execution in Clouds”, ACM Transactions on Internet Technology, 18(2): 15, 2018.

R. Yang, **X. Ouyang**, Y. Chen, P. Townend, J. Xu, “Intelligent Resource Scheduling at Scale: a Machine Learning Perspective”, in the proceedings of the 12th IEEE Symposium on Service Oriented System Engineering (SOSE), 2018.

**X. Ouyang**, C. Wang, R. Yang, P. Townend, J. Xu, “ML-NA: A Machine Learning based Node Performance Analyzer Utilizing Straggler Statistics” in the proceedings of the 24th IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2017.

**X. Ouyang**, H. Zhou, S. Clement, P. Townend, J. Xu, “Mitigate Data Skew Caused Stragglers through ImKP Partition in MapReduce” in the proceedings of the 36th IEEE International Conference on Performance, Computing and Communications (IPCCC), 2017.

**X. Ouyang**, P. Garraghan, C. Wang, P. Townend, J. Xu, “An Approach for Modeling and Ranking Node-level Stragglers in Cloud Datacenters” in the proceedings of the 13th IEEE International Conference on Service Computing (SCC), pp. 673-680, 2016.

P. Garraghan, **X. Ouyang**, R. Yang, D. McKee, J. Xu, “Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters”, IEEE Transactions on Services Computing, 2016.

**X. Ouyang**, P. Garraghan, R. Yang, P. Townend, J. Xu, “Reducing late-timing failure at scale: straggler root-cause analysis in cloud datacenters”, Fast Abstract in the 46th IEEE/IFIP

International Conference on Dependable Systems and Networks (DSN), 2016.

**X. Ouyang**, P. Garraghan, D. McKee, P. Townend, J. Xu, “Straggler detection in parallel computing systems through dynamic threshold calculation” in the proceedings of the 30th IEEE International Conference on Advanced Information Networking and Applications (AINA), pp. 414-421, 2016.

P. Garraghan, D. McKee, **X. Ouyang**, D. Webster, J. Xu, “Seed: A scalable approach for cyber-physical system simulation”, IEEE Transactions on Services Computing, 9(2): 199-212, 2016.

P. Garraghan, **X. Ouyang**, P. Townend, J. Xu, “Timely long tail identification through agent based monitoring and analytics” in the proceedings of the 18th IEEE International Symposium on Real-Time Distributed Computing (ISORC), pp. 19-26, 2015.

# Acknowledgements

First and foremost, I would like to thank my supervisors: Professor Jie Xu and Dr. Paul Townend, my colleagues: Dr. Peter Garragham, Dr. David Mckee, Dr. Stephen Clement, Dr. Renyu Yang, Bernhard Primas, Yaofeng Chen, Jaber Almutairi; and those who offered me help in the School: Professor. Karim Djemame, Dr. Lydia Lau, Dr. Brandon Bennett, Judi Drew and Gaynor Butterwick. They are more like friends to me rather than supervisor, tutor, or colleague. I once thought it would be a challenge to me pursuing my dream alone in a country with different culture and language, however, out of my expectation, these people around are incredibly supportive and professional, generous and patient, helped me hugely in all aspects of my Ph.D. life. It has been a great honor to be part of the Distributed Systems and Services group, I will always remember the days we shared together here in Leeds.

My sincere thanks need to go to Professor Huaiming Wang, Professor Yuxing Peng, Dr. Bo Ding, Dr. Peichang Shi, Dr. Changjian Wang, Ms. Yuhua Hou, Pengfei Zhang, Yang Zhang, Yiying Li, Xiang Fu, Bo Liu, and Xiaoli Sun from the National University of Defense Technology as well. Thank you for sharing me with the data that facilitate our joint research, thank you for helping me in administrative matters while I am away, and thank you for inspiring me with productive discussions despite of the distance. Without all these, my Ph.D. won't be as smooth as it is.

I also want to thank the China Scholarship Council, without their scholarship program with the University of Leeds, it is impossible for me to spend these past four years here, the experience of which changed my life. In addition, I want to thank my friends and room-mates, for the meals we had together, the hiking we went together, and the holidays we spent together. Mengying Zhang, Mengrong Xu, Qian Yang, Yangmei Li, Zhongyang Xing, Guangming Li, Yuqing Wu, Xiaolan Shu, Ying Wang, Chunhong Yin, Ziyi Li, and so many other friends enriched my life here in UK which I really appreciated.

Last but not least, I would like to express my deep sense of gratitude to my family: my sweet mum, my dear dad, and my beloved husband. They are always encouraging, trust me in every decision I made; they are always warm, comfort me every time I feel depressed; and they are always wise, guide me to the right path every time I lost. They are the source of power when I overcome obstacles in life and when I pursue my dreams, and I hope they are proud of what I have achieved with their support. I love them, from the deep of my heart.

# List of Acronyms

| | | |
|---|---|---|
| **AM** | Application Master | 80 |
| **ANOVA** | Analysis of Variance | 38 |
| **AI** | Artificial Intelligence | 158 |
| **CDF** | Cumulative Distribution Function | 103 |
| **CI** | Confidence Interval | 100 |
| **DoS-Index** | Degree of Straggler Index | 68 |
| **DAG** | Directed Acyclic Graph | 108 |
| **DNN** | Deep Neural Networks | 158 |
| **EC2** | Elastic Compute Cloud | 36 |
| **ECT** | Estimated Completion Time | 80 |
| **FIFO** | First In First Out | 39 |
| **FPGA** | Field Pogrammable Gate Array | 1 |
| **GoF** | Goodness of Fit | 130 |
| **GPU** | Graphics Processing Unit | 1 |
| **HDFS** | Hadoop Distributed File System | 151 |
| **IoT** | Internet of Things | 150 |
| **IaaS** | Infrastructure as a Service | 14 |
| **SaaS** | Software as a Service | 14 |
| **PaaS** | Platform as a Service | 14 |
| **MPI** | Message Passing Interface | 21 |
| **ML** | Machine Learning | 157 |
| **NIST** | National Institute of Standards and Technology | 14 |
| **NM** | Node Manager | 125 |
| **OS** | Operating System | 39 |
| **PS** | Progress Score | 82 |
| **PR** | Progress Rate | 61 |
| **PDF** | Probability Density Function | 29 |
| **QoS** | Quality of Service | 150 |
| **REST** | REpresentational State Transfer | 13 |
| **RM** | Resource Manager | 125 |
| **RL** | Reinforcement Learning | 158 |
| **SaaS** | Software-as-a-Service | 14 |
| **SLA** | Service Level Agreement | 25 |

| | | |
|---|---|---|
| **SOA** | Service-Oriented Architecture | xvi |
| **SOAP** | Simple Object Access Protocol | 13 |
| **TPU** | Tensor Processing Unit | 1 |
| **UDDI** | Universal Description Discovery and Integration | 13 |
| **VM** | Virtual Machine | 152 |
| **WSDL** | Web Service Definitions Language | 13 |
| **YARN** | Yet Another Resource Negotiator | 8 |

# Contents

# List of Figures

# List of Tables

# List of Equations

# Chapter 1

# Introduction

## 1.1 Research Motivation

Modern day IT has grown at a substantial rate: with annual global IP traffic surpassing the zettabyte (1,000 exabytes) threshold in 2016 and IP traffic per capita reaching 22GB by 2019 [43]; with datacenters typically equipped with thousands and/or tens of thousands of machine nodes [13] and other heterogeneous hardware such as Graphics Processing Unit (GPU)s, Field Pogrammable Gate Array (FPGA)s, and Tensor Processing Unit (TPU)s [71][72]; with the emergence of new concepts such as the Internet of Things (IoT) [63] and industry 4.0 [126] that amalgamates big data analytics, the Cloud, and computing intelligence, etc. In order to meet the challenges brought by such unprecedented growth, a significant number of large-scale distributed interconnected systems which are capable of providing computing as a service have been developed.

Cloud computing has emerged as a powerful paradigm to facilitate these large-scale computing infrastructures, in which parallel and/or distributed computing techniques are applied to the solution of computationally intensive applications across networks of computers. This concept has achieved huge success in recent years. However, these models face significant performance challenges, especially with the rapid growth of cluster size. For example, under the current parallel computing assumption, when one distributed subtask performs abnormally slowly, all sibling tasks belonging to the same job have to wait for that straggler to complete. This straggler problem leads to significant performance deterioration: for users, Quality of Service (QoS) violation may occur due to job completion delay, which may end up with decreased user satisfaction; for system managers, committed resources wasted while waiting for stragglers result in poor utilization and financial loss. Thus, it is necessary to conduct extensive in-depth research to characterize and quantify straggler behavior, especially in large-scale and complex systems where this problem has already caused severe performance degradation.

All relating research problems proposed in this thesis are real problems that exist in commercial Cloud datacenters, such as Google (USA), Alibaba (China), and production clusters such as the OpenCloud cluster at Carnegie Mellon University which provides MapReduce services for students and faculties. Therefore, this research is of high practical value.

## 1.2 Aims and Objectives

The straggler problem refers to the situation when one or more parallel tasks perform significantly slower than other sibling sub-tasks, despite supposedly having similar durations. This impacts the overall job execution time as the parallel job has to wait until the last task has finished. The abnormally slower tasks are defined as *straggler* tasks. This research has two objectives, one is to improve parallel job execution performance through shortening the execution time by mitigating the stragglers, and the second objective is to save unnecessarily resources that would be spent on mitigating straggler behaviors by improving speculation efficiency.

These aims require an in-depth analysis of the straggler syndrome in current Cloud computing environments, a smart straggler identification method that always chooses the most urgent and suitable stragglers according to different operational environments and system behaviors, and an intelligent mitigation mechanism that predicts straggler occurrence and avoids assigning tasks to weakly performed machine nodes for execution. Stragglers can occur for many reasons, some of which are quite intuitive (for example, data locality will have a huge impact on task performance due to the significant larger latency for remote reads compared to local reads) while some are not so obvious (for example, some OS level background daemons such as garbage collection can temporarily affect machine performance). When stragglers are detected within a system, it is necessary to locate the main reason that leads to this performance degradation, so that appropriate method can then be taken to mitigate the impact of the straggling tasks. Specifically, the objectives of this research are as follows:

1. **Analyzing straggler related statistics within Cloud computing systems.** Parallel computing performance, especially job response time in this thesis, is very important for Cloud systems, due to the fact that services that promptly respond to requests will receive higher user satisfaction than those that take longer. One vital thing in maintaining high performance is to keep the tail of latency distribution short for parallel applications. Quantifying straggler related problems such as straggler impact and straggler reason is a key pre-condition to achieve that goal, and is challenging when the size and complexity of the system scales up and overall user volume increases. The first objective of this research is to measure the inefficiencies caused by stragglers within production Cloud computing systems, including quantifying the affected job population, wasted time, wasted resources, root causes, current mitigation efficiencies, etc.

2. **Identifying the most appropriate stragglers for mitigation.** The identification of stragglers is the foundation of most mainstream straggler mitigation methods, such as speculative execution [162], which follows the steps of (a) identifying stragglers, (b) launching speculative copy of those stragglers, (c) and adopting whichever result comes out first to generate the final response. From this perspective, it is good to conduct the identification process in a quick and precise manner, always picking up those tasks that are most likely to be caught up by the replicas to reduce possible waste. How to achieve this goal forms the second objective of this research.

3. **Avoiding straggler occurrence through modeling and predicting machine node performance.** Apart from mitigation, the other way of handling the straggler problem and relieving the long tail latency effect that deteriorates system performance, is through avoidance. Machine execution performance changes dynamically, and this is an important reason that leads to task response time variation. How to find key indicators to represent and model node performance, how to predict the changing tendency of machine execution performance, and how to develop dynamic node

blacklisting methods to avoid straggler occurrence and improve overall job execution are the key components of the third objective.

4. **Developing a dedicated algorithm to deal with situations when speculative execution is not appropriate.** Currently there are many straggler mitigation methods, but they are mainly based on the speculative execution scheme: copying badly performing tasks and launching new identical ones. These methods alone are not sufficient under certain circumstances, such as when dealing with degraded task performance due to data skew, because of the fact that speculative copies would experience the same delays when processing the identical data, leading to a speculation failure. Therefore, developing a dedicated algorithm that targets skew mitigation is the fourth objective of this research.

## 1.3 Methodology

The main methodology of this research follows the pattern of “data-driven analytics”, “simulation”, and “experimentation”.

For data-driven analytics, it is beneficial for the research to be inspired and enhanced by findings from real-world systems. Production cluster tracelog data is used to generate useful observations about straggler influence, straggler causes, and other straggler related issues such as current mitigation efficiencies, etc. Three production datasets have been available for this research: one is a Cloud datacenter tracelog released by Google. This dataset has been public since November 2011 and is available in [145]. The second version of this trace spans 30 days with the normalized processor and Memory usage metrics collected every 5 minutes. The trace describes the resource consumption of 12,000+ servers in operation, providing information on 25 million tasks grouped in 650,000 jobs. Additional information about the data structure, monitoring, and normalization process of this data can be found in [120]. The second dataset is a Cloud datacenter tracelog from Alibaba, an e-commerce platform in China. The size of this tracelog is relatively small compared to the Google one: it contains over 1,200,000 tasks and 2,800 servers. It should be noted that this is not a public dataset, and due to the confidentiality policies of the company, the raw data cannot be shared; only the final results concluded can be shown. The third dataset is from a Hadoop MapReduce cluster at Carnegie Mellon University [103]. Detailed information about each of the above datasets is introduced in

Section 3.1. Statistical analysis and data mining methods are used for the analysis. For example, Analysis of Variance (ANOVA) or correlation analysis can be used to explore straggler causes through building relationships between straggler occurrence and possible cause candidates.

For simulation, the number of production tracelogs that are of sufficient observational period and system size to perform in-depth analyses is very limited due to business and confidentiality concerns of users and providers in commercial Clouds. It is impossible to build a real production cluster just for research due to financial costs. This is why simulation methods have been used. CloudSim [35] is a representative simulator for Cloud computing environments; however, it is hard to configure and to implement custom algorithms into it. SEED [60] is a distributed environment simulator developed at the University of Leeds that allows for large-scale simulations to be created in a prompt manner. Unlike other simulators, SEED is capable of enforcing event-based synchronous simulation across loosely-coupled off-the-shelf distributed environments with no assumptions concerning the underlying hardware. Meanwhile, it minimizes user interaction through the use of XML-based protocols. All tasks simulated by SEED can be configured to follow the real Cloud datacenter characteristics revealed in [98], and it has been empirically demonstrated to effectively simulate Cloud operational behavior through experiments conducted at different simulation sizes and infrastructure scale [60], making it a good choice when simulating real massive-scale system behaviors; SEED is therefore chosen as the simulator for this research.

Experimentation is always the most convincing research method, since simulation can be inaccurate and may miss minor inconspicuous aspects which in the end lead to a totally different result. In this research, the Univesity of Leeds School of Computing Cloud testbed is used to build a MapReduce [46] cluster with the Hive system [141] running on top. In addition, an ExoGeni VM [52; 53] based Hadoop cluster is built as well, with Ambri [9] deployed as the monitoring tool. By implementing current straggler identification and mitigation methods as benchmarks, experimental comparisons are made as a supplement to the simulation results. Detailed experiment environments used when evaluating each algorithm can be found in each corresponding chapter.

## 1.4 Major Contributions

The major contributions of this thesis, corresponding to the four objectives, are:

1. **Quantitatively analyzed straggler influences, root causes, occurrence patterns and speculation inefficiency that provide new insights to straggler mitigation research.** The straggler problem is discussed in detail in a large volume of related literature; however, not many previous works ever conducted quantitative analysis, especially toward straggler influence and its root causes. Through leveraging three real-world datasets, universal observations are made. For example, around 5% stragglers at task level influence more than 50% of parallel jobs within large-scale systems, and high resource utilization ranks as the main reason behind straggler occurrence. Observations reveal that, on one hand, the straggler problem is severe and calls for a solution, while on the other hand, the dominant straggler mitigation scheme nowadays fails to do its job with good performance: the average speculation failure rate reaches as high as 71% in production systems, and there is another 66% of potential improvement space in average job response time if the straggler problem is solved.

2. **Developed a method that can adaptively identify the most appropriate stragglers according to the changing environment, shortening job execution time while saving resources used on speculation.** It is not easy to define, to what extent a slow task should be classified as a straggler that triggers a mitigation scheme: if too many tasks trigger speculation, the system will suffer from huge resource overhead when launching replicate copies, especially as some copies will end up being killed and wasted. However, if too few tasks are processed, severe stragglers can still exist and deteriorate job performance. The adaptive threshold method developed in this thesis answers this question by taking into consideration three key system parameters. Results show that the proposed method is capable of reducing parallel job response time by up to 20% compared to the current static threshold scheme (stragglers are defined as tasks with an estimated duration 50% longer than the average value), as well as a higher speculation success rate, achieving up to 66.67% against 16.67% in comparison to the static method.

3. **Designed a node performance analyzer that can be used in conjunction with the dynamic node blacklisting method to improve job execution.** Data analytics result reveals that node execution performance is not purely dependent on physical

capacity nor utilization level, but a complicated combination of reasons. Therefore in this thesis, a method is designed that leverages historical data of task executions to measure and model node performance. The machine learning based classifier proposed is capable of predicting node performance categories at an accuracy above 98%, and the dynamic node blacklisting scheme can improve job completion time by up to 55.43% compared to the default Hadoop YARN speculator and is capable of increasing the successful speculation rate by up to 89%.

4. **Proposed a data skew mitigation scheme that managed to decrease data skew caused stragglers for Reduce tasks.** Speculative execution can easily lead to bottlenecks when mitigating data skew caused stragglers due to its replicative nature: identical unbalanced input data will simply lead to slow speculative tasks. In this thesis, focusing on mitigating data skew caused Reduce stragglers, an intermediate key pre-processing framework is proposed that enables an even distributed partition for Reduce inputs. The proposed method can dramatically decrease Reduce skew, achieving a 99.8% reduction in the coefficient of variation of input sizes on average, and an improvement in job response performance of up to 29.37%.

## 1.5 Thesis Organization

The thesis is composed of seven chapters, of which this is the first:

**Chapter 2** is the related work chapter that provides an introduction to the background topics of computing systems and parallel job performance. Existing work in analyzing and modeling parallel job performance, specifically within the area of the straggler problem is introduced. Furthermore, the state-of-the-art research in straggler mitigation is discussed. This chapter is presented in order to understand the challenges associated with studying, quantifying, modeling, and mitigating straggler behavior within large-scale computing environments such as Cloud datacenters.

**Chapter 3** presents a quantitative analysis of straggler behavior, which details the motivation and illustrates the importance of straggler research. Leveraging parallel job execution traces in real-world datacenters, this chapter demonstrates straggler statistics analysis, straggler reason analysis, straggler occurrence pattern analysis, as well as a limitation analysis toward the state-of-the-art straggler mitigation method. The overall system model of the newly proposed intelligent straggler mitigation system is introduced in this section as well.

**Chapter 4** presents the adaptive straggler threshold algorithm, which is used to evaluate tasks, deciding whether a specific task should be defined as a straggler for potential mitigation under different system conditions. This algorithm is implemented into the current YARN platform, which is short for Yet Another Resource Negotiator (YARN). This chapter includes discussions of both experimental results and simulation evaluations.

**Chapter 5** presents the node execution performance modeling and prediction algorithm as well as the dynamic node blacklisting scheme, which can be used to evaluate machine nodes, deciding whether a specific node is suitable for launching task attempts. By avoiding assigning tasks to nodes that are about to experience performance degradation, straggler occurrence can be avoided and job response can be improved. Case studies based on the Google and the OpenCloud data are given, followed by experimental evaluations.

**Chapter 6** presents the mitigation algorithm for data skew caused stragglers, a dedicated method deals with the special type of stragglers. Skew mitigation is a special case that should be differentiated from the general speculative execution scheme, and this chapter discusses the skew mitigation, especially partition skews, under the MapReduce framework. Both synthetic and real-world inputs exhibit the skewed distribution are tested through experiments in this chapter.

**Chapter 7** summarises the related findings, provides conclusions toward the overall research, and outlines potential future directions for this work.

# Chapter 2

# Parallel Job Performance in Large-scale Computing Systems

This chapter describes the basic background concepts of this research - i.e. improving parallel job execution performance in large-scale computing environments with the presence of stragglers. The evolution of computing systems and relevant technologies are presented in order to better understand the emergence of Cloud computing and the prevalence of MapReduce framework. As an important challenge towards parallel job performance, the straggler problem is discussed, along with the concepts of Quality of Service (QoS), system dependability and availability. Finally, the state-of-the-art straggler mitigation methods are introduced, as well as the straggler root cause and skew mitigation methods. This is then followed by a discussion of the gaps in the current literature, highlighting the importance of the work within this thesis.

## 2.1 Evolution of Computing Systems

In order to better understand how the Cloud computing system emerges and develops, and why the MapReduce framework becomes popular in recent years, it is necessary to present the evolution of modern computing models.

### 2.1.1 Tiered Software System Architecture

The conceptual architecture of most software systems are designed by separation into three layers as presented in Figure 2.1 (a): the **presentation layer**, the **application logic layer** and the **resource layer**. There are other names for these layers such as the *business layer*, the *network layer*, or the *communication layer* defined in [42], [55], and [123], however they are all similar in functionality. The presentation layer is responsible for presenting information, interacting and communicating with components that exist outside of the system, for example, with human users or other systems. A presentation layer can be implemented in a number of ways including a graphical user interface or a component that formats data into a given syntax. The application logic layer is responsible for data processing, and it is the component that performs the actual operations requested by the user through the presentation layer. The resource management layer is responsible for the management of the data irrespective of the data source. This includes data residing within databases, file systems, and other data repositories.

The three layers above are conceptual designs that logically separates the functionality of the software system. From the implementation side, such layers can be combined and distributed in a variety of ways, referred to as **tiers**. There are mainly four types of software system architectures dependant on the organization of tiers: the **1-tier**, the **2-tier**, the **3-tier** and the **N-tier** architectures [65].

The 1-tier architecture merged all the three layers including presentation, application logic, and resource management into a single tier shown in Figure 2.1 (a). This type of system is essentially a monolithic piece of code, which is difficult to use and expensive to maintain. As a result, the 1-tier architecture is already viewed as legacy systems today.

The 2-tier architecture is an evolution of the 1-tier architecture, and it emerged due to the development of the PC. Through merging the presentation layer with the client (i.e. a user's PC), the 2-tier architecture separates the presentation layer from the server as shown in Figure 2.1 (b), therefore it is commonly known as “Client-Server” architecture. There are limitations with this architecture in terms of scalability when the number of clients interacting with the system increases, resulting in increased server overhead.

The 3-tier architecture as shown in Figure 2.1(c) features a distinct separation between the three layers. The presentation layer still resides on the client side, which is similar to the 2-tier architecture, while the application logic layer now sits within a middle tier which communicates between the client and the back-end resources. This middle tier, the infrastructure that is used to support the application logic, is referred to as the *Middleware* [23]. When integrating clients with multiple systems, the 3-tier architecture faces issues due to the lack of standards in terms of interfaces and communication protocols.

The N-tier architecture is similar to the 3-tier systems; the main difference is that they are more capable of linking to other systems, especially to the Internet as shown in Figure 2.1(d). The N-tier architecture has the same problem as the 3-tier in terms of lacking standards to enable interoperability between systems over the Internet, and consequently increases complexity and the amount of middleware required for system integration [22]. This is particularly true when application logic is distributed across multiple machines that each use heterogeneous middleware.

From the evolution of the tiered software systems, we observe several trends. For example, software systems are becoming increasingly complex, and there is an increasing requirement for software systems to be integrated together. SOA has emerged as a means to address these challenges, through changing the development of software systems into a dynamic and loosely coupled manner, it enables the integration of multiple systems across the Internet in a standard way.

### 2.1.2 Service Computing

**SOA** is proposed to provide the architectural style, or template, for building service-oriented systems, which *“promotes the idea of assembling application components into a network of services that can be loosely coupled to create flexible, dynamic business processes and agile applications that span organizations and computing platforms”* [114]. And the term **service** here is defined as *“a software implemented business function that is wrapped with a formally documented interface”* [115].

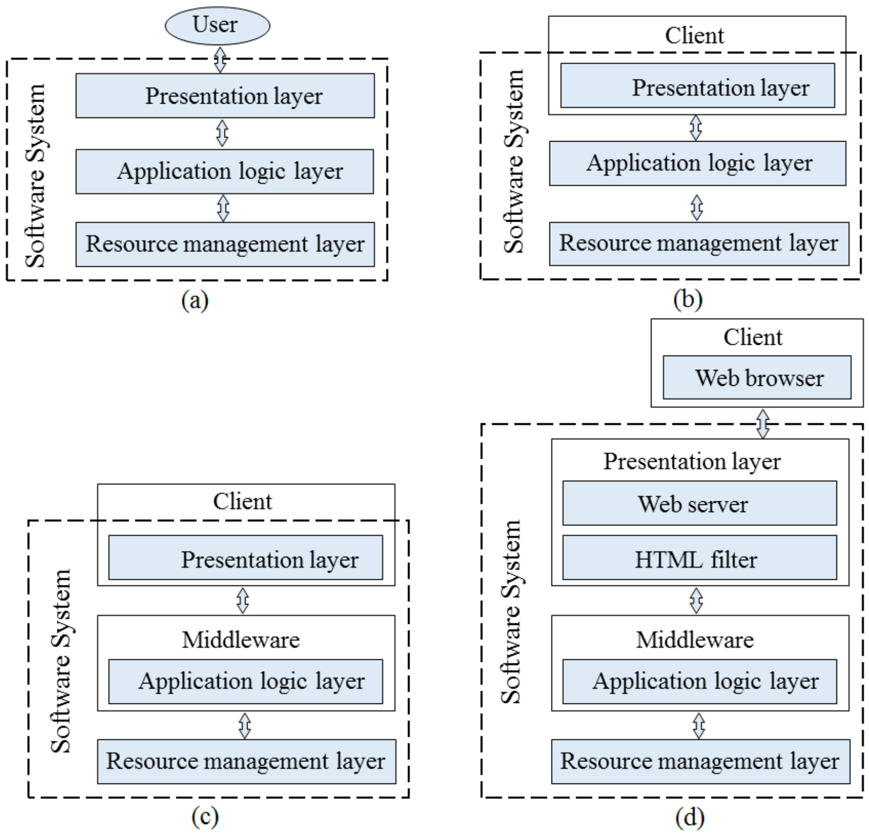


Figure 2.1: The (a) 1-tier, (b) 2-tier, (c) 3-tier, and (d) N-tier software architecture, among which, 1-tier architecture is the same with the conceptual layers of software systems

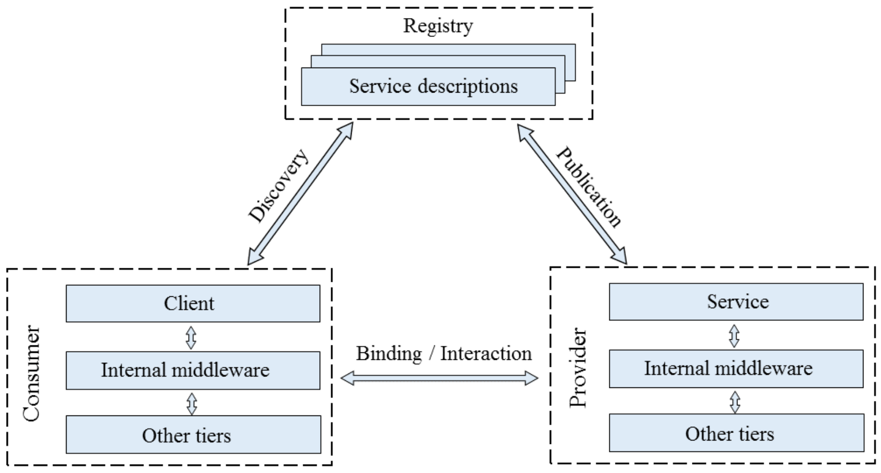


Figure 2.2: The SOA architecture triangle

Such architecture offers advantages over the N-tier architecture because the middleware complexity can be reduced due to middleware decentralization, and modifications are less likely to affect the existing services due to loose-coupling. The SOA system model is shown in Figure 2.2: there must be a service *provider*, a *consumer*, and a *registry* in order to complete the model. The provider publishes their services to the system, while the registry stores the service in the form of its service description. The registry also enables the discovery of new services using the Universal Description Discovery and Integration (UDDI) framework [102], and in some cases, it may provide further assistance to the consumer such as selecting the services, workload balancing and scheduling [93]. Then, the consumers are able to request particular functionality from the SOA system.

In addition, the development of cross-organization systems are well supported through the use of **Web Services**, defined as a *"self-contained, modular business applications that use standard interfaces over the Internet"* [150]. Actually, web service has now become the de-facto standard for most SOA approaches. It uses Web Service Definitions Language (WSDL) [136] when specifying interfaces, and the two popular service-based protocols for communication are Simple Object Access Protocol (SOAP) [96] and REpresentational State Transfer (REST) [124].

The maturity of service computing has enabled the resurgence of a long-sought concept: systems providing services to consumers as computing utilities. The idea of computing utility was proposed as early as 1961 [59], in a speech given by John McCarthy to celebrate MIT's centennial, where it was envisioned that networks would be highly developed, mature enough to make "computer utilities" a reality and worked in a similar principle to electrical and telephone utilities [48]. Through breakthroughs in research and technology, there appears a number of distributed systems within the past few decades, such as the peer-to-peer computing [128] and the grid computing [57]. Each of them with distinct characteristics to pursue specific consumer objective in order to realize the version of computing utility. Among them, Cloud computing is the most representative example.

## 2.1.3 Cloud Computing

The development of two major technologies increases the feasibility of Cloud computing: *communication protocols* and *virtualization* [34]. The former enables the formation of potentially distributed resource pools because computer systems are able to interact across the globe via the Internet. The latter enables the abstraction of computing resource from the physical infrastructure, in other words, users can dynamically add and release computing resource on demand through a virtual management system [19]. A typical use of the virtualization technology is the creation of **Virtual Machine (VM)**s, which are *self-contained environments containing encapsulated state and virtual computing resources*.

Cloud computing becomes increasingly popular after Google introduces its new business model of providing utility computing to consumers, in which computing resources, development platforms or applications are provided to consumers as a service. The two actors related to the Cloud service delivery are users and providers. Within the context of Cloud computing, **providers** are defined as *"entities that own and maintain the underlying infrastructure to provision computing service"* [34]. **Users** are entities that require computing power in order to achieve business objectives, which is defined as *"the actor responsible for creating and configuring the volume of tasks to be computed"* [34].

Although the concept of Cloud computing has emerged for many years, there is still no standard definition for it. One popular definition for Cloud computing is taken from the National Institute of Standards and Technology (NIST), which states that Cloud computing is *"a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"* [92]. Furthermore, from an implementation perspective, Cloud computing is defined as *"a parallel and distributed system consisting of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources base on service-level agreements established through negotiation between the service provider and consumers"* [32].

According to the level of system control, there are mainly three types of deployment models: public Cloud, hybrid Cloud, and private Cloud [85]. They are mainly different in terms of user access, system configuration, and security, etc. And according to the abstraction of the service provided, Cloud computing has three major forms: Software-as-a-Service (SaaS), representative examples including Gmail and Google Docs; Platform as a Service (PaaS), with Google App Engine to be the representative example of this kind; and Infrastructure as a Service (IaaS), with services such as Amazon EC2 fall into this category. The relations of these three forms are shown in Figure 2.3. In addition, there exist a number sub-categories that blur the boundaries of the three service models, including Failure-as-a-Service (FaaS) [64], Security-as-a-Service (SECaaS) [2], Workflow-as-a-Service (WFaaS) [90], etc.

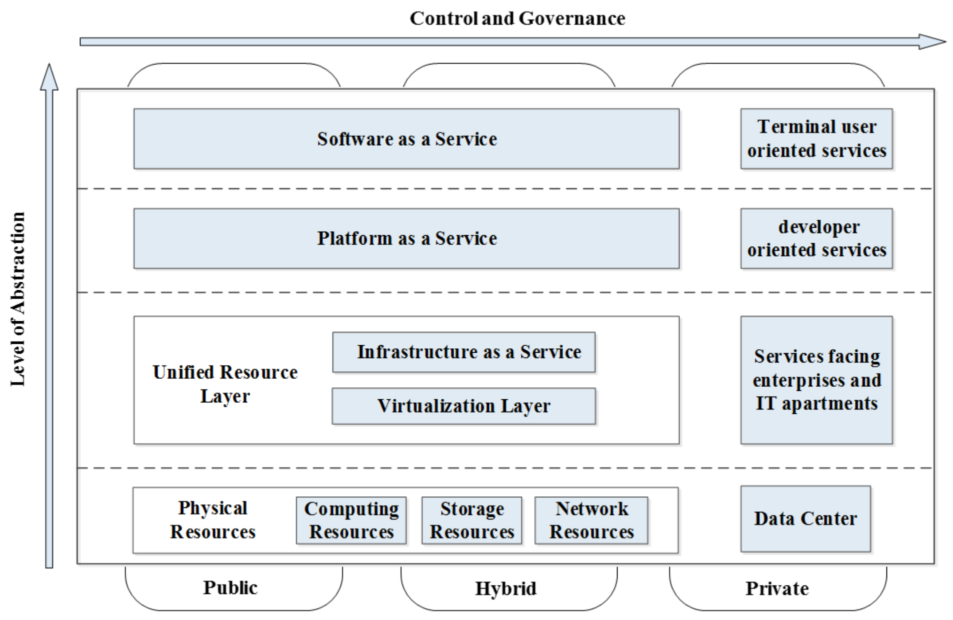


Figure 2.3: The Cloud service layer

There are four most essential features that distinguish Cloud computing as a unique computing model [146], they are: (1) *Pay-as-you-go*: users pay for the use of computing resources (such as storage and software) in an on-demand fashion; (2) *Scalability*: Cloud providers integrate large amounts of resources from thousands of servers. There is no up-front investment for users, so they can scale up or down rapidly whenever they want; (3) *Virtualization*: the key technology to create a pool of resources with different physical resources, and to assign these resources dynamically; and (4) *Internet Centric*: all public Cloud services are delivered over the Internet, so that users can easily access their resources with variety of devices. Due to these characteristics, Cloud computing provides a distinct advantage over traditional computing system models.

Cloud computing systems are typically deployed within datacenters or large-scale clusters. Datacenters are often colocated within the same physical location in order to satisfy common environmental and physical security requirements, as well as ease system maintenance [20]. The users of the datacenter are provisioned with physical space to purchase, install and configure their own IT equipment, while the datacenter providers are responsible for the physical security and operational environmental conditions.

Cloud computing supports the deployment of different services across application domains, with the promise of potentially unlimited power and scalability, and achieved a great success over the past decade. At the same time, there are other recent technological advances continuously been proposed, from which we observe some trend in the development of computing models, such as moving Cloud resources close to users.

### 2.1.4 New Computing Models

One of the future trends is the proposition of the Internet of Things (IoT). The term IoT was first coined in 1999 in the context of supply chain management [14], while recently this definition has been more inclusive covering a wide range of applications such as healthcare, transport, etc [15]. In the IoT paradigm, the objects in the environment surrounding us will be linked to a network in one form or another, in which information and communication systems are invisibly embedded. Technologies such as the Radio Frequency IDentification (RFID) [56], Wi-Fi [18], and sensor network [161] are supporting this new design into reality.

The unprecedented scale of interconnected objects will result in the generation of enormous amounts of data, which have to be stored, processed and presented in a seamless, efficient, and easily interpretable form [63]. Cloud solutions can improve Quality of Service (QoS) for applications in multiple domains, and available platform such as Amazon IoT [70] demonstrates the success of Cloud-centric IoT programming models and resource orchestration techniques. However, there are still some challenges to the pure Cloud-centric solution. Some examples are: offloading huge data in the Cloud comes with associated communication costs, latency issues, and privacy concerns, etc. The edge computing [132] model, which calls for processing the data at the edge of the network, has been proposed to solve these challenges. Figure 2.4 provides an overview of a typical environment that comprises a Cloud, edge, and mobile edge ecosystem [99].

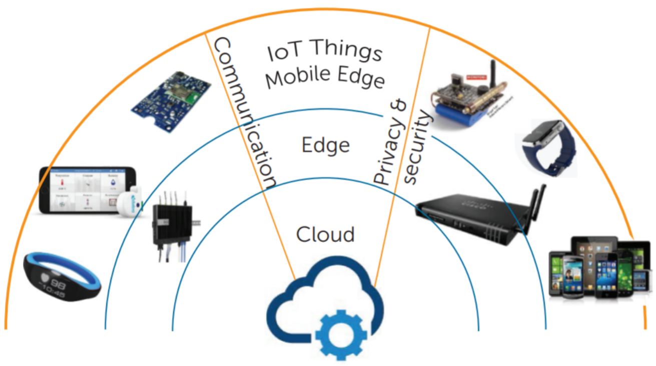


Figure 2.4: The Cloud, edge, and mobile edge ecosystem [99]

Besides edge computing, other forms of computing models such as the *fog computing* [86] and the *osmotic computing* [149], etc. are widely discussed within the community as well. The former one is focusing on mobile users, and the latter one is aiming at highly distributed and federated environments over both edge and Cloud infrastructures.

## 2.2 Parallel Computing and Execution Performance

Parallel computing has emerged as a means to leverage massive-scale computing infrastructure for data-intensive applications, in which parallel computation tasks are executed on multiple machine nodes by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing. The MapReduce framework [46] pioneered parallel computing model, and systems such as Hadoop [133], Spark [164] and YARN [147] generalized its population. This section focuses on introducing the fundamental principles of these parallel computing frameworks.

### 2.2.1 Basic Concepts in Parallel Computing

Basic concepts of parallel computing including cluster, job, and task, are first introduced in order to better understand the further discussion. A **cluster** is defined as *"a series of independent machines connected together by a network and through the use of middleware creating the illusion of a single system by abstracting the underlying infrastructure from users and reducing system complexity"* [31]. Large clusters are typically used in solving complex computing problems since they can deliver the capability at a reduced cost in comparison to traditional supercomputing systems. Under most circumstances, a cluster is composed of heterogeneous server nodes, varying in terms of the physical capacity of memory and CPU.

The **workload** is defined as *"the amount of work assigned to, or done by, a client, workgroup, server, or system in a given time period"* [50]. **Tasks** are defined as *"the basic unit of computation assigned or performed in the Cloud"*, and a parallel **job** is consists of multiple related tasks working towards a common objective.

Figure 2.5 [120] shows the simplified model of the states through which a task progresses. A task will be assigned to the "**pending**" state when it is waiting to be allocated after being

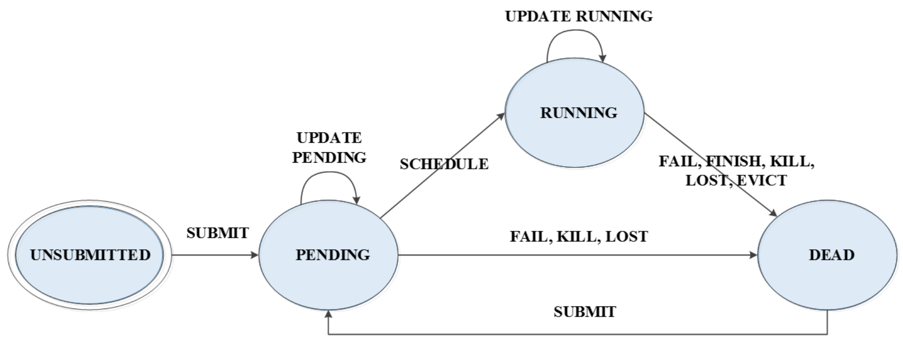


Figure 2.5: The task lifecycle

initially submitted by the user or re-submitted by the task scheduler. Once the scheduler finds a suitable server to allocate the task and the task is deployed, the status will be changed to “**running**”. When a task successfully finishes execution, it will transit to “**complete**” status and will subsequently be removed from the system. The task that is de-scheduled without successful completion will be transited to “**dead**” status.

Tasks have properties that describe their behavior. These attributes include the execution length and the amount of resource utilized as well as which job it belongs. Furthermore, Tasks can be characterized by the constraints which limit where the task can be executed. Such constraints include requiring a specific server hardware architecture, or geographical location due to security and privacy constraints.

The **cluster scheduler** is in responsible for addressing such challenge of appropriately scheduling tasks in the cluster, managing tasks and arbitrating resources between them [118]. This entails tracking machine liveness; starting, monitoring, and terminating tasks; and to decide task placements. Cluster schedulers are different from traditional OS/CPU schedulers, which are invoked for brief periods of time during context switches, and block a user-space process while making their decision. A cluster scheduler runs continuously alongside the cluster workload; its scheduling decisions last for a longer time; and it has more diverse and complex design goals than a single-machine CPU scheduler. Representative cluster schedulers include Mesos (2011) [67], Omega (2013) [130], Fuxi (2014) [166], Apollo (2014) [28], Borg (2015) [148], and gSched (2017) [37], etc.

Within the context of this dissertation, when referring to the term of scheduler, we are talking about cluster schedulers. Existing schedulers are differed in their architecture: the degree to which decisions are made in a centralized or distributed fashion. Figure 2.6 [129] details two different cluster scheduler architectures: the monolithic scheduling ar-

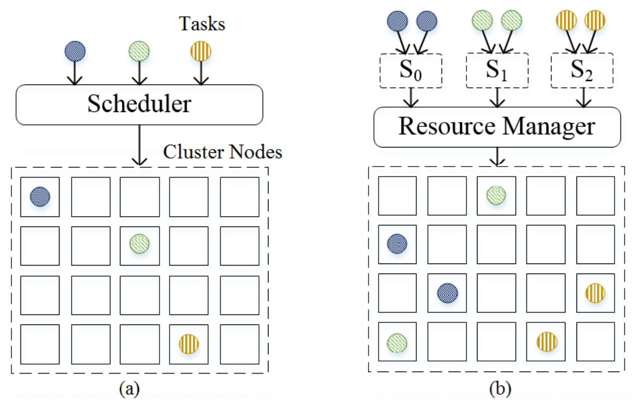


Figure 2.6: Different architectures of the cluster scheduler [129]

chitecture and the two-level scheduling architecture, with circles representing tasks and rectangles standing for machine nodes within the cluster. Figure 2.6 (a) represents the monolithic scheduler, where all tasks run through the same scheduling logic with a single scheduler process running on one machine. This design is simple and uniform, however meets bottleneck handling mixed workloads. In addition, overloaded scheduler can easily encounter single point failure, and there is an obvious limitation in large-scale resource management due to complex states. This is why in the design of the two-level scheduler shown in Figure 2.6 (b), resource allocation and task placement are separated. Representative implementations of each scheduler type is discussed in Section 2.2.3.

Based on these fundamental definitions of parallel jobs, tasks, and cluster schedulers, we now introduce the popular parallel computing frameworks.

## 2.2.2 MapReduce Framework

The most popular parallel programming model over the past years is the MapReduce framework. Defined by Google, **MapReduce** is "*a programming model and associated implementation for processing and generating large datasets*" [46].

The MapReduce framework is commonly used for dividing work across large-scale computing systems since it enables automatic parallelization and distribution of computations, and it can simplify the complexity of running distributed data processing functions across multiple nodes in a cluster. Figure 2.7 [46] illustrates the MapReduce procedure where all actions are logically carried out following the sequence marked. The corresponding meanings of each step are as follows:

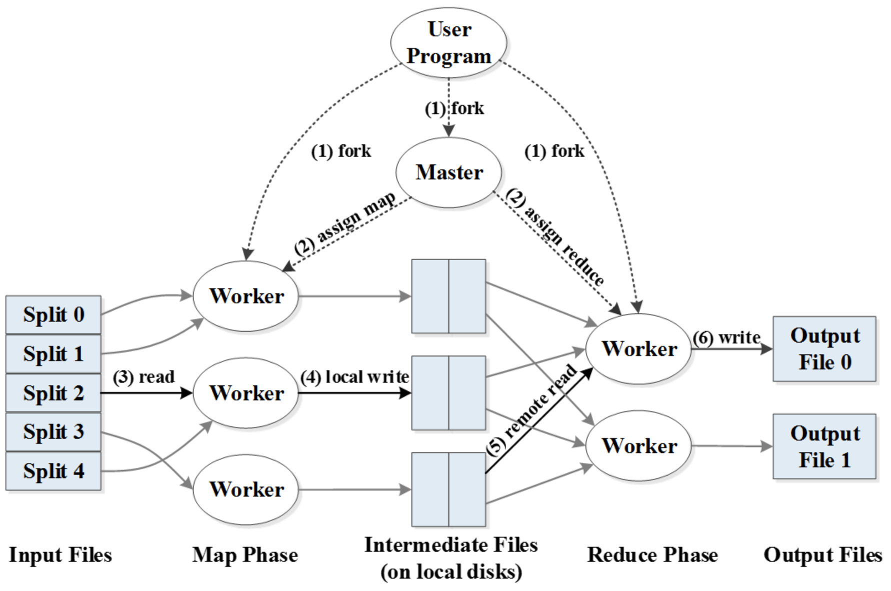


Figure 2.7: The MapReduce workflow [46]

1. **Fork**: The MapReduce library in the user program splits the input files into $M$ pieces, the size of which can be controllable by the user via an optional parameter.
2. **Assign Map / Reduce**: Among the program copies there is a special one called the *master*, while the rest are called the *workers*. The master picks idle workers and assigns each one with a Map task (Mapper) or a Reduce task (Reducer).
3. **Read**: The worker who is assigned with a Map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function.
4. **Local write**: The intermediate key/value pairs produced by the Map function are written to the local disk, partitioned into $R$ regions by the partitioning function. The locations are passed back to the master which is responsible for forwarding these locations to the Reducers.
5. **Remote read**: When a Reducer is notified by the master about the locations, it uses remote procedure calls to read the data from the Map local disks. When the Reducer has read all intermediate data for its partition, it will sort them by the intermediate keys so that all occurrences of the same key are grouped together. Sorting is needed because typically multiple keys will be mapped to the same Reducer.
6. **Write**: The Reducer iterates over the sorted intermediate data. For each unique intermediate key encountered, the Reducer passes the key and the corresponding set of intermediate values to the Reduce function. The outputs will be appended to a final output file for each Reduce partition.

These procedures and the related concepts are important when talking about problems observed in such model. For example, according to the design, the Mappers should be reading local input files, when the cluster scheduler fails to achieve this goal, the MapReduce job performance would experiencing a sharp fall. More performance challenges will be analyzed in the following sections.

Google MapReduce is an extraordinary innovation in parallel computing after the Message Passing Interface (MPI) technique [62], and there are a lot of open-source implementations such as Hadoop, Spark, and YARN.

### 2.2.3 Open-Source MapReduce Implementations

Apache Hadoop [133] is the most popular open-source version of the MapReduce framework, and achieves a huge success among past few years: Google, Yahoo!, Facebook, Amazon use it to build their Cloud services [12], manage and process terabytes of data per day. Web data-intensive applications, scientific data-intensive applications (e.g., natural language processing) prefer to employ the Hadoop system because it has been applied with a high degree of reliability, extensibility, effectiveness and fault tolerance, and all of those are transparent to programmers.

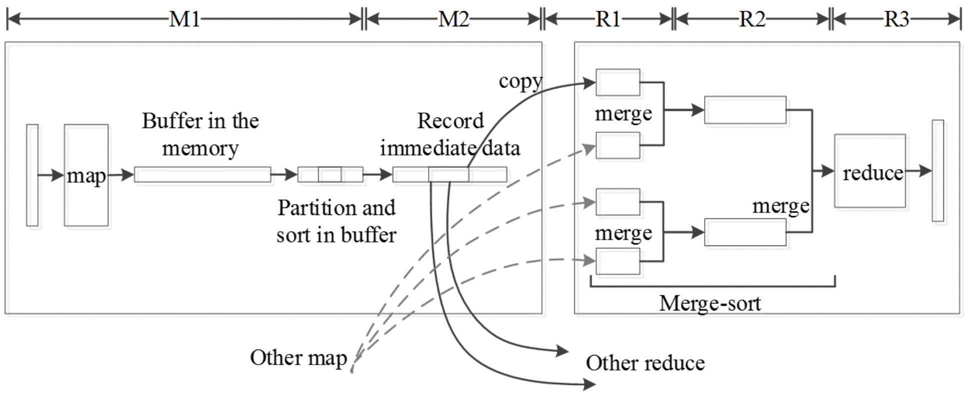


Figure 2.8: Refinement of the Map and the Reduce phases for Hadoop

A typical Hadoop job can be divided into a Map phase and a Reduce phase in coordinate with the MapReduce framework. To be more specific, Map phase can be further refined to the *Map function execution* (M1) and the *sort/partition* (M2) sub-phases. A Reduce task can be divided into sub-phases of *copy* or shuffle (R1), *sort* (R2) and *Reduce function execution* (R3). Figure 2.8 illustrates these sub-phases. Hadoop runs several Map and Reduce tasks concurrently on each machine node, two of each by default, controlled by

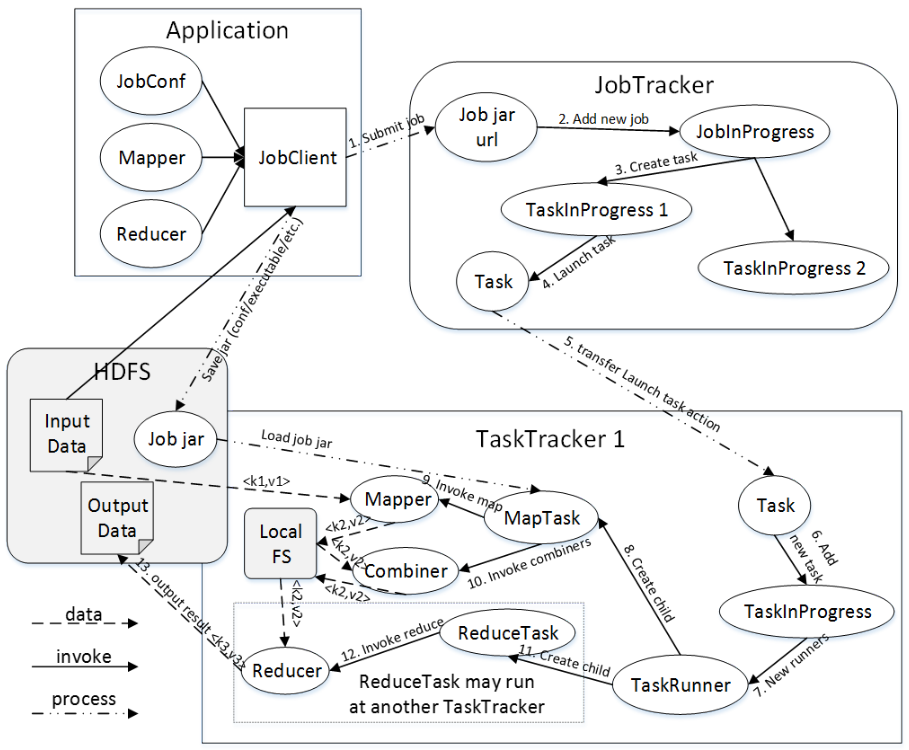


Figure 2.9: The Hadoop V1 architecture

the parameter called task **slot**, which is defined as *"the maximum number of parallel Map and Reduce tasks that can run on a slave node"* [154].

The detailed Hadoop architecture is shown in Figure 2.9. The per-machine daemon **TaskTracker** informs the centralized **JobTracker** when there are empty task slots available for tasks to be assigned. Users submit jobs directly to the JobTracker, the central arbiter of the cluster responsible for admission control, tracking the liveness of TaskTrackers through periodical heartbeat, re-execute tasks whose output becomes unavailable, launching tasks speculatively, reporting job status to users, recording audit logs and aggregate statistics, authenticating users, etc. Such design belongs to the monolithic scheduling architecture discussed in Section 2.2.1.

The Hadoop system evolves into the second version, Hadoop V2 or Hadoop YARN, decoupling the JobTracker into Resource Manager (RM) and Application Master (AM) in order to solve the aforementioned challenges that inspire the design of the two-level scheduling architecture in Section 2.2.1. The system model of YARN is shown in Figure 2.10: RM is the global resource manager that manages cluster resources; Node Manager (NM) is the new per-node slave that responsible for launching containers, monitoring resource usage (CPU, memory, disk, network, etc) and reporting back to the RM. The AM is in

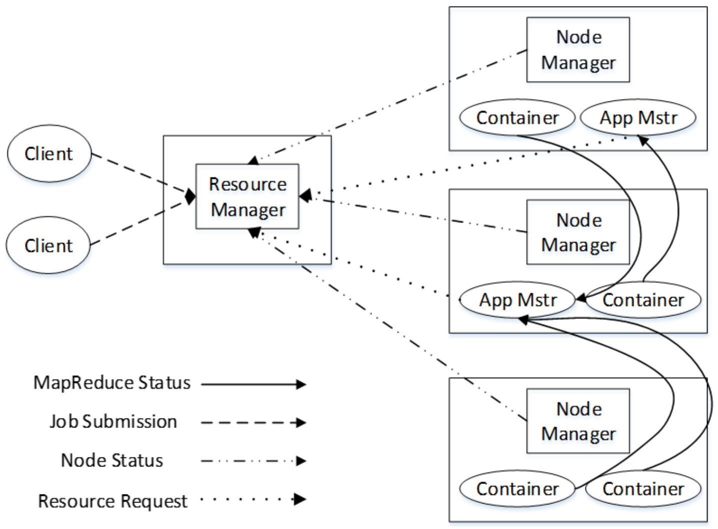


Figure 2.10: The Hadoop YARN architecture

charge of job scheduling and monitoring (one AM per application). Detailed responsibilities of AM include job creation, resource requests from the RM, communications with the NM to run containers, job running status report and speculation. All these functions are implemented in an event trigger mechanism shown in Figure 2.11.

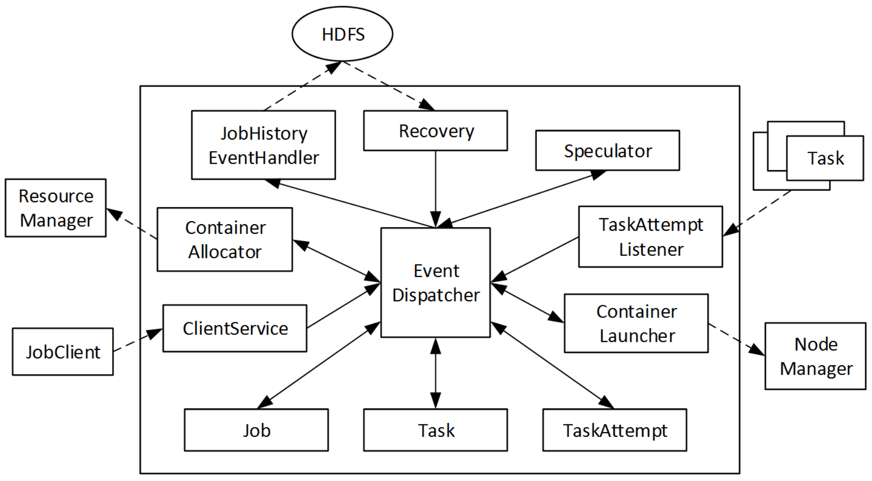


Figure 2.11: The events handled by YARN AM

There are other popular parallel computing systems focusing on specific type of application optimization such as Spark [164]. The Hadoop or the YARN systems are built toward an acyclic data flow, which is not suitable for some of the popular applications. Spark is designed for such application that reuses a working set of data across multiple parallel operations. One representative example is iterative machine learning algorithms, which applies a function repeatedly to the same dataset to optimize a parameter, e.g., through gradient descent, while each iteration can be expressed as a MapReduce job.

There are other systems that also focus on iterative programs, which arise naturally in applications including data mining, web ranking, graph analysis, and model fitting, among them are Twister [49] and HaLoop [29][30]. Twister is an enhanced MapReduce runtime with an extended programming model that supports iterative MapReduce computations. It uses a publish/subscribe messaging infrastructure for communication and data transfer, and provides programming extensions to MapReduce with "broadcast" and "scatter" type data transfers. For the latter, HaLoop allows iterative applications to be assembled from existing Hadoop programs without modification, significantly improves job execution efficiency by creating an inter-iteration caching mechanism, and proposes a loop-aware scheduler to exploit these caches. In addition, it retains the fault-tolerance properties of MapReduce through automatic cache recovery and task re-execution.

Within this thesis, general performance challenge such as the straggler behavior in parallel job execution is discussed, instead of a specific optimization for a target type of application, therefore in later sections, we simply use Hadoop YARN as a showcase for proposed algorithms.

## 2.3 Performance Challenges in Parallel Execution

Parallel computing systems can be characterized by five fundamental properties: functionality, performance, cost, security and dependability [17]. Services have functional and non-functional requirements in terms of performance. Specifically, the functional requirement is the intended purpose of the system, the business logic offered for end users. The non-functional requirement denotes the features of the service, such as the execution time, etc. In this section, the fundamental concepts of performance are introduced, followed by a detailed introduction to a specific performance challenge encountered by parallel computing frameworks in time: the straggler problem.

### 2.3.1 Basic Concepts in Performance Challenge

The quality and performance of the service provided are important in Cloud computing systems, while the level of service required by users can vary significantly depending on their business objectives. As a result, it may not be possible to fulfill all expectations for all users from a service provider's perspective, hence, a trade-off needs to be made via a negotiation process.

Providers and users commit to a Service Level Agreement (SLA) as a result of the negotiation between user requirements and provider's ability to fulfill those expectations. The **SLA** is defined as *"a firm definition of the service agreement between service providers and users, which includes service level guarantees, parameters and actions required in the case of violation"* [117], and it details the level of acceptable service [108]. Service parameters are measurable representations of obligations in order to measure whether service has been satisfactorily provisioned [87] among service parties.

Typical parameters of Cloud SLAs include availability, performance, monitoring, cost, security and reliability [8]. One element provisioned and enforced by the SLA is the Quality of Service (QoS), which comprise of a large number of parameters, ranging from 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. A timing failure [16] occurs when this time frame is violated. Service providers use the SLA to optimize their infrastructure to meet the agreed terms of service, whilst service users use it to ensure that the agreed QoS has been fulfilled. QoS deadline constraint is considered by a lot of research, focusing on different aspects. Some work emphasize the resource management perspective [142], while some explore how the data locality influences QoS, and etc. To fulfill the timing requirement is a very important branch of the performance research.

Apart from above, there are many performance challenges parallel computing systems could encounter. For example in MapReduce clusters, each node may serve as both compute node and data node, therefore tasks could be scheduled on the node containing the task's input data. "Data locality" is used to describe whether a task is executed close to the data. Research in [152] show that for their reference job, although 98% of the tasks running on the node containing the data and another 1% of the tasks running in the same rack as the node containing the data, there is still 1% of the tasks encounter the data locality challenge which may incur significant performance penalties.

User behavior impacts data locality as well, for example, the number of Map tasks configured by the user. Work in [163] claims that, according to their observation, the same rack locality reaches 90% at about 100 Maps per job, and goes up to about 98% as the number of Maps per jobs increases, and the same node locality reaches 90% at about 7,500 Maps per job and goes up to about 92% as the number of Maps per job increases. Besides this, user behavior can influence a wider variety of system performance. For example, the task submission rate in Cloud computing impacts resource usage [97], hence influences letency and energy efficiency [111]. Due to the importance of the user behavior towards Cloud application efficiencies, there are a lot of work focusing on user behavior/demand predictions such as [113] and [110].

Besides user behavior, different workload type leads to different performance challenge as well. Some workloads have repetitive patterns, also named as periodic workload [112]. For example, the Wikipedia workload has a diurnal pattern where requests arrive at a more intense rate during the daytime rather than at night [3]. Some workloads have weak patterns or no patterns at all (also referred as random workload), while other workloads have encounter uncorrelated spikes and bursts due to unusual events [91][26]. Workloads can also be classified by their function. Example workloads such as *batch processing* are normally used for computationally intensive scientific computing [24][105], and *latency-sensitive* are typically seen in real-time applications [74]. Table 2.1 lists some common examples of workloads within Cloud environments, identified by IBM [139].

Table 2.1: Classification of typical Cloud workloads

| Workload Type | Description / Examples |
|---|---|
| Analytics | Business analytics, data mining, temporal and spatial patterns within submitted datasets |
| Gaming | Latency sensitive gaming applications |
| Web Applications | eCommerce, Java application, web searching |
| Batch Processing | CPU intensive render farms for 3D modelling, visualization of large scale geo-data and performing large numerical calculations |
| MapReduce | Large-scale data analytics applications |

The machine and workload heterogeneity and variability, the highly dynamic yet poorly predicted resource demands and availability, are the major issues faced by efficient job execution in Clouds [121]. Among the many parallel job performance challenges, there is a noticeable topic hindering service timing boundary fulfillment that attracts our attention: the straggler problem. The details about this problem is discussed in the following section.

### 2.3.2 The Straggler Problem

The straggler behavior describes the phenomenon when a distributed job - composed of multiple tasks executing in parallel - incurs a significant delay in completion time due to a

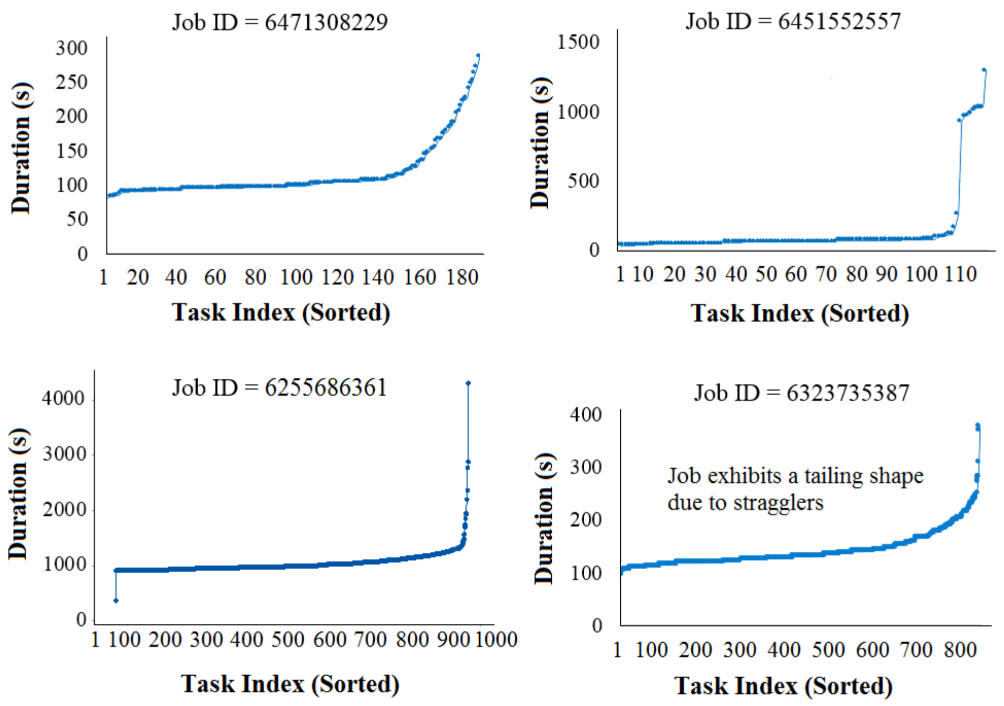


Figure 2.12: Task duration pattern for jobs exhibiting stragglers in the Google cluster.

small subset of its parallelized tasks - known as stragglers - performing much slower than the other sibling tasks. Under the parallel computing assumption, the final result of a job, excepting the approximate ones, will not be generated unless all results from its sub-tasks are being calculated, thus the straggler completion impedes overall job completion [162].

In order to better illustrate the straggler problem, we depict the execution pattern figures for four completed parallel job within a Google cluster [145] [120] in Figure 2.12, the detailed introduction of the Google dataset is given in Section 3.1. From the figure it is observable that, although each job exhibits different task size and duration, they are all characterized by a tailing shape, with the slowest task taking up to more than 10 times longer compared to average duration. As a comparison, Figure 2.13 portraits another two normal parallel jobs within the same cluster that do not contain stragglers, with approximately the same durations for all subtasks.

The straggler problem is intensively discussed under the MapReduce framework, an intuitive example is shown in Figure 2.14, running a MapReduce job (wordcount) in our own Hadoop cluster deployed upon 10 Virtual Machine (VM), with the VM provided by the University cloud testbed running OpenNebula service [144]. A very clear Reduce straggler can be observed from this example, taking more than twice the duration compared with other sibling Reducers. We define the jobs that suffer from the straggler problem as

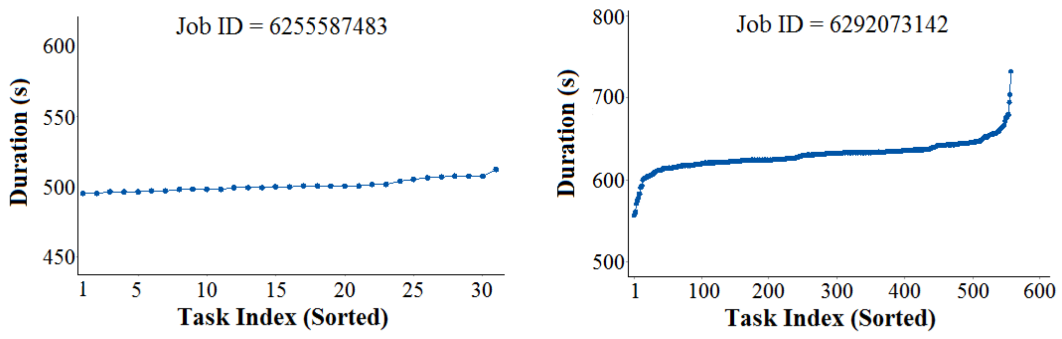


Figure 2.13: Task duration pattern for normal jobs without stragglers in Google.

the *tailing jobs* within this thesis, and there exist other terms representing similar meaning in other literature, including the term "outlier problem" used in [5], and the term "long tail problem" used in [45].

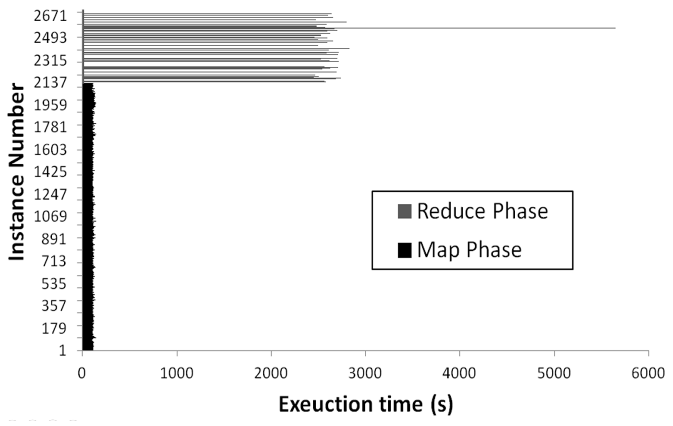


Figure 2.14: Task duration pattern for an example MapReduce job with a Reduce straggler

Stragglers directly lead to two consequences: service late-timing failure and resource waste. For the former, extended response time increases the possibility of QoS timing constraint violation, while for the latter, committed computing resources are wasted on waiting because they cannot be assigned to other users with other tasks. In addition, user satisfaction will be affected if the rapidness of service response time cannot be guaranteed.

Stragglers are not exceptional cases that tasks rarely encounter, on the contrast, they are a common phenomenon that undermines parallel computing efficiencies, and becomes increasingly severe in the face of increased system scale. Industry system such as Google measures that, the slowest 5% of the requests to complete is responsible for half of the total 99%-percentile latency in their production cluster, and in the face of system scale growth, the probability of longer latency increases [45]. A research based on data collected from Microsoft Bing's production cluster claims that, 80% of the stragglers (the term *outliers* is used in their research, which is a synonym for stragglers) have a uniform probability of being delayed by between 150% to 250% compared to the medium job duration, while 10% take more than 1000% the median duration [5].

The straggler problem is a general challenge in the parallel computing field, and can be observed in many environments rather than only limited within large-scale clusters. This includes environments such as multi-core computers [82] or Internet-scale applications [165]. Within this thesis, we mainly focused on stragglers observed in large-scale computing systems that act as the infrastructure supporting Cloud services.

### 2.3.3 Straggler Related Formulation

As mentioned previously, some literature call straggler problem as the "long tail problem" due to the fact that the job encounters the straggler behavior normally exhibits a tailing-shaped task execution distribution. In mathematics, there is a concept named *long tail distribution*. Let $F$ be a Cumulative Distribution Function (CDF) and let the associated complementary Probability Density Function (PDF) be $F^c(x) = 1 - F(x)$. Then, a long tail distribution (also known as fat tail or heavy tail) is defined as the distribution when its $F^c$ decays slower than exponential [54], i.e., for all $\gamma > 0$

$$\lim_{x\to\infty} e^{\gamma x} F^c(x) = \infty \tag{2.1}$$

In contrast, a short tail distribution is defined as a distribution that with a $F^c$ decaying exponentially, i.e., there exists some $\gamma > 0$ such that

$$\lim_{x\to\infty} e^{\gamma x} F^c(x) = 0 \tag{2.2}$$

These definitions given in Equation 2.1 and Equation 2.2 are intended for general classification, and do not describe the actual decay rates of $F^c$ well. In other more specific definitions such as the one in [151], a typical long tail is described as a polynomially decaying distribution that, for $t > 0$,

$$\begin{cases} \lim_{x\to\infty} \frac{F^c(tx)}{F^c(x)} = t^{-\xi} & \text{if } \omega(F) = \infty \\ \lim_{x\to 0^+} \frac{F^c(\omega(F)-tx)}{F^c(\omega(F)-x)} = t^{\xi} & \text{if } \omega(F) < \infty \end{cases} \tag{2.3}$$

where $\omega(F)$ represents the upper end point of $F_X(x)$

$$\omega(F) \triangleq sup\{x : F_X(x) < 1\} \tag{2.4}$$

The lower case in Equation 2.3 corresponds to the case that $F^c$ has a heavy tail with a finite upper bound. Similarly, the short tail is described that, there exists $\eta(x) > 0$ for

$$\lim_{x \to \omega(F)^-} \frac{F^c(x + t\eta(x))}{F^c(x)} = e^{-t} \tag{2.5}$$

To simplify the understanding, we illustrate these definitions with two graphs as shown in Figure 2.15, the PDF and CDF of exponential distribution and Pareto distribution. Among these two, exponential distribution is a short tail distribution in mathematical definition, while Pareto distribution is a famous long-tail distribution.

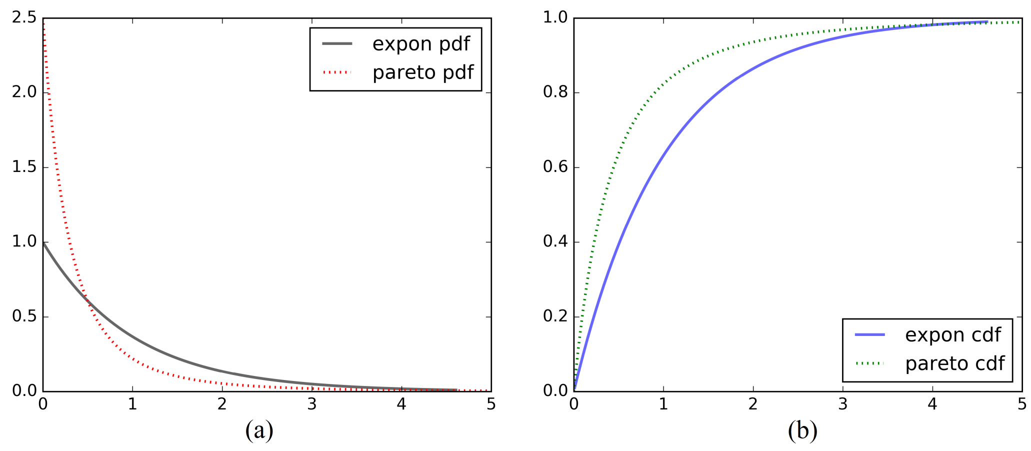


Figure 2.15: The (a) PDF and (b) CDF of the exponential and pareto distribution.

From the figure, especially the PDF figure, it is shown that, compared to the exponential distribution, it is better to use Pareto distribution when modeling task duration patterns of a parallel job, because it can better describe the extreme slow stragglers when they occur at a low possibility. Other long-tail distributions includes Weibull distribution, and etc.

## 2.4 Overview of Straggler Mitigation Techniques

Given the challenge of handling stragglers and their impact toward system performance, this section introduces the existing prevalent techniques related to straggler mitigation.

### 2.4.1 Speculative Execution and Its Variations

Speculative execution is the most popular method used in eliminating the straggler problem. It first identifies a straggler and launches a replica task of that straggler which typically performs identical work; then it will adopt the result of whichever copy that finishes first to complete the job and abandon the other slower one. The main idea of the naive speculative execution is shown in Figure 2.16. At time $t_0$, the four parallel tasks $T_{jA}$, $T_{jB}$, $T_{jC}$ and $T_{jD}$ are laughed, with initial estimated completion time computed. The system monitors the progress of these four tasks, at time $t_1$, the completion time of task $T_{jA}$ is estimated to be much larger (exceeds a threshold) than the average completion and be identified as a straggler. As a result, a speculative copy $T_{jE}$ is launched at time $t_2$ to mitigate $T_{jA}$. In the end, at time $t_3$ when $T_{jB}$, $T_{jC}$, $T_{jD}$ and $T_{jE}$ are finished, the scheduler will adopt the result from $T_{jE}$ and kill $T_{jA}$.

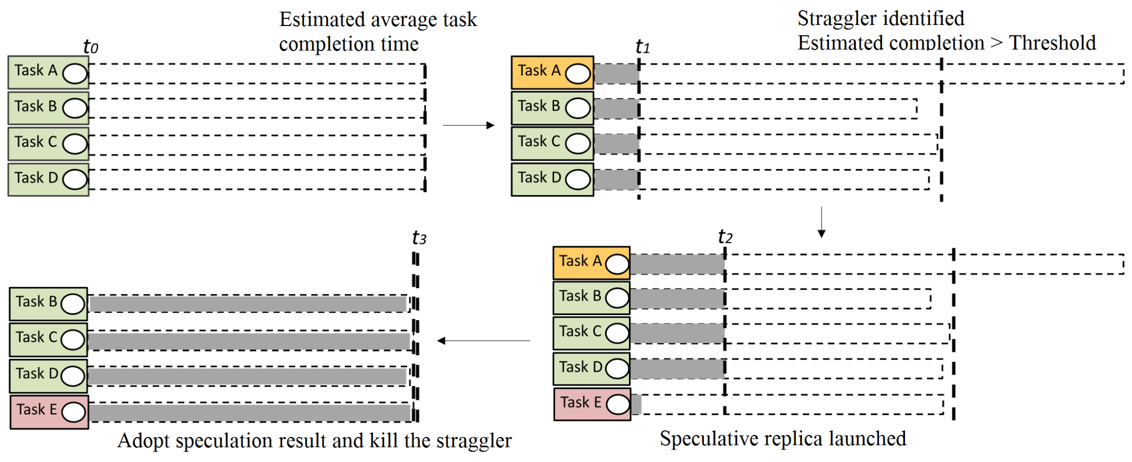

Estimated average task completion time
t0
Task A
Task B
Task C
Task D
Straggler identified
Estimated completion > Threshold
t1
t2
Task E
Speculative replica launched
t3
Adopt speculation result and kill the straggler

Figure 2.16: Schematic diagram of the speculative execution algorithm

Based on the above analysis, it is known that, the identification of stragglers plays an important role in speculative-based approaches, and it is largely influenced by the straggler **threshold**, which is defined as *"a ratio number that calculates the difference between an individual task and average task progression for a job, representing to what extent a slower task should be defined as a straggler within the system"*. Currently, there are three popular types of threshold in state-of-the-art literature, they are:

- Progress Score (PS) based threshold

Hadoop's default speculative mechanism uses the metrics of PS, ranging from 0 to 1, to measure the execution progress of a task [155], the calculation of which is given in Equation 2.6

$$PS_{ji}^{t} = \begin{cases} L/N & \text{For Map tasks} \\ 1/3 * (P + L/N) & \text{For Reduce tasks} \end{cases} \tag{2.6}$$

where $PS_{ji}^{t}$ represents the progress score of task $T_{ji}$ at time stamp t, the $i^{th}$ task of job $J_j$. The value of $PS_{ji}^{t}$ is bounded between 0 to 1, representing the start and the end point of $T_{ji}$, respectively. For a Map task, PS is the fraction of input data read. In Equation 2.6, the number of key/value pairs need 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 the *copy* phase, the *sort* phase and the *reduce* phase. Detailed functionalities of these three phases is introduced in section Section 2.2.3. Each phase accounts for 1/3 of the final PS. This even weighting can be modified through changing scheduler settings, for example the work of [40] assigns different weightings based on historical data to calculate more accurate PS for straggler identification. 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 two thirds of key/value pairs have been processed, while 0.667 for a Reduce task indicates it has finished the *copy* and *sort* phases, and will shortly commence the *reduce* phase.

When adopting this type of threshold, the scheduler identifies a task $T_{ji}$ as a straggler only if when $PS_{ji}^{t} \leq Th_j * \overline{PS_j}$, where $\overline{PS_j}$ is the average PS of all tasks belonging to job $J_j$ and $Th_j$ is the threshold. This type of threshold has an unavoidable limitation where tasks that have completed more than the pre-defined progress can never be speculatively executed. For example, according to the definition, if we define stragglers as the tasks with a PS 20% less than average PS for a certain job, than the straggler phenomenon after 80% PS will never be tacked by the system. To avoid this problem, the progress rate based threshold is proposed and adopted for straggler identification.

- Progress Rate (PR) based threshold

The calculation of PR is defined in Equation 2.7, dividing the PS with the corresponding elapsed time, and is a metrics used to measure the task's processing speed.

$$PR_{ik}^{t} = \frac{PS_{ik}^{t}}{t - t_0^{T_{ji}}} \tag{2.7}$$

In the equation, $t_0^{T_{ji}}$ represents the start time of $T_{ji}$ while $t$ denotes the current time stamp.

To note that, in reality, due to queueing reasons, $t_0^{T_{jk}}$ and $t_0^{T_{jp}}$ $(k \neq p)$ can be different time stamps. However in our model, we simplify the case with the assumption that $t_0^{T_{jk}} = t_0^{T_{jp}}$. We adopt this assumption in the following context of the thesis unless specifically pointed out. Similarly with the previous threshold, a task will be identified as a straggler if $PR_{ji}^t \leq Th_j * \overline{PR_j}$. Dolly [6] uses this type of threshold, identifying stragglers as the tasks with PR less than 50% of the average PR compared to their siblings, in which case $Th_j = 50\%$. Worth mentioning, Wrangler [159] uses the PR reciprocal as the threshold, represented as $nd(T_{ji})$ and calculated as $\frac{t - t_0^{T_{ji}}}{PS_{ik}^t}$, identifying stragglers as the tasks that fulfill the condition of $nd(T_{ji}) \geq \beta \times \underset{\forall T_{ji} \in J_j}{median}\{nd(T_{ji})\}$.

This type of threshold comes with its own limitations as well: the PR can change during different progress phases. Taking the following scenario as an example, if task $T_{11}$ is three times slower in PR than the average yet has a PS of 0.9, while task $T_{12}$ is two times slower but is only at 10% of its execution lifecycle, a PR based threshold would detect $T_{11}$ as a straggler due to its slower progress rate than $T_{12}$. However in reality, it is $T_{12}$ that will significantly impede total job completion time. This shortcoming inspires the timing based threshold.

- Estimated Completion Time (ECT) based threshold

$ECT_{ji}^t$ is calculated following Equation 2.8.

$$ECT_{ji}^t = t + \frac{1 - PS_{ji}^t}{PS_{ji}^t}(t - t_0^{T_{ji}}) \tag{2.8}$$

This type of threshold focuses on the actual remaining time and performs better in improving final response, and is the most commonly used threshold type. For example, the LATE speculator [162] uses it and achieves an improvement by a factor of two compared with Hadoop V1, and Mantri [5], another popular method adopts this type of threshold, gets a further 32% improvement in completion time.

To note that, within the scope of this thesis, the notion $ECT_j^t$ and $D_{J_j}$ are different in meaning. The former is the estimated completion of job $J_j$ at time $t$, while the latter is the actual duration of $J_j$ when it finishes. Similarly, $ECT_{ji}^t$ and $D_{T_{ji}}$ are not the same as well. Within the MapReduce model, $D_{J_j} = \max_{T_{ji} \in J_j}\{D_{T_{ji}}\}$, and the notion $\overline{D_{J_j}}$ used in the thesis represents the average duration of all $T_{ji}$s belong to $J_j$.

- Other thresholds

There are some other types of threshold used in specific systems out there, for example, the Fuxi scheduling system in AliCloud [166] adopts a threshold of Degree of Straggler Index (DoS-Index) to measure the relative speed of data processing, defined in Equation 3.1; Wrangler [159] uses $nd\left(T_{ji}\right)$ defined as the ratio of task execution time to the amount of work done (bytes read/written) by task $T_{ji}$, and etc.

There are some methods follow another way when identifying stragglers to avoid the time wasted on waiting in the "wait, identify, react" methodology. Instead of setting a straggler threshold, they try to set a proportion threshold for speculation, either from the beginning along with other normal tasks or throughout the whole job execution period. For example, the work in [6] adopts a full cloning methodology that launches two copies of each task regardless of the execution status, which wasted a lot of resources on non-straggler tasks. D. Wang, etc analyzed a resource performance tradeoff for setting this type of proportion threshold in the work of [151]. Hadoop YARN speculator sets a speculator number threshold, indicating a methodology that, as long as this number of upper bound limit is not met, the system will always pick up a current task with the longest ECT to do the speculation.

Besides the naive speculation policy, there are huge numbers of variation algorithms been proposed over the past few years based on the speculative scheme [77]. Representative works include LATE [162], Mantri [5], MCP [38], Coworker [69], Bobtail [157], CREST [81], eSplash [153], Wrangler [159], Hopper [122], Grass [7], and etc.

LATE [162] is the most popular speculation-based method that targets at heterogeneous environments. It is the first literature that proposes the concept of the *Longest Approximate Time to End (LATE)*, and uses it as the metrics when measuring stragglers. In addition, to handle the fact that speculative tasks cost resources, the authors augment the algorithm with two heuristics: (1) a cap on the number of speculative tasks that can be running at once, which is denoted as the *SpeculativeCap*, and (2) a *SlowTaskThreshold* that determines whether a task is "slow enough" to be speculated upon, which prevents needless speculation when only fast tasks are running.

Mantri [5] is another influential variation method based on the speculative scheme since LATE, and it monitors tasks and culls outliers based on their causes. Mantri performs intelligent restarting of outliers: a task that runs for long because of the imbalanced work will not be restarted. And if a task lags due to reading data over a low-bandwidth path, it will be restarted only if a more advantageous network location becomes available. In addition, Mantri protects against data loss induced re-computation through a cost-benefit analysis: upon a task's completion, Mantri replicates its output if the benefit of not having to recompute outweighs the cost of replication. Following these three main strategies: restarting outliers cognizant of work imbalances, network-aware placement of tasks and protecting outputs of valuable tasks, Mantri runs live in all of Bing's production clusters and achieved a very good result.

The MCP algorithm [38] improves speculation through the usage of both PR and process bandwidth within a phase (phase here refers to the sub-phases within Map and Reduce procedures) to select slow tasks, based on the observation that the time duration ratio of phases varies a lot in different types of jobs and environments. In addition, this smart speculation strategy uses exponentially weighted moving average to predict process speed and calculate a task's remaining time, and determines which task to backup based on the load of a cluster using a cost-benefit model, with the cost to be the computing resources occupied by tasks, and the performance to be the shortening of job execution time and the increase of the cluster throughput.

Similarly, the SAMR algorithm [40] also improves the progress prediction through fine-grained phase level tunning. It adjusts the time weight of each stage of Map and Reduce tasks according to the historical information, and updates the stored information on every node after every execution. ESAMR [137] is the enhanced version of SAMR that takes consideration of not only hardware heterogeneity, but also different job types and sizes, which also affect stage weights.

Coworker [69] proposes an idea of using coworkers to help with stragglers. In traditional speculation, duplicate tasks are launched on other nodes to process the same data as the stragglers. Any completion of these copies implies the finish of the task, and the other duplicate attempt can then be aborted. However, aborting task misspends resources. Coworker is proposed to solve this problem. Similar to speculation, a coworker executes on another node, however, instead of processing identical data, the coworker parcels out data between itself and the straggler. In other words, the coworker cooperates with the straggler to finish the task rather than compete with it. The authors also design a heuristic method to find out the most favorable data size for the coworker to achieve the shortest completion time according to the PR of the straggler and the coworker. Through this way, not only resource misspending is solved because no tasks are aborted, but also a shorter completion time than the original speculation can be achieved because no work is wasted.

Bobtail [157] focuses on Cloud environments and cares about the characteristic of virtualization. In the paper, the authors claim that virtualization used in platforms such as

Amazon Elastic Compute Cloud (EC2) exacerbates the long tail problem by factors of two to four compared to those observed in dedicated centers. And one big reason behind this phenomenon is the co-scheduling of CPU-bound and latency-sensitive tasks. While sharing is inevitable in multi-tenant Cloud computing, the Bobtail system is designed to proactively detects and avoids these bad neighboring Virtual Machine (VM)s without significantly penalizing node instantiation. Cloud customers can use Bobtail as a utility library to decide on which instance to run their latency-sensitive workload.

CREST [81] is also an algorithm that designed for Cloud environments rather than local clusters which assume insufficient bandwidth. Most of the methods, such as LATE, implicitly assume that the time cost for data movement on launching speculative task is trivial, which does not always stand for the virtualized Hadoop clusters in campus Clouds. In this paper, the authors propose a combination re-execution scheduling technology (CREST), which can achieve the optimal running time for speculative Map tasks and decrease the response time of MapReduce jobs. The main idea is that, re-executing a combination of tasks on a group of computing nodes may progress faster than directly speculating the straggler task on a random target node due to data locality.

There are other works that also take the target node into consideration when doing speculation. For example, eSplash [153] focuses on the identification of straggler nodes and effectively launch speculative tasks through avoiding such nodes. In the algorithm design, a performance vector is maintained by each node. The dimension of the vector is the number of distinct type of tasks this node has ever finished, and each value in the vector is the execution time of each type of tasks. The scheduler clusters all the nodes into multiple groups by using *k*-means. The system then detects the abnormally slow node by comparing the task performance on the node with the statistic data for the group it belongs to, and submit the speculative request with parameters that guide the future execution.

Wrangler [159] is a proactive method which avoids situations that cause stragglers. The initial idea is first proposed in work [158], which performs regression using the node-level statistics such as CPU/memory utilization to predict the task execution time, and using a decision tree based approach to generate interpretable rules that the cluster scheduler can easily use. These rules can guide the scheduler to a task assignment that avoids or minimizes the number of stragglers. Wrangler further improves itself in aspects of training times, and automate the whole learning process. Furthermore in [160], the enhanced Wrangler introduces a notion of a *confidence measure* to overcome the modeling error problems. This confidence measure is then exploited to achieve a reliable task scheduling with all the predictions toward situations that may cause stragglers.

There are some algorithms that function through enhanced scheduling that bearing speculation in mind. For example, Hopper [122] is a method that design the cluster scheduler through the coordination of scheduling and speculation, based on the fact that scheduling a speculative copy of a task has a direct impact on the resources available for other jobs. The key insight behind Hopper is that, a scheduler must anticipate the speculation requirements of jobs and dynamically allocate capacity depending on the marginal value of extra slots in terms of performance, which are likely to be used for speculation. One advantage of the Hopper scheduler is that it is compatible with all current speculation algorithms.

Grass [7] is an algorithm that targets at a special type of application: the approximation jobs. In big data analytics, timely results, even if based on only part of the data, are often important. For this reason, approximation jobs that require only a subset of their tasks to complete, are projected to dominate big data workloads. This type of jobs normally have either a deadline bound or an error bound, and the key idea of Grass is to dynamically prioritize tasks based on the deadline/error-bound while choosing between speculative copies for stragglers and unscheduled tasks, delicately balances immediacy of improving the approximation goal with the long-term implications of using extra resources for speculation. Other research that also targets at approximate jobs include [68] and [76].

Each of the related work has its own characteristics in terms of suitable target environment or straggler type. Stragglers can occur due to many reasons, not only from the application itself (such as a badly designed program that easily leads to unbalanced workload), but also from the execution environment perspective (for example, a disk fault, or a CPU over-heating). The next section focuses on introducing the literature that explores those reasons.

### 2.4.2 Straggler Root Causes

Considering the cause of straggling tasks, [38] categorizes them into internal and external reasons under the MapReduce background. Internal causes are the ones that can be solved by the MapReduce service provider, such as the block size configuration in the fork procedure, slot number parameter, and etc., while external reasons are the ones that cannot and are more relied on user behavior and system environment.

For internal reasons, [73] refines MapReduce jobs into (i) Map-only jobs, (ii) Map-mostly

Table 2.2: Straggler reasons (external) and corresponding meanings

| | |
|---|---|
| Shared Resources | Machines might be shared by different applications contending for shared resources, such as CPU cores, memory bandwidth, and network bandwidth |
| Global Resource | Applications running on different machines might contend for global resources such as network switches and shared file systems |
| Daemons | Background daemons may also compete for resources and cause hiccups |
| Maintenance Activities | Background activities such as data reconstruction in distributed file systems, periodic log compactions in storage systems |
| Queuing | Multiple layers of queuing in intermediate servers and network switches |
| Power Limits | Modern CPUs are designed to mitigating thermal effects by throttling if it run above average standard for a long period |
| Garbage Collection | The need for solid-state storage devices to periodically collects garbage data blocks |

jobs, and (iii) Reduce-mostly jobs, and determines which Hadoop specific parameters have the most influence on each kind of job's completion time by using one-way Analysis of Variance (ANOVA). The initial results from [73] are: for Map-only and Map-mostly jobs, the number of Map tasks launched and the amount of data read to HDFS are the key factors to final completion time, while for Reduce-mostly jobs, the completion time are primarily influenced by the number of Reduce tasks launched. There are works that pay particular attention to specific Hadoop parameters, for example, [154] finds out the influence bought by the *slot number* parameter (also refined as Map slot and Reduce slot), tests several slot assignment policies regarding different workload types. By default, the slot numbers are static, configured manually before the system starts running. Normally, the Map and Reduce slots are 1-2 and 2-4 times the number of the CPU cores respectively [167]. The work in [156] resolves this problem caused by fixed slot number though dynamically adjusting this number during task assignment based on CPU utilization of each node.

For external reasons, or in other words, the reasons that apply not only to MapReduce jobs, [45] lists seven possible candidates that could lead to the long tail problem, among them are shared resources, global resource sharing, background daemons, maintenance activity, queuing, power limits and garbage collection. Detailed meanings of those reasons are explained in Table 2.2.

There are works conduct the straggler cause analysis within other environments rather than clusters. For example, [157] claims that, for Internet-scale applications deployed on commercial Clouds such as Amazon EC2, in which virtualization is a key characteristic when providing multi-tenancy with some degree of isolation, the straggler phenomenon is more of a property of nodes rather than topology or network traffic. In addition, it also emphasizes that, this property of node is both pervasive through EC2 and persistent over time. Virtualization adds randomness to the general task execution, therefore in theory, it would enlarge straggler occurrence and severity. DeTail [165] targets at web applications, believing that packet drops, retransmission and the absence of flow prioritization are the main contributors to the web site's page creation time variation. As for the poor latency in high throughput services executing on multi-core machines, [82] explores the hardware sources such as the NUMA effect influence; the Operating System (OS) related sources such as the First In First Out (FIFO) scheduling and the interrupted processing; as well as the application level sources such as the task arrival distribution, and etc.

Except listing all possible reasons, there are a number of works focus on analyzing one specific type of straggler reason, exploring how that reason affect system behavior. The most representative branch is the research on skew caused stragglers. For example, in the work of [125][78][84], skews are categorized as either Map phase skew or Reduce phase skew. The factors that will contribute to each kind of skew type and how to avoid them are concluded in the paper as well. The next section details these skew mitigation methods.

### 2.4.3 Skew Mitigation

While the speculation-based works are shown to be effective in mitigating stragglers caused by reasons such as resource contention or hardware heterogeneity, they encounter unavoidable bottleneck when dealing with data skew caused stragglers: due to the duplication nature, the replica task processes identical input file with the skew type of straggler will still suffer from the uneven input distribution. It is shown that, a lot of stragglers in MapReduce framework are caused by the curse of skew: the Zipf distribution of the input or intermediate data [84]. In order to alleviate this bottleneck, MCP [38] and Mantri [5] improve speculation by deliberately avoid creating task copies for skew caused stragglers. However, while resources are saved from needless speculation, these avoidance-base methods do not mitigate the skew at all, and the job response time would still be influenced by the skew stragglers.

For skew handling approaches, coworker [69] functions in a way that as long as a straggler is identified, the reserved co-worker task will help process the remaining data. Its effectiveness is dependent on the choice of the reserved co-worker number, and introduces resource overhead when there is no skew. SkewTune [80] is another popular skew mitigation method that works through re-partition. As long as there is a free slot within the system, the task with the greatest remaining time will be re-partitioned. However, the Reduce outputs of both these two methods have to be reconstructed due to the fact that the MapReduce requires all tuples sharing the same key to be dispatched to the same Reducer, and this reconstruction introduces additional complexity.

There are methods rely on node performance when dealing with skews. For example, the work detailed in [101] splits the cluster into two groups depending on machine processing capacity. The intermediate data number per Reducer is counted by the system. As long as the number for a certain Reducer surpasses a pre-defined threshold, this Reducer will be assigned to the quick node group for execution. There are some shortcomings of this approach, for example, the threshold to decide the skew level differs with workloads, and the coarse grain node classification is insufficient for effective skew mitigation.

The mainstream method for Reduce skew mitigation focuses on optimized partition approaches to distribute the intermediate keys to Reducers. Hash and range are two of the most popular partition methods. Hash partition is relatively straightforward, requires only the Reducer number to generate the $< intermediateKey, Reducer >$ mapping decision through hash calculation, while range partition requires the developer to know the data distribution, therefore needs sampling. LIBRA [39] is the representative work of this type. It first launches selected sample Map tasks to estimate the intermediate data distribution for partition decision making before the real execution. However, the efficiency of this method is largely dependent on the estimation accuracy, which varies with different sampling strategies and sample size selections.

### 2.4.4 Gaps in the literature

Despite the large number of related literatures in straggler mitigation, straggler reason analysis, and skew mitigation, there still leave some space for improvement. For example, for the straggler reason research, the stragglers caused by the MapReduce internal reasons are relatively well explored from various angles, however, the analysis of the external causes are quite limited. Most literatures are simply introducing possible reason candidates, as for how exactly each one of them affects system behavior has never been discussed. In addition, when stragglers are observed within a system, there is a lack of work that explores the exact reason that causing those specific stragglers, and this limits the choice of the most suitable methods to be undertaken. For straggler identification, there are not many literatures that are resource-aware, and most of the current methods imply static straggler threshold, which is not flexible in face of the changing environment. A brief summary of the characteristics and shortcomings of the representative literature is illustrated in Table 2.3.

Table 2.3: Representative straggler mitigation approaches

| Methods | Naive Spec [155] | LATE [162] | Mantri [5] | Dolly [6] | SkewTune [80] | CREST [81] | Grass [7] |
|---|---|---|---|---|---|---|---|
| Metrics | PS | ECT | $t_{rem}$, $t_{new}$ | Cloning | $t_{rem}$, $t_{par}$ | PR | ECT |
| Target Type | – | – | – | Small Jobs | – | – | Approx Jobs |
| Node Hetero | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ |
| Dy- NP | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ |
| Dy- Thresh | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| Extra Reso- | ✗ | ✗ | ✓ | 5% | ✓ | ✗ | ✓ |
| Spec Cap | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ |
| Data Skew | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ |
| Bench mark | WC, Sort | WC, Sort | WC, Hive | – | II, PageR | GSA | Hive, Scope |

Some abbreviations in the table are: "*Hetero*", Heterogeneity; "*Dy-*", Dynamic; "*Thresh*", Threshold; "*NP*", Node Performance; "*Selec*", Selection; "*Spec*", Speculation; "*Reso-*", Resource; "*Approx*", Approximate; "*II*", Inverted Index; "*WC*", WordCount; "*PageR*", PageRank and "*GSA*", Genome Sequence Analysis. The "*Metrics*" represents the speculation meterics used when identifying stragglers. The "–" in the target row indicates that, the corresponding method is designed for general type of applications, while in the benchmark row, this indicates that the exact benchmark used is not given (for Dolly, the workload from Facebook and Microsoft Bing are used according to the paper, however, details of the exact benchmark is not exposed).

## 2.5 Summary

Computing systems experienced an unprecedented evolution in the past decades, and Cloud computing had now become an indispensable part of information technology to satisfy the increasing demands for Internet services such as web search, social networking, and machine learning applications. A Cloud datacenter typically consists of hundreds / thousands of heterogeneous machine nodes to provide reliable computing and storage services to customers, in which tasks are executed on multiple server nodes by systems that automatically provide scheduling, fault tolerance, and load balancing. The MapReduce framework pioneered this computing model, and systems like Hadoop YARN and Spark generalized its population. Through virtualization, multiple tenants are enabled to share the cluster resources and services. The exhibited heterogeneity of workload characteristics such as task scale, execution time and resource usage pattern has raised new challenges in terms of performance interference, resource utilization, power consumption, system resilience, etc.

Among the many challenges, the straggler problem is a representative issue that hinders parallel job execution performance as well as system availability, leading to potential Quality of Service (QoS) degradation and the late-timing failure. Various straggler detection and mitigation approaches are discussed, such as simple cloning, blacklisting, and speculation. Among them, speculative execution is the dominant method which functions in a three-phase manner: it firstly identifies task stragglers, then launches redundant task copy for an identified straggler, and finally adopts whichever result that comes out rst. Each of the related work has its own characteristic in terms of suitable target environments or straggler types, and face its own shortcomings, all of which are summarised in this chapter.

In the following research, stragglers are intensively discussed under the MapReduce framework. There are mainly two reasons why this research focuses on MapReduce: firstly, it is a representative parallel computing model, especially under the big data background; secondly, most (more than 90%) of the related literature about the straggler research is focusing on this framework, therefore in order to compare with the state-of-the-art methods, the proposed algorithms are built based on this concept. There exists a dependency in MapReduce jobs that the Reduce tasks have to wait for Map tasks to finish before they can start. This is a very important difference between a MapReduce job and a non-MapReduce job. The current assumption adopted by the following research does not consider this dependency, which means that, the methodology of the proposed approach can be applied to general systems. Hadoop YARN is a case study when implementing the approach into practical platforms in this thesis. Although focusing on MapReduce, the approach is a universal idea, and there are some general discussions when analyzing the straggler behavior in real-world Cloud datacenters where mixed workload type co-exists. Details are discussed in the next chapter.

# Chapter 3

# Quantitative Analysis of the Stragglers

The severe consequences brought by stragglers toward efficient parallel job execution, the complicated reasons for straggler occurrence, and the low effectiveness of state-of-the-art straggler mitigation techniques are the main motivations of this research. This section focuses on analyzing the straggler-related behaviors in a quantitative way using real-world cluster tracelog data, including straggler influence analysis, straggler reason analysis, and speculation limitation analysis, showing the importance and the urgency of this research. The overall system model of the proposed intelligent straggler mitigation system is outlined in the end.

## 3.1 Data Set Introduction

Three real-world cluster datasets have been used to conduct the analysis, including (1) the Google cluster tracelog, (2) the AliCloud cluster tracelog, and (3) the OpenCloud Hadoop cluster tracelog. As these operational trace data are semi-structured and voluminous - composed of multiple files detailing information concerning task resource usage, event logs, and server utilization - it is necessary to filter the trace data within each system in order to identify different job types. Specifically, within each data source, we are particularly interested in certain jobs which fulfill specific criteria, such as the batch jobs (i.e. DAG, MPI) and the MapReduce jobs. The respective tasks of these job of interest are filtered for the following analysis.

### 3.1.1 The Google Dataset

For the Google data, it comprises 29 days detailing job / task behaviors on 12,532 server nodes that share a common cluster management system, and can be downloaded from [145]. Work arrives at this cluster in the form of jobs that comprise several tasks, and a task is represented as a Linux program that executes on a single node. The cluster contains numerous application types including batch, latency sensitive, system monitoring, and MapReduce. However, due to commercial confidentiality, Google does not reveal the precise information that directly points out what type of workload a specific job belongs to. The whole dataset consists of mainly 6 tables covering server and workload (task and user) attributes, among which, ***Job_events*** table describes event records for job submissions and completions, and ***Task_events*** table records task submissions / completions.

The raw dataset is voluminous, approximately 400GB in size when unzipped, and contains traces of 672,074 jobs composed of 25,228,174 tasks. Therefore it is important to filter out noisy information and properly decide suitable target jobs in order to conduct further analysis. In our analysis, we design a set of filters and extract a batch job subset. Batch jobs are possible to derive given the characteristics of the job priority (in the Google definition [120], this equals 4, which represents production jobs), job start and completion time in relation to task submission and completion (information pertaining to ownership of tasks is identified through the use of recorded *jobID* attached to all submitted tasks), as well as resource characteristics of tasks (i.e. all tasks within a job have the same requested resources and are submitted at the same timestamp with each other).

Stragglers in the following research refer to tasks that perform slowly, but not failed tasks. Therefore, to further decrease the target data size, another important assumption adopted is to ignore tasks that are been killed through the erroneous events recorded in the *Job_events* table. After filtering, we managed to identify 3,362 batch jobs comprising of 282,950 tasks.

Servers in the Google cluster vary in terms of physical capacity (mainly memory RAM

Table 3.1: Server proportions and properties within the Google system

| Platform ID | Server Type | CPU Capacity | Memory Capacity | Proportions (%) |
|---|---|---|---|---|
| A | 1 | 0.25 | 0.25 | 0.99 |
| B | 2 | 1.00 | 1.00 | 6.34 |
| | 3 | 1.00 | 0.50 | 0.018 |
| C | 4 | 0.50 | 0.25 | 30.76 |
| | 5 | 0.50 | 0.75 | 7.93 |
| | 6 | 0.50 | 0.50 | 53.46 |
| | 7 | 0.50 | 0.97 | 0.04 |
| | 8 | 0.50 | 0.12 | 0.43 |
| | 9 | 0.50 | 0.03 | 0.024 |
| | 10 | 0.50 | 0.06 | 0.008 |

size and CPU cores) and platform type - a combination of microarchitectures and memory technologies that result in different clock rates and memory speeds. The combination of unique CPU capacity, memory capacity, and platform for a server result in a unique server architecture type. There are a total of 10 unique server architectures within the trace log, however, 6 of them account less than 1% of the total server population. Therefore, we postulate that these servers are only reserved for tasks which have specific constraints on the hardware architecture, and we ignore them in our analysis. The detailed information of each server type is shown in Table 3.1, with the shaded type representing the majority.

### 3.1.2 The AliCloud Dataset

Alibaba runs the biggest e-commerce website in China, and AliCloud is the large-scale Cloud provider supports the e-commerce service and many other services [4]. The AliCloud data we use in our analysis is a private dataset the company handed over to us under a joint project, which is not publicly accessible due to commercial reasons.

The attributes provided in the data include *JobID, TaskID, StartTimeOnWorker, EndTimeOnWorker, PID, IP,* and *Port*. The TaskID is named in a style that comprises JobID so that the task-job relationship can be understood. Table 3.2 gives one record from the AliCloud table as an example, revealing the execution detail of the $45^{th}$ subtask from job 321.

Table 3.2: AliCloud data structure

| JobID | TaskID | StartTimeOnWorker | EndTimeOnWorker | PID | IP | Port |
|---|---|---|---|---|---|---|
| 321 | 321_45 | 1386696383 | 1386696399 | 70346 | 10.138.0.144 | 44346 |

The start and the end time are represented in Unix timestamp, in this example, the task started on 17:26:23, 10th December 2013, Chinese time, and ended 16 seconds later. In total, there are 875 jobs comprised of altogether 1,233,879 tasks in the provided trace. We do not hold any machine node related information apart from the IP attribute, such as the physical capacity or resource utilization.

### 3.1.3 The OpenCloud Dataset

OpenCloud is a research cluster at the Carnegie Mellon University [103] that consists of 116 machine nodes running the Hadoop platform. The cluster supports research activities for different departments within the University. OpenCloud releases its task execution tracelog for public research covering the first 9 months in 2012. There are 6 tables provided, from which the *task_attempt_history* table contains the information of interest such as *jobID*, *tasktype*, *taskID*, *start / shuffle / sort / finish time* of the task attempt (represented as UTC timestamp in milliseconds), *status* (success, failed or killed), and *hostname*.

After filtering, 18,935 successful parallel jobs consist of 8,734,974 tasks are analyzed. The machine nodes within this cluster are homogeneous in physical configuration [103]. Each of them has a 2.8 GHz dual quad core CPU (8 cores), 16 GB RAM, 10 Gbps Ethernet NIC, and four Seagate 7200 RPM SATA disk drives.

To briefly summarise, general information of each above dataset, including the cluster size, server heterogeneity, and job patterns, are given in Table 3.3.

Table 3.3: General data pattern summary

| System | Google | AliCloud | OpenCloud |
|---|---|---|---|
| Cluster size | 12,532 | 2,841 | 116 |
| Time period | 29 days | 14 days | 9 months |
| Server types | Heterogeneous | Not given | Homogeneous |
| Job Number | 3,362 | 875 | 18,935 |
| Task Number | 282,950 | 1,233,879 | 8,734,974 |
| Maximum Job Size | 9,999 | 9320 | 21,383 |
| Average Job Size | 84 | 1962 | 534 |
| Medium Job Size | 32 | 552 | 165 |
| Maximum Job Duration | 65,906 s | 356 s | 99,246 s |
| Minimum Job Duration | 56s | 1 s | 1 s |
| Average Job Duration | 1779 s | 47 s | 335 s |
| Medium Job Duration | 583 s | 32 s | 16 s |

## 3.2 Straggler Related Statistics

The above datasets make it possible for us to explore straggler related issues within the context of real system operation through statistical analysis. This includes the research of studying the frequency of task straggler occurrence and the impact stragglers impose on parallel job performance within large-scale computing systems.

### 3.2.1 Task-Level Statistics

Here in the analysis, we define stragglers to those parallel tasks that have an execution time longer than 150% of job average/medium duration, which is consistent with many popular straggler identification technique such as [162][6][80][125] introduced in Section 2.4.1. From the data, the grouping of tasks to a specific job can be established according to the *jobID* value attached. Their execution time is calculated through recorded start and completion events within the trace, and once the duration for all tasks has been determined, the difference between an individual task's duration and the average/median duration of all tasks within a job can be calculated.

Figure 3.1, Figure 3.2, and Figure 3.3 show the difference between an individual task's execution duration and the mean and median execution of all tasks within the same job in the Google system, the AliCloud system, and the OpenCloud system, respectively. From the figure it is observable that, calculating the difference using different central tendency measurements of mean and median results in substantially different patterns for straggler detection. This is particularly noticeable within AliCloud as shown in Figure 3.2 (a) and (b), which exhibit different dispersion patterns for task execution. This is resultant of extremely fast or slow tasks affecting the central tendency and the dispersion for task completion when using the mean, while the median duration within a job is less affected by extreme execution times of task stragglers.

Regardless of the comparison standard, it is observable that, within all three studied systems, the majority of tasks exhibit similar proportions for completion situated around 100%, a number indicating the fact that the duration of the individual task equals (or similar) to the average/median job duration. Meanwhile, a trivial portion of straggler tasks can also be observed within all three figures, characterized by a much longer duration compared with job average/median, larger than 150%. Detailed task straggler proportions for each cluster are listed in Table 3.4, all featuring a number less than 10% except the

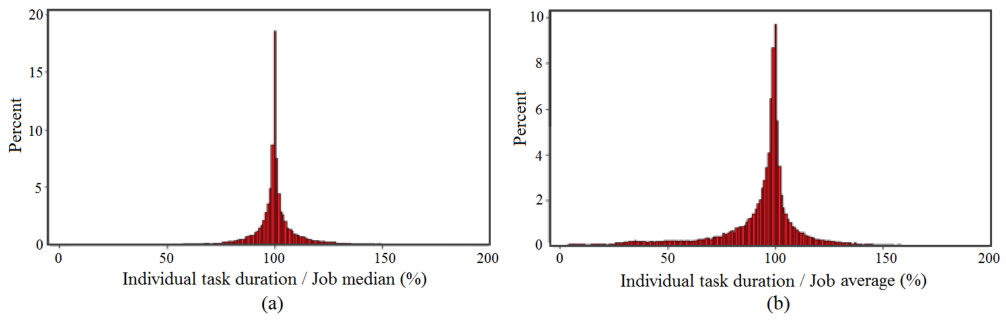


Figure 3.1: Google task-job (a) median completion histogram; task-job (b) mean completion histogram

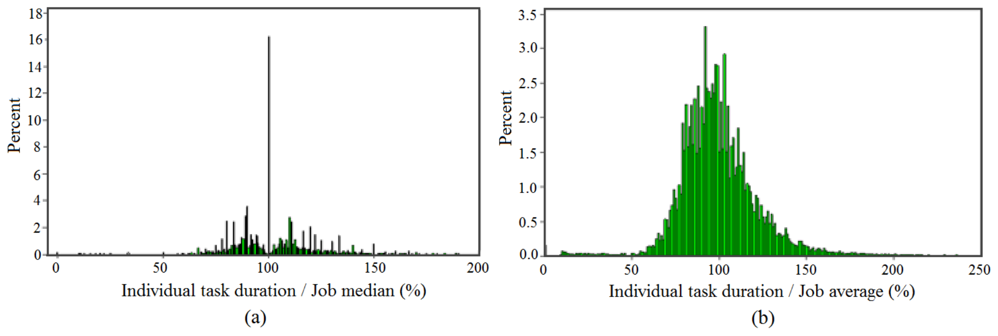


Figure 3.2: AliCloud task-job (a) median completion histogram; task-job (b) mean completion histogram

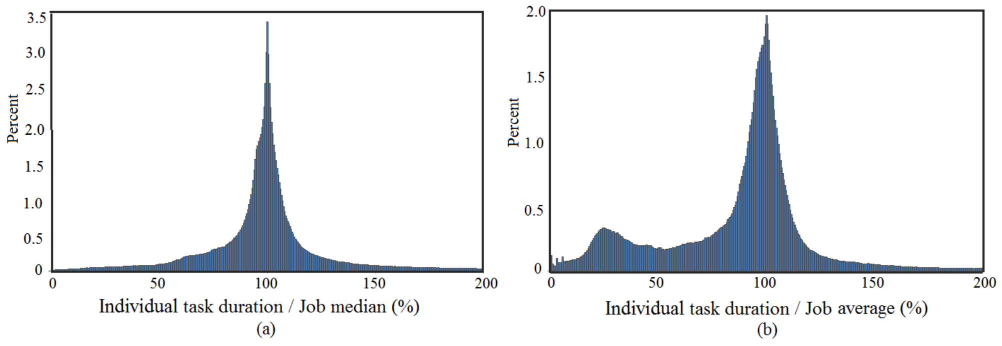


Figure 3.3: OpenCloud task-job (a) median completion histogram; task-job (b) mean completion histogram

OpenCloud cluster when using median duration as the comparison standard.

Despite the small proportion, the additional time wasted by the stragglers are huge. The figures only show partial of the entire distribution within 200% / 250% scale in x-axis in order to make the histogram neat, however, the actual maximum value for the straggler tasks take more than 100 times mean/median duration in all three systems. This observation corroborates to the findings revealed in [5], demonstrating the fact that jobs can be delayed by up to more than 1000%.

### 3.2.2 Job-Level Statistics

Besides the very expensive stragglers, due to the large size of current parallelization, the total number of affected jobs are huge as well.

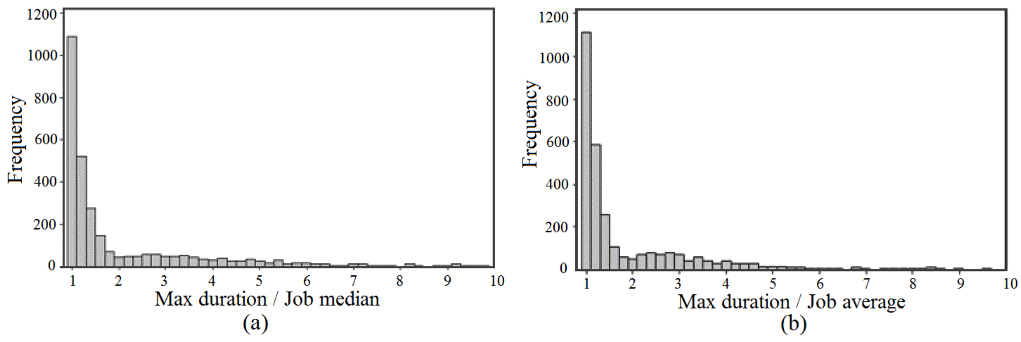


Figure 3.4: Google job tailing extent compared with job (a) median, (b) mean

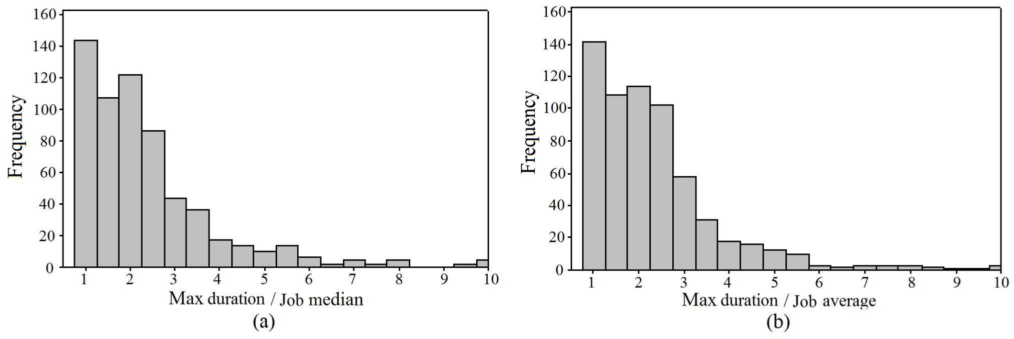


Figure 3.5: AliCloud job tailing extent compared with job (a) median, (b) mean

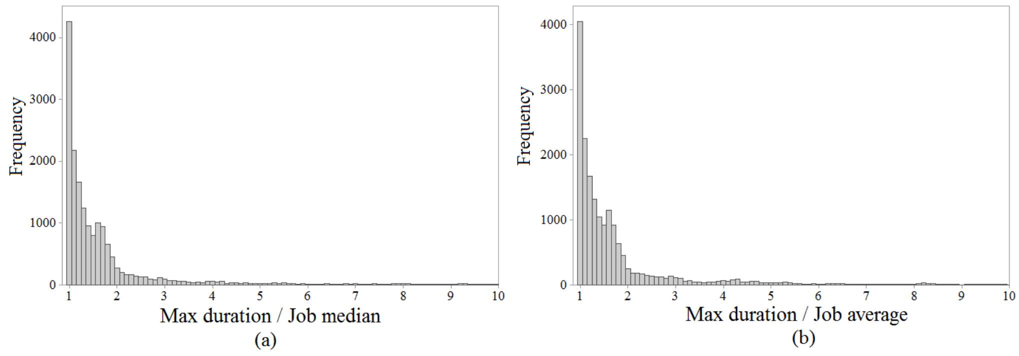


Figure 3.6: OpenCloud job tailing extent compared with (a) median, (b) mean

We define *tailing jobs* as the ones that contain task stragglers and exhibit a tailing execution shape. The tailing extent is calculated as the maximum duration within a job divided by job median / average duration. Findings demonstrated in Figure 3.4, Figure 3.5, and Figure 3.6 show the frequencies of tailing jobs in different extent. It is observable that,

Table 3.4: Straggler occurrence and impact on production systems

| | Google Datacenter | | AliCloud Datacenter | | OpenCloud Cluster | |
|---|---|---|---|---|---|---|
| | Mean | Median | Mean | Median | Mean | Median |
| Total tasks | 282,950 | | 1,233,879 | | 8,734,974 | |
| Task stragglers | 11,210 | 16,543 | 33,322 | 42,925 | 603,973 | 1,067,103 |
| Task stragglers % | 3.96 | 5.85 | 2.70 | 3.48 | 6.91 | 12.22 |
| Total jobs | 3,362 | | 875 | | 18,935 | |
| Job stragglers | 1,280 | 1,351 | 512 | 433 | 8,151 | 8,224 |
| Job stragglers % | 38.07 | 40.18 | 58.51 | 49.49 | 43.05 | 43.43 |

large amount of jobs within all three clusters are negatively affected, between 38.07% and 58.07% in number, accounting for approximately half of the total population.

The degraded performance is due to a job's inability to complete unless its respective tasks, including stragglers, have all completed execution. Such behavior resonates with the theorized impact of stragglers in large-scale system discussed in [45]. The quantified numbers in terms of tailing job within the three target systems are shown in Table 3.4.

We also explore the relationship between job sizes and straggler possibilities, and this analysis has discovered a non-intuitive finding: the chance of jobs encountering straggler does not directly link with their size. It is observed from all three datasets that, no matter what size the job is, the possibility of the job containing stragglers is randomly distributed: the straggler rate observed in small parallel jobs consist only less than 10 tasks ranges between less than 1% to as high as almost 50%; and this is the same for large jobs with more than 10,000 tasks, with some large job contains no stragglers while some encounter more than 40%.

Besides the workload analysis, the environmental factor is another important perspective that affects straggler behaviors. The next section focuses on node level statistics.

### 3.2.3 Node-Level Statistics

When exploring the manifestation of task stragglers on servers, the studied results of straggler percent over different nodes observed in the Google and the AliCloud dataset are depicted in Figure 3.7(a) and (b), respectively. To note that, the *ranked machine ID* in the graph is not the actual ID marked in the system log, but a modified (or ranked) identifier to make the graph neat. The *straggler%* is calculated as the identified straggler number over the total number of tasks assigned on this specific node over the whole time the data covered. This is to eliminate the effects of unbalanced workloads.

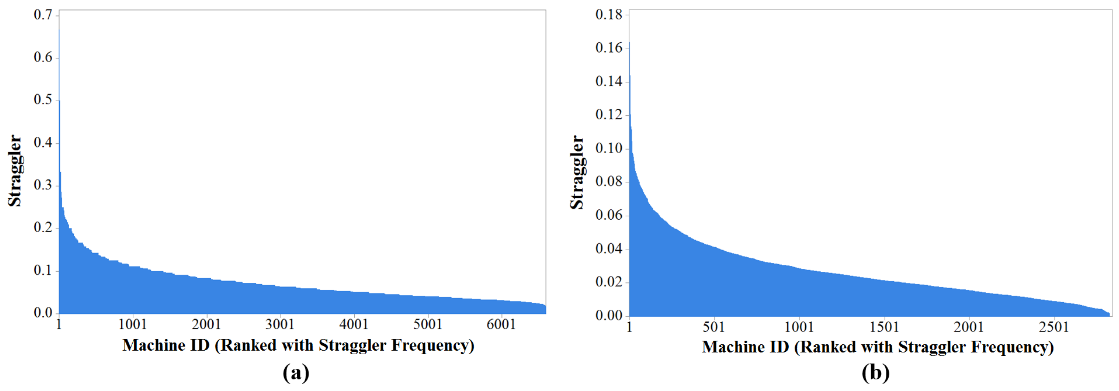


Figure 3.7: Straggler percentage per node in the (a) Google, and (b)AliCloud system

It is observable that, the straggler percentage over node distribution is slightly skewed.

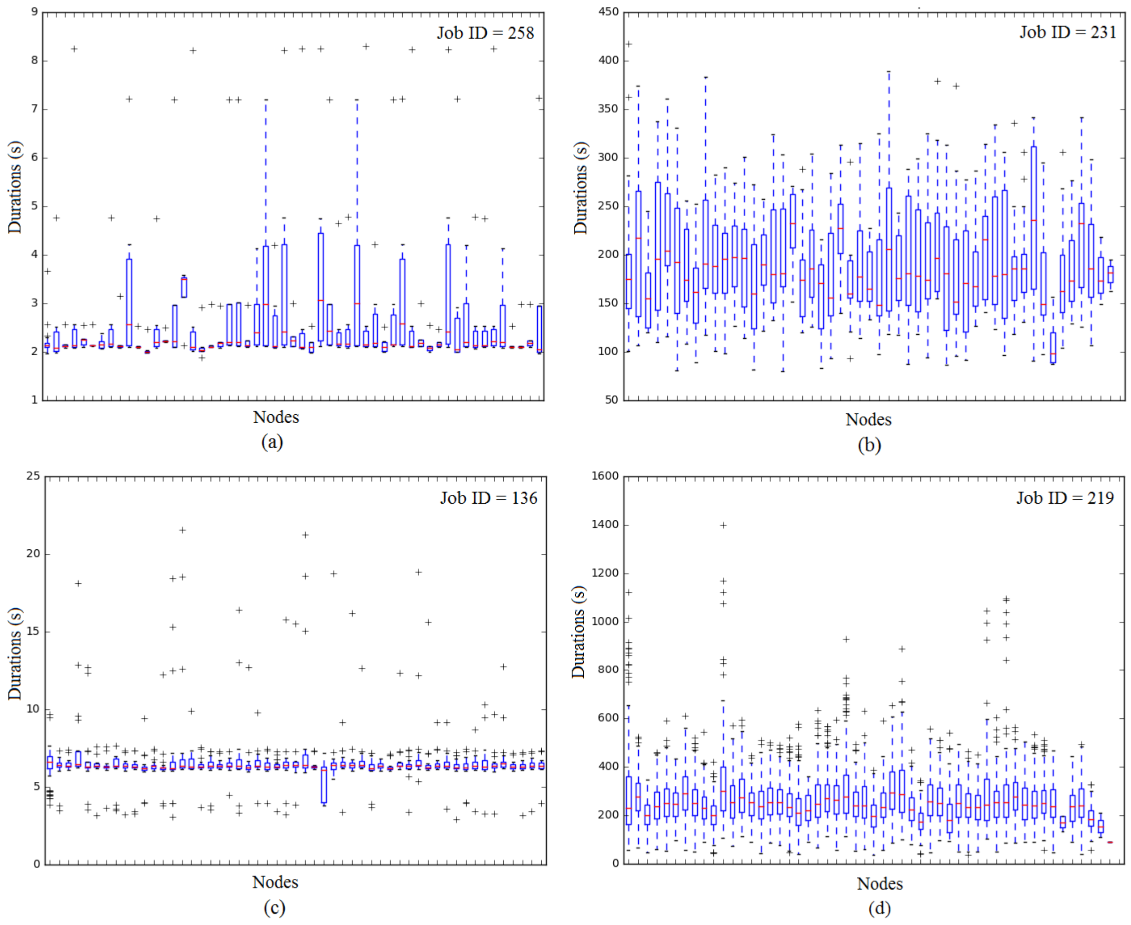


Figure 3.8: Map tasks execution on different machine nodes from four example MapReduce jobs within the OpenCloud cluster

For example, within the Google system, some machines encounter almost 70% straggler rate while the others have less than 10%. For a specific job, once its subtasks assigned on different nodes for execution, even with designed similar duration, the performance can still vary. Figure 3.8 illustrates this with four examples from the OpenCloud system.

Each subfigure represents a boxplot of Map tasks' durations of a single MapReduce job that is assigned to different machines. While similar execution time is expected within each job, the actual duration of each Map task varies, and this variation is related to the node performance. For example, in Figure 3.8 (a), there are a few nodes that exhibit an obvious longer average duration and a much larger variation in task completion. These weakly performed nodes generate non-negligible impact on parallel job completion time.

This performance heterogeneity stems from the dynamic operational situation and different aging condition. The straggler behavior can be exaggerated by the heterogeneous trend of clusters with different nodes consisting of CPU, GPU, TPU, FPGA, etc. Therefore, it is important to model and to predict the node execution performance when mitigating the stragglers. In the thesis, *node execution performance* is defined as the measurement of effective task execution within a node in the presence of stragglers. A server exhibits poor execution performance indicating a high task straggler occurrence possibility. To periodically blacklist this kind of slow nodes can help increase straggler mitigation efficiency and improve overall job execution.

Figure 3.9 (a) illustrates the straggler number per node distribution over the 9-month time in the OpenCloud system, with machine IDs in each sub-figure remaining the same. The blank machines in some sub-figures reflect the fact that not all nodes are in use for the whole time, some are only turned on in certain months. For example, nodes with $ID \in (80, 100]$ are used only in the $5^{th}$ month. It is observable that, for each month, there are some nodes experiencing much more stragglers than the others, labeled with circles in Figure 3.9 (a). Considering the homogeneous physical configuration of the OpenCloud cluster, this shows that, node performance is not purely dependent on their capacities.

Some related works such as [158] use resource utilization instead of physical capacity. However, the node performance diversity is not solely dependent on contention or utilization as well. Figure 3.9 (b) shows the total task number per node distribution. The number of tasks assigned is used to partially represent contention level of the node due to the lack of utilization data. It is observable that, during each month, the task number for each node is relatively even. The $7^{th}$ month is the only exception, with three obvious busier nodes (again labeled with circles). For the rest months, different straggler numbers are not due to contention. The node IDs in Figure 3.9 (b) are consistent through the 9 sub-graphs, same with Figure 3.9 (a).

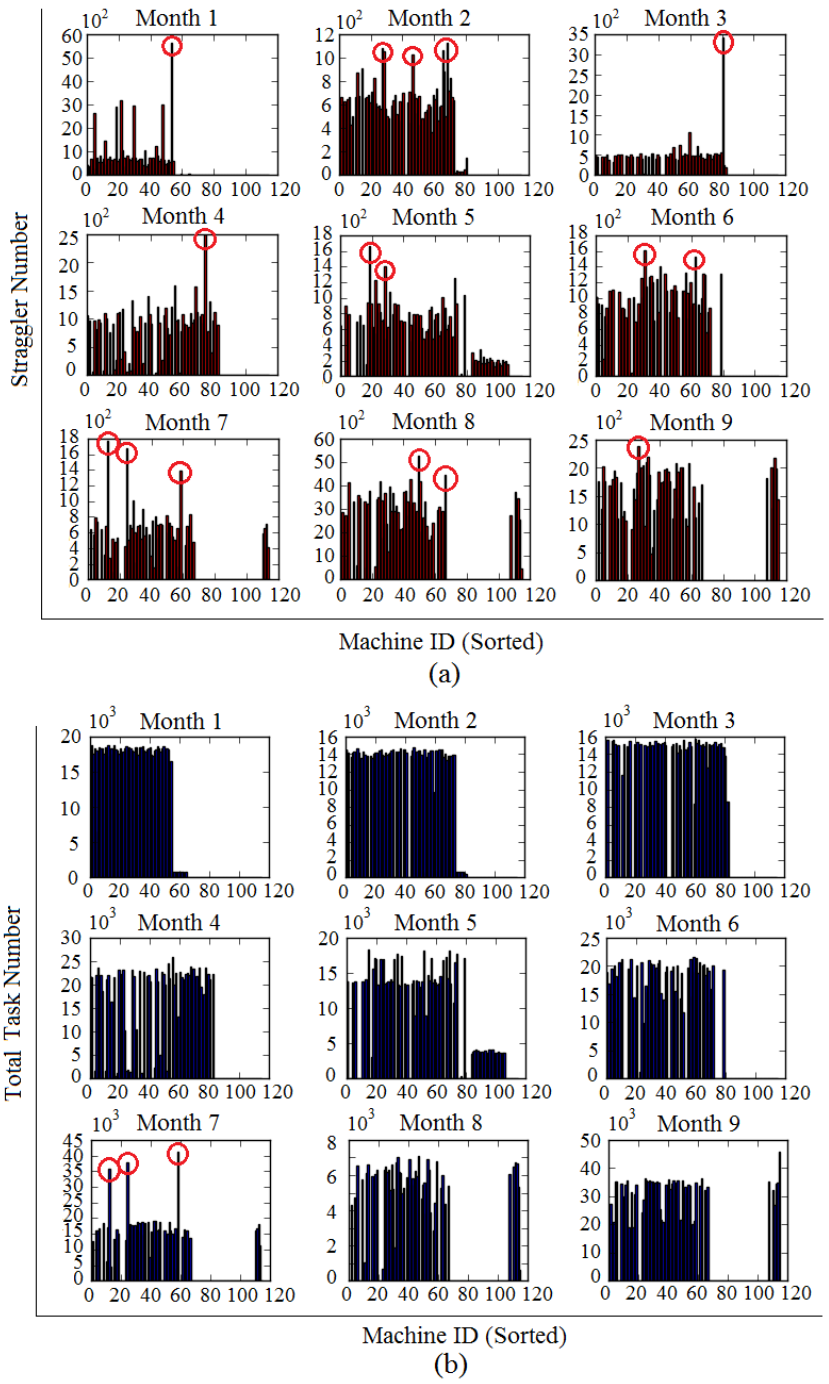


Figure 3.9: OpenCloud (a) straggler number per node distribution, (b) total task number per node distribution over the 9-month period

These straggler related statistic analysis within production datacenters attracts particular interest and emphasizes the importance of the straggler research. Specifically, while stragglers occur in less than 10% of total tasks submitted, they impact a greater proportion of jobs up to more than half of total number. In addition, stragglers are not restricted on a limited machine set, nor exhibit regular pattern on the time of occurrence. By demonstrating these effects stragglers impose, researchers and industry can then be able to convey the scale and importance of addressing straggler behavior to the wider community. In the next section, we investigate the underlying causes that produce these identified stragglers.

## 3.3 Straggler Reason Analysis

It is advantageous to understand the precise operational scenarios and causes that result in stragglers. This is important in order to focus technical and developmental efforts toward reducing future straggler occurrence within the system. This section details the straggler root-cause analysis we conducted based on the AliCloud data.

As presented in Section 2.4.2, stragglers stem from numerous reasons, and in order to derive a deep insight into the most important root cause, an investigation of correlation is conducted after straggler filtration. A periodic execution of a health checker process using Tsar [143] and Nagios [100] to monitor system metrics at a specific time interval is performed within the AliCloud system, with recorded key performance indicators such as CPU and memory utilization, network package loss rate, hardware faults, and etc. As our project partner, such monitoring data covering 20 days period is provided to us, and this forms the foundation for us to conduct the correlation analytics.

The AliCloud production system adopts a criterion for straggler detection termed as Degree of Straggler Index (DoS-Index) [166] instead of the Progress Score (PS) based or Estimated Completion Time (ECT) based threshold discussed in Section 2.4.1. DoS-Index is a system metric comprising task execution time and input size for an individual task $T_{ji}$ in job $J_j$ as shown in Equation 3.1 to eliminate the influence of imbalanced input size.

$$DoS_{Index} = \left( \frac{D^t_{T_{ji}}}{Inp(T_{ji})} \right) \div \left( \frac{\frac{1}{n}\sum_{i=1}^{n} D_{T^t_{ji}}}{\frac{1}{n}\sum_{i=1}^{n} Inp(T_{ji})} \right) \tag{3.1}$$

Within the equation, $D^t_{T_{ji}} = t - t_0^{T_{ji}}$ is the current execution duration of $T_{ji}$ at time $t$ ($t_0^{T_{ji}}$ is the start timestamp of $T_{ji}$), and $Inp(T_{ji})$ is the data volume that $T_{ji}$ is required to process. The DoS-Index indicates a relative speed of data processing, i.e., the time consumed when processing one unit of input data, for an individual task contrasted against the other tasks within the same job. Based on this definition, it is possible to control the strictness for straggler detection. The straggler threshold is configured as DoS-Index $\geq 10$ in our analysis to focus on the extreme stragglers due to their noticeable impact to user perception of application performance.

Table 3.5: Straggler detection with DoS-Index in AliCloud datacenter

| Day | DoS-Index$\geq 10$ | System Condition at Detection | | |
|---|---|---|---|---|
| | | $Util_{CPU} \geq 80\%$ | $Util_{Disk} \geq 80\%$ | Slow Req Handling |
| 1 | 127 | 46 | 61 | 29 |
| 2 | 213 | 114 | 14 | 23 |
| 3 | 161 | 161 | 84 | 35 |
| 4 | 147 | 147 | 23 | 43 |
| 5 | 453 | 158 | 149 | 69 |
| 6 | 215 | 129 | 184 | 71 |
| 7 | 352 | 352 | 128 | 82 |
| 8 | 363 | 348 | 129 | 75 |
| 9 | 253 | 121 | 94 | 98 |
| 10 | 267 | 116 | 77 | 233 |
| 11 | 241 | 100 | 150 | 132 |
| 12 | 254 | 179 | 239 | 168 |
| 13 | 247 | 126 | 247 | 161 |
| 14 | 267 | 117 | 41 | 125 |
| 15 | 259 | 104 | 259 | 85 |
| 16 | 699 | 131 | 66 | 138 |
| 17 | 510 | 236 | 92 | 67 |
| 18 | 227 | 163 | 63 | 142 |
| 19 | 326 | 172 | 83 | 58 |
| 20 | 279 | 154 | 101 | 27 |
| Total | 5,860 | 3,174(54.2%) | 2,284(39.0%) | 1,861(31.8%) |

Table 3.5 presents statistics of stragglers within the 20-day period. The profiling data we adopt in the analysis includes server CPU utilization $\geq 80\%$, Disk usage$\geq 80\%$, and slow read-write request handling (latency from file system $\geq 400ms$). It is observable that, approximately 54.2% and 39.0% of stragglers with DoS-Index $\geq 10$ occur under the presence of high server CPU and disk overloading, respectively, and it is also observed that 31.8% of stragglers experience slow request handling. This result indicates that high server resource utilization is an important cause for straggler occurrence. This is consistent with the insights provided in [159], stating that disk utilization (I/O) and memory

contention are the primary bottlenecks contributing to the creation of stragglers on a node.

To note that, condition overlapping is not considered in this analysis, explaining the reason why the sum of these three reasons sometimes exceeds 100% in Table 3.5: it is possible for CPU utilization and disk utilization to be correlated. In addition, there are some cases such as day 16 and 17 where the number of detected stragglers is larger than the sum of stragglers that caused by these three listed reasons. This reveals other more complicated situation when stragglers can occur, including network package loss, hardware faults, application errors, etc. Table 3.6 shows the categorization of the dominant factors that cause stragglers and their corresponding frequency.

Table 3.6: Classification for straggler root-cause

| Category | Specified Description | Freq |
|---|---|---|
| High CPU utilization | Low time-slice sharing and process scheduling due to certain bad user-defined worker logic, unbalanced workload aggregation etc. | 30% |
| High disk utilization | Local disk read and write conflicts, unbalanced tasks aggregation, disk faults etc. | 23% |
| Unhandled request | Distributed file system request surging(usually read request) and overpass the capability of request handling. | 23% |
| Network package loss | Network traffic package loss, resulting in repeating intermediate file and data transmission. | 13% |
| Hardware faults | Server timing-out, hang etc. | 7% |
| Data skew | Uneven file block input resulting in data skew. | 4% |

It is observable that high CPU utilization is the dominant type of cause. In production systems, this is often caused by two reasons: unbalanced workload aggregation and poor user code. Unbalanced aggregation is the result of excessive workload co-allocation within a server caused by inefficient scheduling. Poor user code refers to the inefficiently designed executable logic (e.g. orphan processes, looping conditions) compiled and executed by the user. Both of these reasons result in CPU bursting within a short time period; leading to inefficient time-slice sharing within the server kernel.

Another important straggler cause is the faults within a server node, specifically, this includes transient disk faults which result in slow disk I/O and file operations; and resource interference generated by co-locating tasks with the same resource characteristic (e.g. IO intensive) within the same machine. It is discovered within the AliCloud system that, it is possible for tasks to read and write to the same disk block simultaneously, resulting in large amounts of disk resource competition which requires conflict resolving.

The request handling inefficiency due to overloaded and surging file requests is another reason observed. Specifically, a typical MapReduce job generates a large number of operational requests including reads and writes to the distributed file system such as HDFS. Once the surging request number surpasses the handling capability of the file system master, it becomes a bottleneck. Even when the master has multiple replicas. Therefore, many requests ended up being queued, waiting to be allocated. Based on our analysis, it is observable that, in some cases, the unreasonable configuration of Map / Reduce number or block size can lead to unexpected request increase, thereby increases the load of file system master and causes slow request handling.

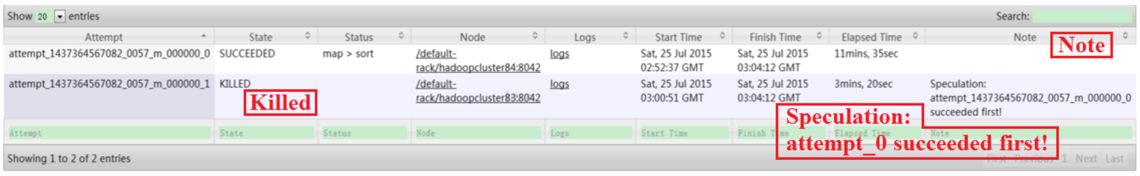
Furthermore, the network condition is also a variable that will affect reliable task execution, due to remote operations after the copy / shuffle phase in MapReduce are all being sent through the network. From our analysis after running "tsar retran" [143], 13% of stragglers were caused due to network package loss. Higher package re-transmission results in not only extended job end-to-end time-span, but also aggravates the network congestion. Finally, other common factors include time-out faults and data skew, comprising around 10% of straggler root-cause. This result could be used as inspirable instructions to handle with different stragglers, and can cover multiple scenarios and fault-injection practices to simulate the straggler behavior.

## 3.4 Speculation Limitation

As the dominant straggler mitigation scheme, speculative execution [46] is commonly deployed in industrial clusters such as Facebook, Google, Bing, and Yahoo!, and is integrated into the default Hadoop versions. It observes the progress of each individual task and creates replicas for stragglers. The original straggler will not be killed upon speculation: the system will let the two copies compete with each other, adopting the quicker result to shorten the overall job completion. Although widely used in production systems, current speculation is low in its efficiency, characterized by a high failed speculation rate.

### 3.4.1 High Speculation Failure Rate

A failed speculation is defined as the redundant copy that does not surpass the straggler task in progress and ended up being killed by the speculator. Figure 3.10 shows an example of a failed speculation. In the Hadoop naming system, the original tasks are attempts marked with suffix 0 while the speculative copies are represented with suffix 1. From the example it is observable that, the created speculation got abandoned in the end due to the straggler succeeded first (refer to the "*Note*" column in Figure 3.10).



Show 20 entries  Search:

| Attempt | State | Status | Node | Logs | Start Time | Finish Time | Elapsed Time | Note |
|---|---|---|---|---|---|---|---|---|
| attempt_1437364567082_0057_m_000000_0 | SUCCEEDED | map > sort | /default-rack/hadoopcluster84:8042 | logs | Sat, 25 Jul 2015 02:52:37 GMT | Sat, 25 Jul 2015 03:04:12 GMT | 11mins, 35sec | |
| attempt_1437364567082_0057_m_000000_1 | KILLED | | /default-rack/hadoopcluster83:8042 | logs | Sat, 25 Jul 2015 03:00:51 GMT | Sat, 25 Jul 2015 03:04:12 GMT | 3mins, 20sec | Speculation: attempt_1437364567082_0057_m_000000_0 succeeded first! |

Showing 1 to 2 of 2 entries

Note
Killed
Speculation: attempt_0 succeeded first!
First Previous 1 Next Last

Figure 3.10: An example of a killed speculation for a Hadoop job

We use the OpenCloud trace as a case study, trying to find out that whether it is a common case for speculations to be killed in vain in production systems. Since the OpenCloud system provides Hadoop platform to its users, we get to know the information of whether a task is a speculative attempt or an original task through differentiating its suffix, either 0 or 1 as introduced above.

Figure 3.11 depicts the distribution of total task number versus speculation number versus killed speculation number for jobs within the OpenCloud system, from which we get a clear observation that a large proportion of speculations are actually been killed in the end. Figure 3.12 further demonstrates the statistical result of this proportion within the system. The speculation efficiency turns out to be surprisingly low, with average failure rate reach as high as more than 70%.

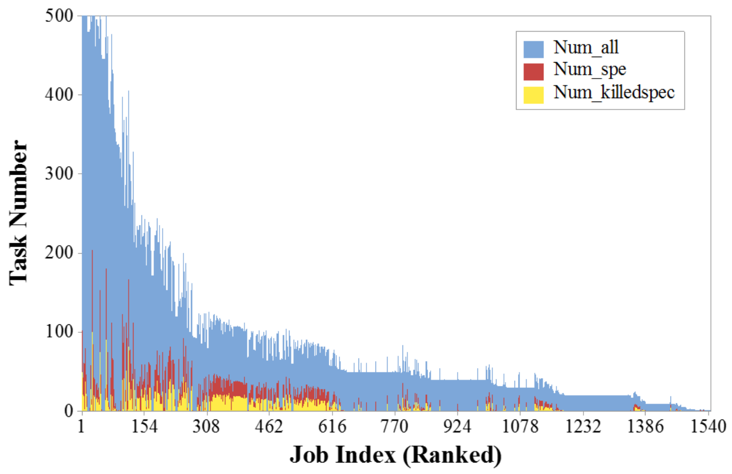


Figure 3.11: Numbers of speculation failure rate in the OpenCloud cluster

Similar findings are reported within other literature such as [27], which claims that in Yahoo!'s system, as many as 90% speculations are actually ended up being killed, with

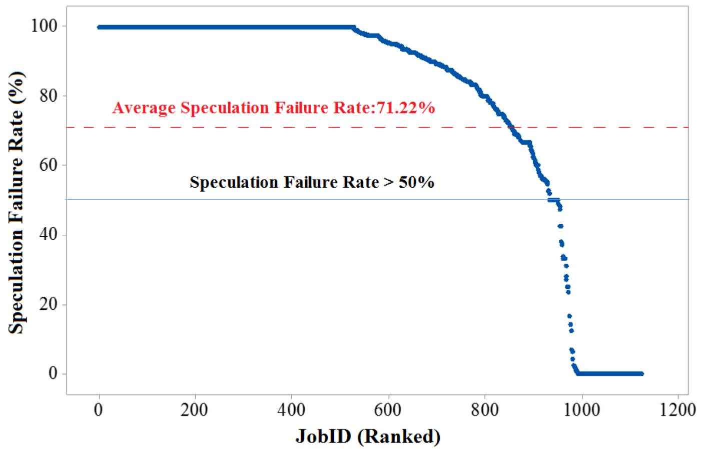


Figure 3.12: Statistics of speculation failure rate in the OpenCloud cluster

no benefits achieved in execution performance improvement. The high speculation failure rate observed in different clusters revealing a fact that current speculation method still has a long way to go in solving the straggler problem.

## 3.4.2 Improvement Potential

There are several reasons that can lead to the high speculation failure rate. For example, the stragglers are not identified in time, as a result, the corresponding replication is created too late to catch up; or the Estimated Completion Time (ECT) is not accurately estimated because the progress speed changes over time, etc. Figure 3.13 details three examples of how a MapReduce job can progress.

The progress completion score on the y-axis was derived from the Hadoop log. The Application Master in the YARN system records this information for all its task attempts, in the form of a fraction. Filtering out the corresponding progress report with event time generates Figure 3.13. And from the figure, the visible different gradient indicates Map and Reduce phases with different Progress Rate (PR) exhibited.

From the task progress pattern it is observable that, some stragglers exhibit their slowness at the very early stage. Improvement can be done in response to this situation, for example, to encourage speculation at this time point. Figure 3.14 explains this idea with high-level figures, showing the sensitivity of the straggler threshold towards the Map tasks. The shaded part covering stragglers calculated after applying a certain threshold. For Figure 3.14 (a), there is only one task being classified as the straggler until 00:00:10;

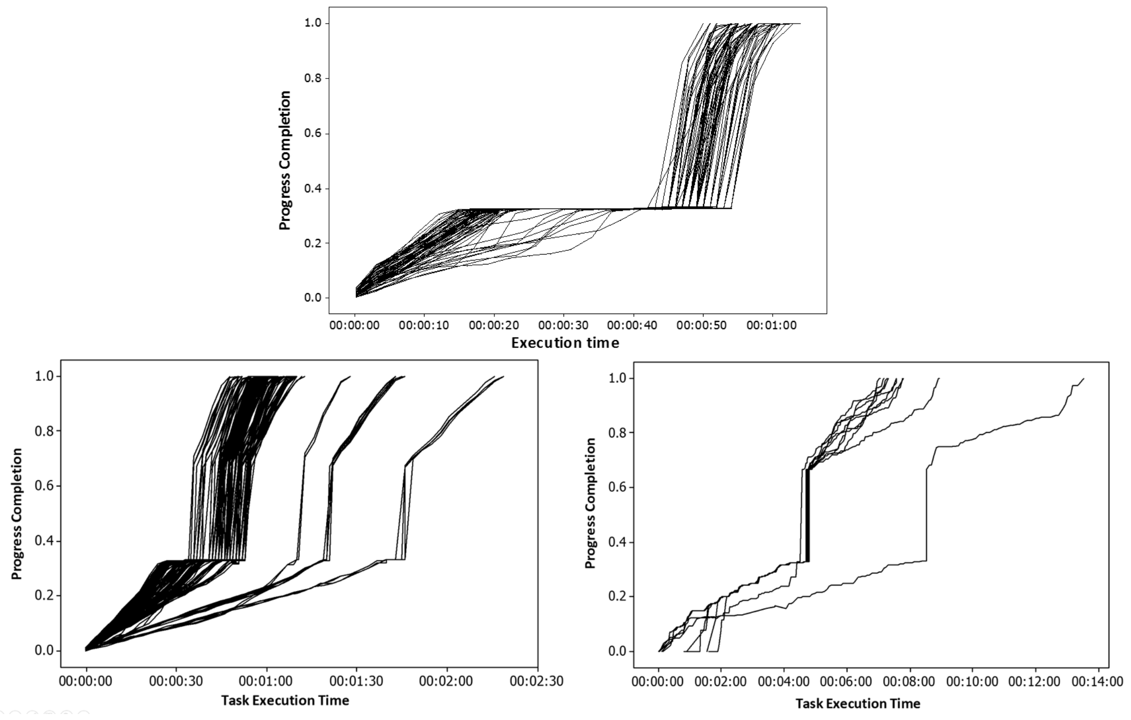


Figure 3.13: Three examples of how a typical MR job can progress

while (b) has already identified four of them at this time point.

There are many efforts can be done in order to improve current straggler mitigation efficiency besides the above example. Concrete achievements made in this thesis are detailed in the following sections. In addition, if we assume the theoretical best case performance a speculative-based method can achieve is to eliminate all stragglers and replace their duration with the average job execution time, we then get the speculation performance improvement potential.

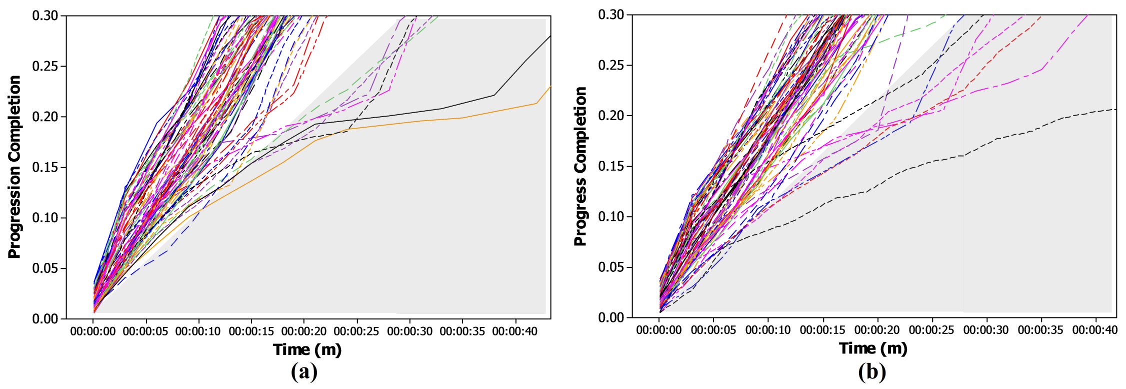


Figure 3.14: Threshold sensitivity toward straggler identification, (a) normal detection and (b) potential early detection

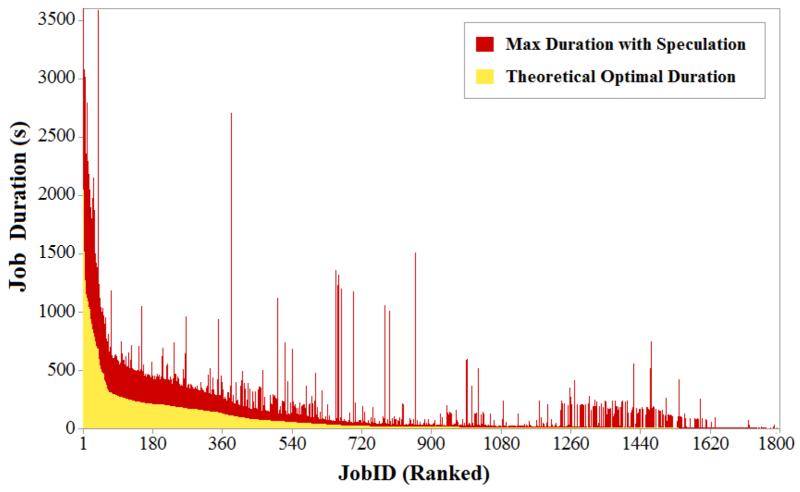


Figure 3.15: The improvement potential of the speculation in the OpenCloud cluster for jobs with duration less than an hour

Figure 3.15 illustrates this theoretical potential for OpenCloud jobs with a duration less than one hour. The huge gap between the actual execution time and the theoretical optimal duration indicates another 65.7% performance improvement in average for current speculation mechanism. Our research is trying to fill in this gap, and next section introduces the overall system model of our design.

## 3.5 Straggler Mitigation System Model

In order to solve the aforementioned challenge caused by the straggler problem and improve parallel computing performance, an intelligent straggler mitigation system is designed which works in conjunction with current parallel computing models. The overall system model is shown in Figure 3.16, and it can be embedded into popular platforms such as the Hadoop YARN implementation.

The main components of the system include a *History Statistic Calculator* that collects tasks execution tracelogs and does initial calculations; an enhanced *Adaptive Speculator* that adaptively calculates the most suitable threshold for straggler identification, a *Node Performance Analyzer* that models and predicts machine execution performance for a more efficient straggler mitigation; and a *Skew Pre-processor* to mitigate special stragglers caused by unbalanced input size.

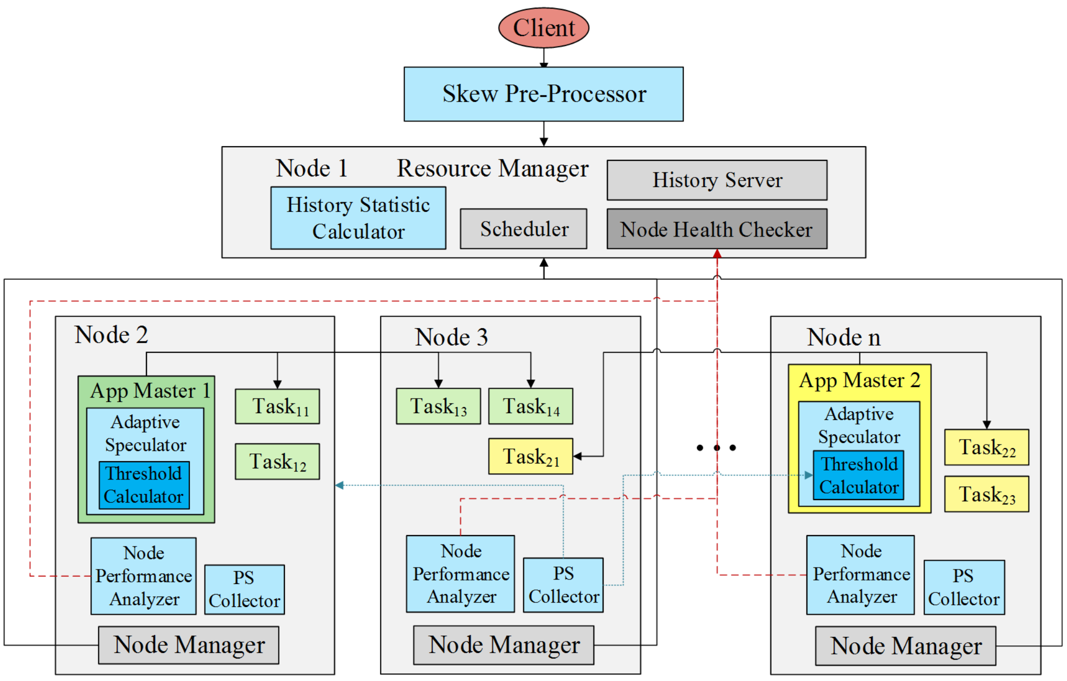


Figure 3.16: The intelligent straggler mitigation system model

For the *History Statistic Calculator*, its main obligation is data collection and initial calculation. For example, to collect Progress Score (PS) for each task, to classify them into jobs and machines for potential per job / per node analytics, to calculate the Estimated Completion Time (ECT) for all tasks for potential straggler identification, and to collect system environment statistics such as the resource utilization, etc. Besides the pre-calculations of the information needed in other components, the History Statistic Calculator is also responsible for data analytics that produces insights of the system. For example, the straggler influence analytics and the straggler reason correlation analytics discussed in this chapter are all categorized under the duty of this component.

For the *Adaptive Speculator*, its responsibility is to generate the adaptive straggler threshold according to the most up-to-date environmental factors, so that the suitable task stragglers can be picked up for speculation. This is important especially for those tasks that sit around the original static threshold. For example, for the tasks with an ECT at 45% or 55% larger than average when the threshold is pre-set to be 50%, whether or not to deal with them should depend upon the changing system conditions. Once the most appropriate stragglers are identified, the corresponding speculators will be submitted to the cluster scheduler and be assigned for execution following the default scheduling policy. The detailed design of this component is introduced in chapter 4.

For the *Node Performance Analyzer*, due to the fact that most straggler is caused by node-level reasons such as resource contention or disk faults, it is believed that, through avoiding assigning tasks onto slow performed nodes, or nodes that are about to experience a performance drop, can effectively reduce straggler occurrence. In addition, if speculative copies are launched on the fast nodes, it is predictable that the chances of speculation overtaking the straggler would be highly increased. To model and predict machine node performance is key to achieve those goals, and is the responsibility of the Node Performance Analyzer, which is discussed in chapter 5. In this design, the cluster scheduler remains untouched, but the available resources revealed to the scheduler.

And for the *Skew Pre-processor*, its goal is to eliminate data skew caused stragglers so that speculative-based method can play its due role. Skews are quite common in the MapReduce framework, especially for the Reduce phase because the distribution of the intermediate data is not pre-known. The Skew Pre-processor in our design works as a supplement component for the enhanced speculator that particularly deals with the Reduce skews. The detailed algorithm is given in chapter 6.

There are some general notation representations used throughout the whole thesis, some of them already appear in previous sections when discussed the straggler problem. They are briefly summarized as follows:

$J_j$: The $j^{th}$ job submitted into the system

$T_{ji}$: The $i^{th}$ task in job $J_j$; the total number of tasks for each job is $n$ $(0 < i \leq n)$

$M_k$: The $k^{th}$ machine in the cluster; it is assumed that there are $m$ nodes within cluster in total $(0 < k \leq m)$

$ECT_{ji}^t$: The estimated completion time of $T_{ji}$ at time $t$, $ECT_j^t = \max(ECT_{ji}^t)$

$\overline{ECT_j^t}$: The average estimated completion of $J_j$ at time $t$

$D_{ji}$: The duration of $T_{ji}$ from historical data

## 3.6 Summary

Increased complexity and scale of distributed systems has resulted in the manifestation of emergent phenomena substantially affecting overall system performance. There is limited work that empirically studies straggler root-cause and quantifies its impact upon system operation. Such analysis is critical to ascertain in-depth knowledge of straggler occurrence for focusing developmental and research efforts towards the long tail challenge.

This chapter provides an empirical analysis of straggler root-cause within Cloud datacenters; we analyze three large-scale production systems to quantify the frequency and impact stragglers impose. The contributions of this chapter are highlighted as follows:

- *As the minority, stragglers non-intuitively impact a huge proportion of jobs within large-scale systems.* It has been demonstrated that stragglers impose a substantive challenge towards rapid and predictable service execution for parallelizable applications, and is further aggravated by increased occurrence at growing system scale and complexity. Results demonstrate approximately 5 percent of task stragglers impact 50 percent of total jobs for batch processes

- *Straggler stems from numerous root-causes, among which, server-related issues predominantly influences straggler occurrence.* While root-cause analysis is a universal challenge across all fault-diagnosis research, there is a lack of in-depth analysis within datacenters which quantifies the underpinning root-cause and its frequency for stragglers. Such work is urgently needed for researchers to ascertain an intrinsic understanding of stragglers within real systems. In order to achieve this objective, the correlation analysis conducted within this chapter come from real-world systems discovers scientific understanding of straggler behavior that its occurence is predominantly influenced by server resource utilization. 53 percent of stragglers occur due to high server resource utilization.

- *Current straggler mitigation technique is far from effective in production systems with noticeable improvement potential.* Data analytics result based on OpenCloud cluster show that the failed speculation rate reaches as high as 71.22% in average, leads to a dramatic resource waste, and there is still another 65.7% improvement potential for job response times under current speculation scheme.

Refer to the system model described in Section 3.5, the results of the analysis contribute to the History Statistic Calculator component. Detailed descriptions of how other components are realized are presented in the following sections.

# Chapter 4

# Task-level Detection: Adaptive Straggler Threshold

A straggler is identified when applying a threshold once the Progress Score (PS) is collected and the related Estimated Completion Time (ECT) is calculated: if the ECT of a certain task is estimated to be longer than a certain percentage compared to the average value of all tasks within the same job, it will be marked 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 identified as stragglers. Therefore, the straggler threshold plays an important role in straggler identification and mitigation. This chapter proposes an adaptive straggler threshold calculation method that evaluates the most suitable tasks for speculation.

## 4.1 Algorithm Motivation

One of the biggest impacts the threshold value generates is the corresponding number of stragglers identified. Figure 4.1 (a) shows how the proportion of stragglers and the

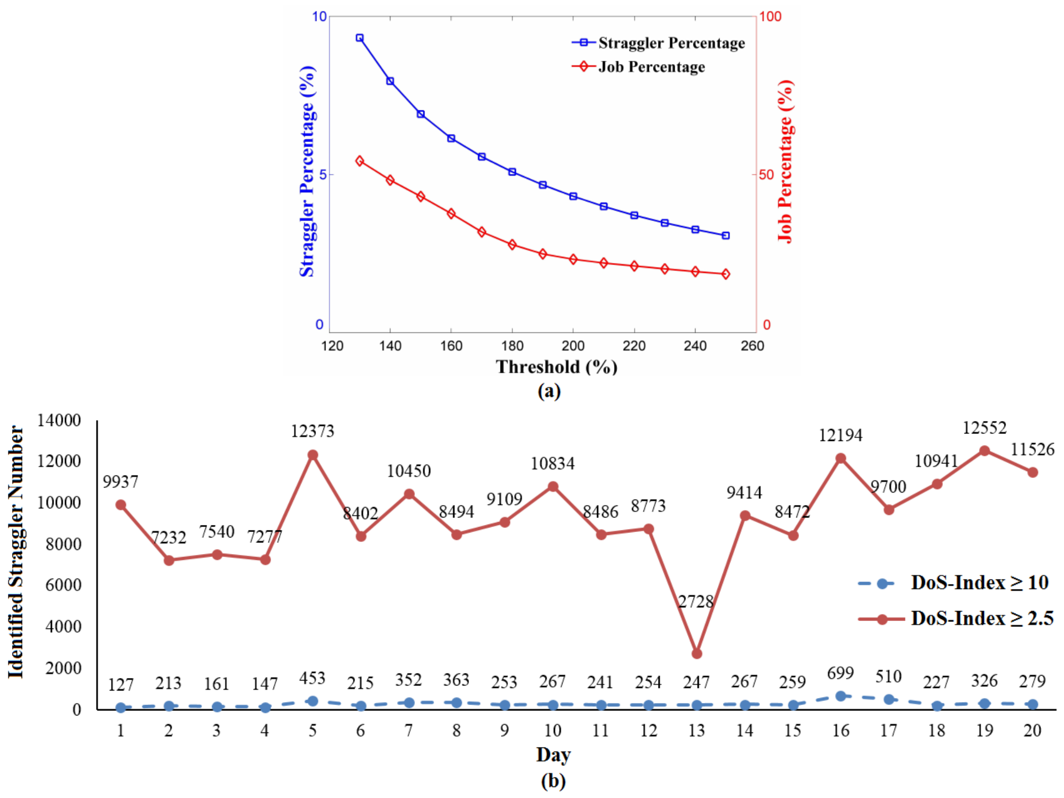


Figure 4.1: Relation between threshold setting and (a) straggler / tailing job proportion in the OpenCloud cluster; (b) different straggler numbers in AliCloud across the 20-day period using different threshold value

corresponding tailing jobs within the OpenCloud cluster are affected by different threshold values (Estimated Completion Time (ECT) based threshold) ranging from 120% to 260%. Tailing jobs are the jobs that containing stragglers. It is observable that, a larger (stricter) threshold can reduce the identified straggler number. A similar trend is seen in other systems as well. Figure 4.1 (b) illustrates the relationship between threshold setting and different straggler numbers in AliCloud system across the 20-day period when using $Degree of Straggler Index$(DoS-Index) as the threshold. Changing the threshold value from 2.5 to 10 results in the number of filtered stragglers down from 186,434 to 5,560 (from 9,322 to 293 a day on average).

Since speculative execution results in the creation of task replicas upon straggler detection, different straggler number will directly impact the availability of the system and further influence resource allocation. In addition, different types of application often have different requirements toward timely response, and should correspond to different levels of effort when improving their performance. Therefore, it is important to choose the most suitable number of straggler tasks for speculation at certain time point for a certain type of

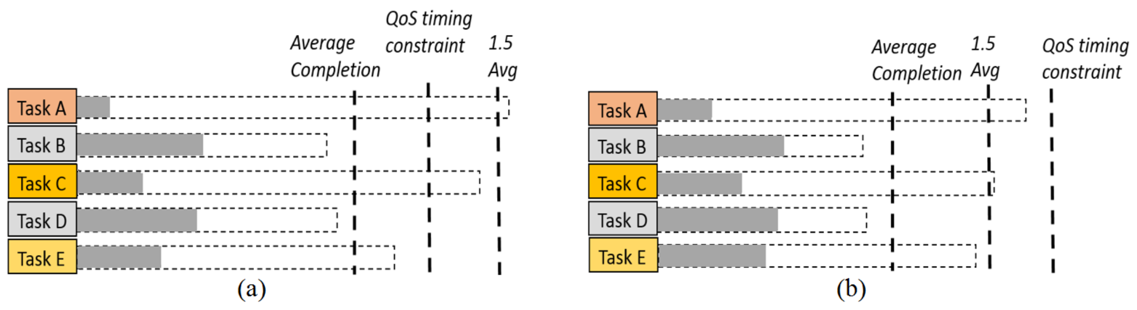


Figure 4.2: Dynamic threshold motivation dealing with (a) strict and (b) lax QoS timing constraint

application. Figure 4.2 illustrates the goal that motivates our design and the need for the adaptive threshold. As a comparison, the static threshold in the graph is set to the value of 1.5. This is the most common value used in related literature, and can be changed if needed. Actually, the exact value of the threshold is not the key issue here, it is the static nature that debilitates speculation effectiveness.

The timing requirement of a certain application is normally given in its Quality of Service (QoS) specifications. 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 4.2 (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 constraint, 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 4.2 (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* so that the saved resources can be used for other applications. 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 adaptively capturing these scenarios.

In conclusion, a static threshold can debilitate speculation effectiveness as it fails to capture the intrinsic diversity of timing constraints of applications, as well as the 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. And this motivates the design of the adaptive threshold calculation algorithm.

## 4.2 Algorithm Design

An algorithm that adaptively calculates a dynamic threshold which can automatically adjust its value according to different operation situations is proposed in order to improve speculation efficiency. This algorithm leverages three key factors: job QoS timing constraint, task lifecycle progress, and system resource utilization to judge whether a task replica should be created to tolerate task stragglers.

The dynamic threshold is periodically updated at a certain time interval $t$ in order to allow the algorithm to adapt to the up-to-date system environment. The scheduler will label task $T_{ji}$ as straggler if condition $ECT^t_{ji} > Th^t_{j,dyn} * \overline{ECT^t_j}$ is fulfilled. Upon straggler detection, within current Hadoop YARN implementation [147], the duplicate task will be created and assigned for running using the $addSpeculativeAttempt(taskID)$ function. Equation 4.1 depicts the calculation of the adaptive threshold value per job at a high level:

$$Th^t_{j,dyn} = Q^t_j + \alpha * P^t_j + \beta * R^t \tag{4.1}$$

where $Q^t_j$ denotes the threshold baseline determined by job Quality of Service (QoS) timing constraint. $P^t_j$ 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 current cluster resource utilization level. Weight parameters $\alpha$ and $\beta$ can be specified by the system administrator to demonstrate a particular emphasis on resource utilization or progress. The additional notations used within this chapter has been summarized in Table 4.1.

The weighted sum of $Q^t_j$, $P^t_j$ and $R^t$ produce the threshold value for detecting stragglers, and the pseudo code for this dynamic straggler threshold calculation is given in algorithm 1. At every timestamp $t$, stragglers are identified according to the calculated threshold and their Estimated Completion Time (ECT), the calculation of which varies. In this work, the most commonly applied estimating approach illustrated in Equation 2.8 is adopted. Other sophisticated method can be used to replace current calculation if needed.

A higher value for $Th^t_{j,dyn}$ indicates a stricter straggler threshold enforced at that particu-

**ALGORITHM 1:** DynamicStragglerThreshold

```
Inputs:
        Jobs: the list of all jobs within the cluster, each "job" element has attributes such as
        Tasks list and PS list, etc.
        Nodes: the machine node list within the cluster
        Interval: the time interval of the threshold calculation
        alpha, beta: $\alpha$, $\beta$, the adjustor weightings
        mu: $\mu$, the progress standard parameter
        phi, omega: $\phi$, $\omega$, the CPU and the memory utilization standard parameter
1  while Jobs.size > 0 do
2      for each Jobs[j] in Jobs do
3          Tasks = Jobs[j].Tasks, PS = Jobs[j].PS;
4          Q[j] = $TimingConstraintBaseline(Jobs[j], Tasks, PS, Jobs[j].QoS)$;
5          P[j] = $TaskLifeCycleAdjustor(Jobs[j], Tasks, PS, mu)$;
6          R[j] = $UtilizationAjustor(Nodes, phi, omega)$;
7          Th[j] = Q[j] + alpha*P[j] + beta*R[j];
8          for each Tasks[i] in Jobs[j].Tasks do
9              Jobs[j].PR[i] = Jobs[j].PS[i] / ($CurrentTime -$ Jobs[j].startTime);
10             Jobs[j].ECT[i] = $CurrentTime$ + (1 $-$ Jobs[j].PS[i]) / Jobs[j].PR[i];
11             if (Jobs[j].ECT[i] > (Th[j] * average (Jobs[j].ECT))) then
12                 if (Tasks[i].AlreadySpeculated == False) then
13                     AddSpecAttempt(Tasks[i]);
14                     Tasks[i].AlreadySpeculated = True;
15                     Jobs[j].size += 1;
16     $sleep$(Interval);
```

Table 4.1: Additional notations used in the adaptive threshold

| | |
|---|---|
| $PS_{ji}^t$ | The progress score at time $t$ for $T_{ji}$ |
| $PR_{ji}^t$ | The progress rate at time $t$ for $T_{ji}$ |
| $Th_{j,dyn}^t$ | The dynamic threshold at time $t$ for $J_j$ |
| $Th_{j,stat}^t$ | The static threshold at time $t$ for $J_j$ |
| $Q_j^t$ | The QoS adjustor at time $t$ for $J_i$ |
| $P_j^t$ | The progress adjustor at time $t$ for $J_j$ |
| $R^t$ | The cluster average resource utilization adjustor at time $t$ |
| $\alpha$ | The progress weight parameter |
| $\beta$ | The resource utilization weight parameter |
| $\mu$ | The progress standard parameter |
| $\phi$ | The CPU utilization standard parameter |
| $\omega$ | The memory utilization standard parameter |
| $\Omega_k^t$ | The memory utilization of machine $M_k$ at time $t$ |
| $\Phi_k^t$ | The CPU utilization of machine $M_k$ at time $t$ |

lar time, in which case, fewer task would meet the standard to be classified as stragglers and trigger the speculation, while a lower $Th^t_{j,dyn}$ value allows a more relaxed condition when generating speculated replicas. The values for $Q^t_j$, $P^t_j$ and $R^t$ are derived by lower levels of calculation.

### 4.2.1 QoS Timing Constraint

The QoS timing constraint is an important factor to be considered when deciding how rigorous the straggler threshold should be based on the nature of the application. For example, a real-time service might emphasize a compulsory response time 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 for such job is more important than saving resources on speculation. In our design, the *QoS Timing Constraint* parameter is used to set the threshold baseline. This allows for different degrees of strictness for generating replicas when tolerating the impact of stragglers. The calculation of the QoS baseline at time $t$ is given in Equation 4.2 where $S^t_j = \{x|x = ECT^t_{ji} > QoS\}$.

$$Q^t_j = \begin{cases} QoS/\overline{ECT^t_j} & \text{if } QoS \geq \max(ECT^t_{ji}) \\ \{\min\limits_{\forall ECT^t_{ji} \in S^t_j} ECT^t_{ji}\}/\overline{ECT^t_j} & \text{if } QoS < \max(ECT^t_{ji}) \\ 1.5 & \text{if no QoS} \end{cases} \tag{4.2}$$

In addition, QoS stands for the time requirement defined in the QoS parameter. For cases when the maximum $ECT^t_{ji}$ is estimated to be within the deadline constraint, the threshold is set to be the quotient of QoS and the average $ECT^t_{ji}$. This relatively large threshold guarantees that, while no QoS breakdown happens, there is no need to create a lot of replicas because no threats from severe performance consequences would occur. And for cases when there exists an $i$ that $ECT^t_{ji} > QoS$ stands, the threshold is then calculated to be the minimal ECT that is larger than QoS divided by the average $ECT^t_{ji}$, because such a value can capture all such $i$.

Two examples of how this is calculated are given as follows: assume that a job with QoS timing constraint of 300ms has five tasks with ECTs at time $t$ to be 290ms, 290ms, 300ms, 380ms, and 400ms, respectively. In this scenario, the minimal $ECT^t_{ji}$ greater than QoS is

380ms, and the average ECT is 300ms. Therefore the value for $Q_j^t$ to be used calculating the final threshold $Th_{j,dyn}^t$ is 127% (380 ÷ 300). If all tasks are estimated to complete prior to the specified QoS (change QoS in above example from 300ms to 450ms, then all $ECT_{ji}^t$ values will be smaller than QoS), the value for $Q_j^t$ according to Equation 4.2 will then change to 150% (450 ÷ 300). This results in no tasks detected as stragglers (if $P_j^t$ and $R^t$ are zero as in this example).

**ALGORITHM 2:** TimingConstraintBaseline

```
Inputs:
        Job: the parallel job
        Tasks: the list of tasks within this job
        PS: The progress score list of every task within this job
        QoS: QoS timing constraint
Output:
        Q: the timing constraint baseline
1  for each Tasks[i] in Tasks do
2      PR[i] = PS[i] / ($CurrentTime$ − Job.startTime);
3      ECT[i] = $CurrentTime$ + (1 − PS[i]) / PR[i];
4  if (QoS ≠ null) then
5      $Sort$(ECT);
6      Q = (ECT > QoS).$first$;
7      if (Q ≠ null) then
8          Q = Q / $average$(ECT);
9      else
10         Q = QoS / $average$(ECT);
11 else
12     Q = 1.5;
```

It is worth highlighting that, the adaptive straggler threshold algorithm also functions well for applications that do not specify an explicit QoS timing request. In such an event, a static time proportion value of 150% used in current literatures [162] [125] [80] can be applied to set the threshold baseline, and the dynamic change for $Th_{j,dyn}^t$ will then depend on $P_j^t$ and $R^t$. The algorithm 2 details the calculation process of $Q_j^t$.

## 4.2.2 Task Lifecycle Progress

It is also important to consider the current completed progress 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 final job completion time. For example, when a task experiences slowdown in its later phase, the 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 value 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 4.3

$$P_j^t = \overline{PS_j^t} - \mu \tag{4.3}$$

where $\overline{PS_j^t}$ is the average Progress Score (PS) for job $J_j$ at time $t$, representing the current phase in the entire job lifecycle. For example, for a job consistis of 3 tasks, at time $t = 1$, if $PS_{11}^1 = PS_{12}^1 = PS_{13}^1 = 0.1$, we say $\overline{PS_1^1} = 0.1$. With task progress, at time $t = 2$ if $PS_{11}^2 = PS_{12}^2 = 0.5$ while $PS_{13}^2 = 0.2$, then we say the overall job is at its progress of $0.4$ ($\overline{PS_1^2} = 0.4$).

Progress standard parameter $\mu$ is used to denote the specified maximum point within the lifecycle suitable for generating a replica. For example, $\mu = 0.5$ represents that any job with a PS smaller or equal to 0.5 is still considered as in its early stage, leading to a smaller $Th_{j,dyn}^t$ by generating a negative $P_j^t$ value, increasing the likelihood of replica generation. And any job that progresses past half of the entire lifecycle will be treated as in its late stage, resulting in a positive $P_j^t$ to enlarge the threshold to limit replica generation. This procedure is detailed in algorithm 3.

**ALGORITHM 3:** TaskLifeCycleAdjustor

```
Inputs:
        Job: the parallel job
        Tasks: the list of tasks within this job
        PS: The progress score list of every task within this job
        mu: $\mu$, the progress threshold
Output:
        P: the task lifecycle adjustor
1 sumPS = 0;
2 for each Tasks[i] in Tasks do
3     sumPS += PS[i];
4 P = sumPS / Job.size − mu;
```

### 4.2.3 System Resource Usage

One of the most important considerations for straggler mitigation systems is the overhead incurred by speculations, especially toward different system conditions. Creating replicas in a high resource utilization situation poses a greater threat to system stability and can further increase the likelihood of straggler occurrence due to severe contention [107], while low system utilization allows for additional speculation to improve job completion such as the full cloning algorithm [6]. Furthermore, replicas themselves could have the chance to become stragglers. Observations proposed in [5] state that 3% of replica executions still take ten times longer than normal tasks 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 large, 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 system utilization levels. The resource adjustor is represented as parameter $R^t$ in the algorithm to tune the value of $Th^t_{j,dyn}$ dependent on system utilization at time $t$. This calculation is given as

$$R^t = \max\left(\frac{\sum_{k=1}^{n} \Omega_k^t}{n} - \omega, \frac{\sum_{k=1}^{n} \Phi_k^t}{n} - \phi\right) \tag{4.4}$$

where $n$ denotes the total number of server nodes within the cluster, while $\Omega^t_k$ and $\Phi^t_k$

**ALGORITHM 4:** UtilizationAdjustor

```
Inputs:
        phi, omega: $\phi$, $\omega$, the CPU and the memory threshold
        Nodes: the machine nodes list within the cluster
Output:
        R: the utilization adjustor
1 sumCPU = 0, sumMem = 0;
2 for each Nodes[n] in Nodes do
3     sumCPU += Nodes[n].getCpuUti();
4     sumMem += Nodes[n].getMemUti();
5 C_Adjustor = sumCPU / Nodes.size − phi;
6 M_Adjustor = sumMem / Nodes.seze − omega;
7 R = $max$(CPUadjustor, MEMadjustor);
```

represent the memory and CPU utilization of machine $M_k$, respectively. If either one of these two parameters surpasses the optimal utilization specified by the user (the memory standard $\omega$ and the CPU standard $\phi$), the equation will increase the threshold by generating a positive $R^t$, resulting in a stricter requirement for replica generation. For now, only CPU and memory utilization are considered, and the calculation catches the influence of whichever that is higher in its utilization. Additional resources can be included to generate a complete computation in the future. The detailed process is summarized in algorithm 4.

## 4.3 Theoretical Examples

In this section, the key idea of the proposed algorithm is explained by giving two theoretical examples, illustrating how the dynamic threshold value will change according to different parameters, and what impact this change would exhibit upon final job completion. The series of Progress Score (PS) is given one timestamp at a time. Among the two examples, one summarizes the case when the job starts in a low system utilization state while the other is for high utilization.

The static and dynamic thresholds are applied to both examples and the results are presented in four tables. In both examples, a single job $J_1$ consisting of ten tasks $T_{1,1}, \dots, T_{1,10}$ is considered 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 the 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.

- Example 1: Job starts in low utilization environment

Table 4.2 shows the job PS and the Estimated Completion Time (ECT) at each time interval for the adaptive threshold while Table 4.3 indicates static threshold. The initial system utilization starts from 15% to represent the idle condition of the cluster. It is assumed that the job does not have an Quality of Service (QoS) deadline request in this example, and the values for $T_{1,1}, \dots, T_{1,10}$ in the table are ECTs of the initial tasks, while R-$T_{ji}$ stands for corresponding speculative replicas. $ECT_{compare}$ in both tables represent the result of $Th^t_{1,dyn} \times \overline{ECT^t_1}$.

Observed from Table 4.3, the static threshold identifies two stragglers and the job completes at timestamp $t = 12$. Task $T_{1,7}$ is the first straggler identified, being detected at timestamp $t = 2$ when its ECT is first found to be larger than $1.5 * \overline{ECT_1^2}$. Replica R-$T_{1,7}$ is created upon detection, and its ECT is calculated at time $t = 3$ when PS is first generated. The other straggler identified is task $T_{1,3}$ with R-$T_{1,3}$ created at timestamp $t = 7$. Among the two replicas, R-$T_{1,7}$ completes at time $t = 8$ prior to task $T_{1,7}$, however for $T_{1,3}$, due to the late detection, R-$T_{1,3}$ does not have sufficient time to catch up. From this demonstration it is observable that, the static threshold cannot capture slow tasks such as $T_{1,3}$ promptly; even in its early stage, it already shown slow progress (at timestamp $t = 3$ when its ECT is 13, however this number does not pass the 1.5 threshold because it is slightly smaller than 13.5).

On the contrary, for the dynamic threshold demonstrated in Table 4.2, this gap of late detection is reduced. Altogether four stragglers are identified to shorten the job execution from timestamp 2 to 4. Although the larger replica number imposes a greater overhead compared with the static threshold, due to the idle system state, it will not cause severe burden towards the system. This is why tasks $T_3, T_5, T_7$ and $T_{10}$ are identified in comparison to only $T_3$ and $T_7$ when 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.

It is also observable that, the threshold value gets larger through job execution. This is due to the fact that, the dynamic approach limits replica creation in late stages of the lifecycle by generating greater threshold values according to Equation 4.3. Increasing the straggler threshold indicates a stricter condition for a slow task to be classified as a straggler and to trigger speculation. In late phases of a job, due to the fact that speculations are not likely to over-pace the original task unless it is a really severe straggler, increasing threshold won't undermine the makespan, but would save resources on unnecessary speculation. And as time goes by, the threshold stops increasing and stays at a relatively stable value. This is because that, when tasks start finishing, the usage of the resource adjustor defined in Equation 4.4 yields a decreasing effect that neutralizes the increasing effect brought by the late progress phase.

- Example 2: Job starts in high utilization environment

For the second example, Table 4.4 and Table 4.5 show an application that adopts the dynamic and the static straggler thresholds, respectively, in which, job starts at a high utilization state. This time, the job with a QoS constraint $QoS = 12$ is considered to demonstrate the algorithm methodology. Other attributes including the task number and the algorithm parameter settings are identical to the previous example.

Table 4.2: Example 1: speculative execution with dynamic threshold

| Time | Uti | $Th^t_{1,dyn}$ | $ECT_{compare}$ | PS | $T_{1,1}$ | $T_{1,2}$ | $T_{1,3}$ | $T_{1,4}$ | $T_{1,5}$ | $T_{1,6}$ | $T_{1,7}$ | $T_{1,8}$ | $T_{1,9}$ | $T_{1,10}$ | R-$T_{1,3}$ | R-$T_{1,5}$ | R-$T_{1,7}$ | R-$T_{1,10}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.15 | 1.09 | 8.20 | 0.14 | 7 | 7 | 8 | 7 | 8 | 5 | 8 | 8 | 7 | ***10*** | | | | |
| 2 | 0.15 | 1.15 | 9.31 | 0.25 | 7 | 6 | 9 | 6 | ***10*** | 6 | ***13*** | 7 | 7 | 10 | | | | 8 |
| 3 | 0.20 | 1.21 | 10.58 | 0.31 | 7 | 8 | ***13*** | 8 | 10 | 7 | 15 | 7 | 7 | 9 | | 8 | 8 | 7 |
| 4 | 0.50 | 1.42 | 11.77 | 0.44 | 7 | 7 | 12 | 8 | 10 | 6 | 16 | 7 | 5 | 9 | 6 | 8 | 7 | 8 |
| 5 | 0.55 | 1.51 | 12.39 | 0.57 | 8 | 7 | 12 | 8 | 9 | 6 | 14 | 6 | 6 | 10 | 7 | 7 | 7 | 8 |
| 6 | 0.55 | 1.58 | 12.74 | 0.71 | 7 | 6 | 11 | 7 | 10 | 6 | 14 | 7 | 7 | 9 | 7 | 8 | 6 | 8 |
| 7 | 0.55 | 1.63 | 13.19 | 0.82 | 7 | | 13 | 7 | 10 | | 13 | 7 | 7 | 10 | 7 | 7 | 6 | 7 |
| 8 | 0.25 | 1.54 | 12.45 | 0.93 | | | 12 | | 11 | | ***14*** | | | ***10*** | 6 | 7 | 6 | 7 |
| 9 | 0.20 | 1.54 | 12.40 | 0.97 | | | 12 | | ***10*** | | | | | | 7 | 7 | | |
| 10 | 0.20 | 1.55 | 12.40 | 1.00 | | | ***11*** | | | | | | | | 7 | | | |

Table 4.3: Example 1: speculative execution with static threshold.

| Time | Uti | $Th^t_{1,sta}$ | $ECT_{compare}$ | PS | $T_{1,1}$ | $T_{1,2}$ | $T_{1,3}$ | $T_{1,4}$ | $T_{1,5}$ | $T_{1,6}$ | $T_{1,7}$ | $T_{1,8}$ | $T_{1,9}$ | $T_{1,10}$ | R-$T_{1,3}$ | R-$T_{1,7}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.15 | 1.50 | 11.25 | 0.14 | 7 | 7 | 8 | 7 | 8 | 5 | 8 | 8 | 7 | 10 | | |
| 2 | 0.15 | 1.50 | 12.15 | 0.26 | 7 | 6 | 9 | 6 | 10 | 6 | ***13*** | 7 | 7 | 10 | | |
| 3 | 0.30 | 1.50 | 13.50 | 0.33 | 7 | 8 | 13 | 8 | 10 | 7 | 15 | 7 | 7 | 9 | | 8 |
| 4 | 0.30 | 1.50 | 12.82 | 0.49 | 7 | 7 | 12 | 8 | 10 | 6 | 16 | 7 | 5 | 9 | | 7 |
| 5 | 0.35 | 1.50 | 12.68 | 0.61 | 8 | 7 | 12 | 8 | 9 | 6 | 14 | 6 | 6 | 10 | | 7 |
| 6 | 0.40 | 1.50 | 12.27 | 0.76 | 7 | 6 | 11 | 7 | 10 | 6 | 14 | 7 | 7 | 9 | | 6 |
| 7 | 0.35 | 1.50 | 12.55 | 0.85 | 7 | | ***13*** | 7 | 10 | | 13 | 7 | 7 | 10 | | 6 |
| 8 | 0.25 | 1.50 | 12.38 | 0.86 | | | 12 | | 11 | | ***14*** | | | 10 | 6 | |
| 9 | 0.20 | 1.50 | 12.38 | 0.90 | | | 12 | | 10 | | | | | 10 | 7 | |
| 10 | 0.20 | 1.50 | 12.13 | 0.95 | | | 11 | | 10 | | | | | 10 | 6 | |
| 11 | 0.20 | 1.50 | 12.38 | 0.96 | | | 12 | | | | | | | | 7 | |
| 12 | 0.20 | 1.50 | 12.38 | 0.98 | | | 12 | | | | | | | | ***7*** | |

Table 4.4: Example 2: speculative execution with dynamic threshold

| Time | Uti | $Th^t_{1,dyn}$ | $ECT_{compare}$ | PS | $T_{1,1}$ | $T_{1,2}$ | $T_{1,3}$ | $T_{1,4}$ | $T_{1,5}$ | $T_{1,6}$ | $T_{1,7}$ | $T_{1,8}$ | $T_{1,9}$ | $T_{1,10}$ | R-$T_{1,8}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.80 | 1.52 | 11.39 | 0.14 | 8 | 7 | 7 | 8 | 7 | 5 | 8 | 8 | 7 | 10 | |
| 2 | 0.80 | 1.59 | 12.85 | 0.26 | 9 | 6 | 7 | 10 | 6 | 6 | 7 | ***13*** | 7 | 10 | |
| 3 | 0.95 | 1.68 | 13.91 | 0.37 | 13 | 7 | 6 | 10 | 7 | 6 | 6 | 13 | 6 | 9 | 8 |
| 4 | 0.90 | 1.84 | 13.41 | 0.59 | 11 | 6 | 5 | 11 | 5 | 6 | 5 | 12 | 5 | 7 | 7 |
| 5 | 0.85 | 1.96 | 15.65 | 0.66 | 11 | 6 | 6 | 12 | 6 | 6 | 6 | 14 | 6 | 8 | 7 |
| 6 | 0.80 | 1.88 | 15.88 | 0.75 | 11 | 6 | 7 | 13 | 7 | 6 | 7 | 14 | 7 | 9 | 6 |
| 7 | 0.80 | 1.79 | 15.29 | 0.84 | 13 | | 7 | 12 | 7 | | 7 | 13 | 7 | 10 | 6 |
| 8 | 0.50 | 1.82 | 15.38 | 0.93 | 12 | | | 11 | | | | ***14*** | | 10 | 6 |
| 9 | 0.30 | 1.48 | 12.62 | 0.95 | 12 | | | 12 | | | | | | 10 | |
| 10 | 0.35 | 1.43 | 11.98 | 0.98 | 11 | | | 11 | | | | | | 10 | |
| 11 | 0.20 | 1.46 | 12.36 | 0.98 | 12 | | | 12 | | | | | | | |
| QoS = 12 | 0.15 | 1.43 | 12.21 | 1.00 | 12 | | | 12 | | | | | | | |

Table 4.5: Example 2: speculative execution with static threshold

| Time | Uti | $Th^t_{1,sta}$ | $ECT_{compare}$ | PS | $T_{1,1}$ | $T_{1,2}$ | $T_{1,3}$ | $T_{1,4}$ | $T_{1,5}$ | $T_{1,6}$ | $T_{1,7}$ | $T_{1,8}$ | $T_{1,9}$ | $T_{1,10}$ | R-$T_{1,1}$ | R-$T_{1,4}$ | R-$T_{1,8}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.80 | 1.50 | 11.25 | 0.14 | 8 | 7 | 7 | 8 | 7 | 5 | 8 | 8 | 7 | 10 | | | |
| 2 | 0.80 | 1.50 | 12.15 | 0.26 | 9 | 6 | 7 | 10 | 6 | 6 | 7 | ***13*** | 7 | 10 | | | |
| 3 | 0.95 | 1.50 | 12.41 | 0.37 | ***13*** | 7 | 6 | 10 | 7 | 6 | 6 | 13 | 6 | 9 | | | 7 |
| 4 | 0.90 | 1.50 | 10.75 | 0.55 | 11 | 6 | 5 | ***11*** | 5 | 6 | 5 | 12 | 5 | 7 | 7 | | 7 |
| 5 | 0.95 | 1.50 | 11.77 | 0.59 | 11 | 6 | 6 | 12 | 6 | 6 | 6 | 14 | 6 | 8 | 7 | 7 | 6 |
| 6 | 0.90 | 1.50 | 12.35 | 0.69 | 11 | 6 | 7 | 13 | 7 | 6 | 7 | 14 | 7 | 9 | 7 | 6 | 6 |
| 7 | 0.85 | 1.50 | 12.35 | 0.79 | 13 | | 7 | 12 | 7 | | 7 | 13 | 7 | 10 | 6 | 5 | 6 |
| 8 | 0.70 | 1.50 | 12.00 | 0.91 | 12 | | | 11 | | | | ***14*** | | 10 | 7 | 6 | 6 |
| 9 | 0.65 | 1.50 | 12.35 | 0.93 | 12 | | | 12 | | | | | | 10 | 7 | 7 | |
| 10 | 0.50 | 1.50 | 12.23 | 0.98 | ***11*** | | | 11 | | | | | | 10 | 7 | 7 | |
| 11 | 0.50 | 1.50 | 12.35 | 1.00 | | | | ***12*** | | | | | | | | 7 | |
| QoS = 12 | | | | | | | | | | | | | | | | | |

As presented in Table 4.5, due to the slow progress, the static approach identifies three tasks $T_{1,1}, T_{1,4}$ and $T_{1,8}$ as stragglers regardless of the utilization level, followed by the creation of replicas R-$T_{1,1}$, R-$T_{1,4}$ and R-$T_{1,8}$ at times 3, 4 and 2, respectively. These additional replicas further increase utilization as well as the probability of straggler occurrence. In addition, some of them do not generate obvious improvement toward execution performance, for example, R-$T_{1,1}$, R-$T_{1,4}$ are both only one step earlier than the original task. For the dynamic method demonstrated in Table 4.4, due to the awareness of the environmental conditions through using the resource adjustor defined in Equation 4.4, it only identifies the most noticeable straggler $T_{1,8}$. As a result of taking QoS into consideration ($QoS = 12$), the slower ECT of $T_{1,1}$ and $T_{1,4}$ (ECT $= 12$) are ignored as a trade-off for better resource efficiency while still guarantees acceptable response time. To summarize, the dynamic threshold creates fewer replicas in the case of high system utilization to avoid overloaded system, while guaranteeing the fulfillment of the QoS requirement.

## 4.4 Implementation and Experiments

The proposed method is implemented into the real system for experimental evaluation. This section first introduces the default speculator design in current YARN system, followed by the introduction of how the modification is made in order to integrate the dynamic threshold in. Experiment setups and results are discussed at the end.

### 4.4.1 Default Speculator Component

The default speculator component in current YARN 2.5.2 [66] implementation mainly consists of three key classes: the *TaskRuntimeEstimator* class which is responsible for estimating task Estimated Completion Time (ECT)s; the *AppContext* in charge of sharing information between different objects; and the *Speculator* class itself. Key methods and parameters are detailed in Figure 4.3.

Every time when the "*speculation*" event is triggered (refer to Figure 2.11 for the detailed events handled by YARN Application Master (AM)), the *Speculator* will check whether a speculation action should be launched according to conditions given by the parameters. By defualt, this time interval is set to be 1 second after no speculation, and 15 seconds after speculation, and the parameters mainly include the *minimum_allowed_speculative_tasks*

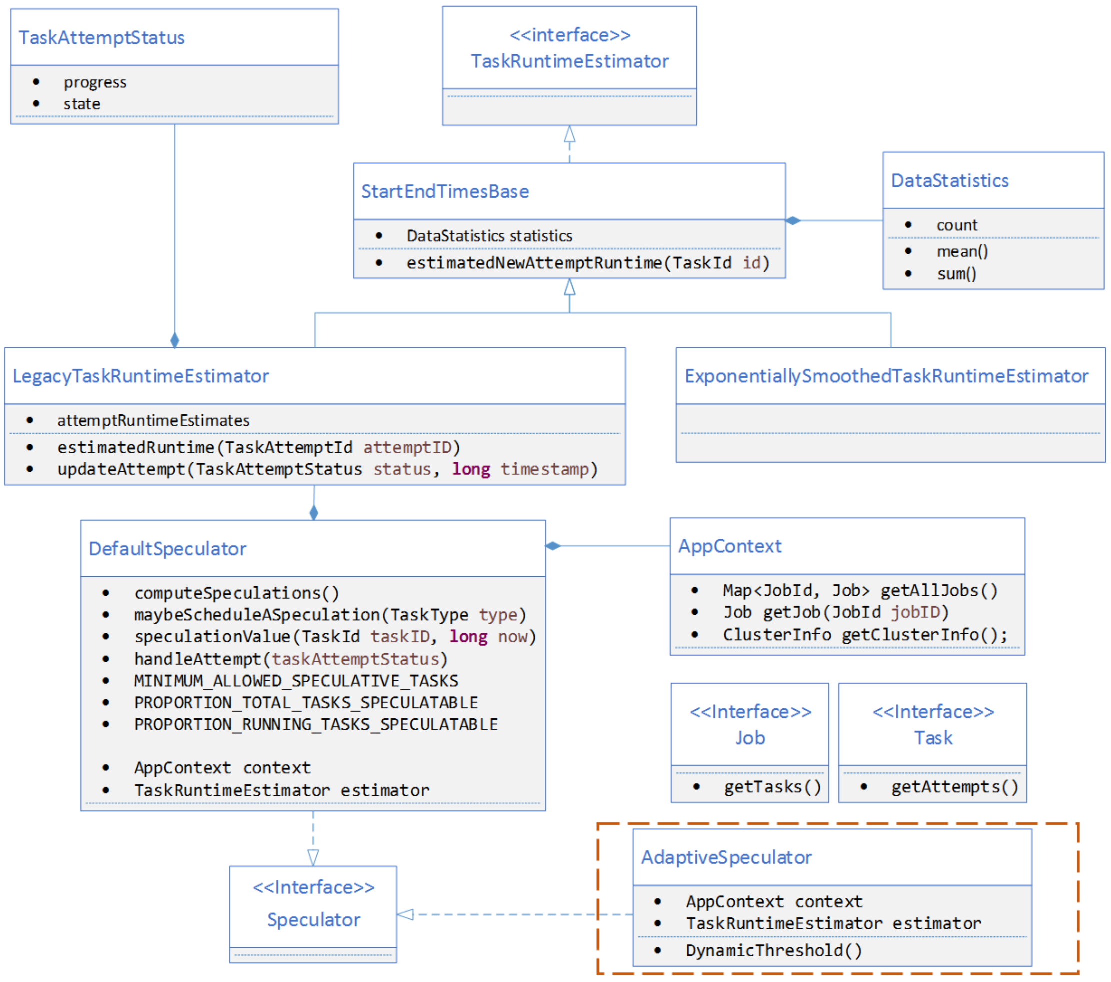

TaskAttemptStatus
progress
state
<<interface>> TaskRuntimeEstimator
StartEndTimesBase
DataStatistics statistics
estimatedNewAttemptRuntime(TaskId id)
DataStatistics
count
mean()
sum()
LegacyTaskRuntimeEstimator
attemptRuntimeEstimates
estimatedRuntime(TaskAttemptId attemptID)
updateAttempt(TaskAttemptStatus status, long timestamp)
ExponentiallySmoothedTaskRuntimeEstimator
DefaultSpeculator
computeSpeculations()
maybeScheduleASpeculation(TaskType type)
speculationValue(TaskId taskID, long now)
handleAttempt(taskAttemptStatus)
MINIMUM_ALLOWED_SPECULATIVE_TASKS
PROPORTION_TOTAL_TASKS_SPECULATABLE
PROPORTION_RUNNING_TASKS_SPECULATABLE
AppContext context
TaskRuntimeEstimator estimator
AppContext
Map<JobId, Job> getAllJobs()
Job getJob(JobId jobID)
ClusterInfo getClusterInfo();
<<Interface>> Job
getTasks()
<<Interface>> Task
getAttempts()
<<Interface>> Speculator
AdaptiveSpeculator
AppContext context
TaskRuntimeEstimator estimator
DynamicThreshold()

Figure 4.3: Class diagram of the speculator component.

parameter (default value is 10), the *proportion_total_tasks_speculatable* parameter (default value is 0.01), and the *proportion_running_tasks_speculatable* parameter (default value is 0.1). The meaning of each parameter is relatively straight forward, for example, the latter two indicate the maximum number of new tasks the *Speculator* can create ($\max \#speculation = \min\{(proportion\_total\_tasks\_speculatable \times \#total\_tasks), (proportion\_running\_tasks\_speculatable \times \#running\_tasks)\}$).

If conditions are fulfilled for speculation, for example, the number of current speculative copy is smaller than the maximum speculatable value, the *Speculator* will call the *TaskRuntimeEstimator* class to get the estimated task durations. The *speculationValue* function then calculates the speculation value (SV) as:

$$SV = ECT - ERCT \tag{4.5}$$

where *ERCT* is short for Estimated Replacement Completion Time. Similar with ECT, *ERCT* is also generated by the *TaskRuntimeEstimator* class, but with normal task progress rather than the straggler behavior. *SV* is the time difference between the original slow task and the replacement task, representing the total time that can be saved from launching a replica for a specific task. The YARN scheduler calculates *SV* for each task, at each time when the speculator event is triggered, it then creates a replica for the task with the largest *SV*, by adding an attempt for this specific task into a waiting queue utilizing the *addSpeculativeAttempt* function.

### 4.4.2 Speculator Modification

Based on the above analysis of the default speculator in the YARN architecture, it is known that Hadoop already includes several reporting counters such as the progress info for each task attempt in *TaskAttemptStatus* class, and provides functions to calculate several key intermediate results such as the *estimatedRuntime*. The *TaskRuntimeEstimator* class provides an interface that can be inherited by different ECT calculating method, among which the *LegacyTaskRuntimeEstimator* is now been used by version 2.5.1.

Theoretically, the straightforward way of implementing our own algorithm is to add another *"AdaptiveSpeculator"* class similar with the current *DefaultSpeculator* that also inherit from the general *Speculator* interface, as shown in Figure 4.3 with the red dotted square. In reality, in order to minimize the changes made into this complex system, instead of creating a new class, we simply add a new function calculating $Th^t_{i,dyn}$ according to Equation 4.1 to replace the *"SpeculationValue"* function in the *DefaultSpeculator*. This way of implementation avoids potential disruption to other event handling components that rely on *DefaultSpeculator* internally. In other words, our algorithm does not introduce additional monitoring and computation overheads to the system. The only "extra work" performed is at $O(1)$ cost (function replacement), and all external APIs for the Speculator class remain unchanged.

For the $Th^t_{j,dyn}$ calculation, $P^t_j$ is straight forward since all Progress Score (PS) values are already recorded, and it is simple to get the average. As for the resource utilization adjustor $R^t$, we manage to get this information through the *AppContext* class. In the original system, the *RMContainerAllocator* is responsible for communicating with the Resource Manager (RM) to get the cluster information for the AM. In our implementation, two additional attributes are added in the *AppContext* class (namely "cluster capacity" and

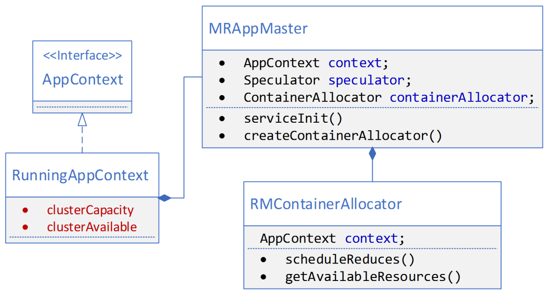


Figure 4.4: Modification to the *AppContext* class to get the resource values from RM

```
<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>
```

Figure 4.5: Parameter configuration example.

"cluster available" as shown in Figure 4.4) to record the available CPU and memory from *RMContainerAllocator*.

In addition, the parameters used in the calculation can be passed in through modifying the *mapred-site.xml* file just like the other YARN related parameters. An example is given in Figure 4.5. 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 as well, so that we can visualize key information including threshold value, replica number and job execution time, etc., for later evaluation purpose.

### 4.4.3 Experiment Setup

Our experiments are run on the OpenNebula platform [104], with a typical Virtual Machine (VM) configuration of 1GB memory, 1 virtual core with 2.34 GHz capacity and 10GB disk space on the potentially shared hard drive. The VM uses KVM virtualization software and runs an Ubuntu 12.04 x86_64 operating system. In all experiments, we configure the HDFS to maintain two data replicas for each chunk. The job types we run include Sort, WordCount, and Hive query (mainly Group By). The Apache Hive [141] is a data warehouse software residing in distributed storage using SQL. It is an open-source implementation [11] that can be deployed on top of the Hadoop cluster, and Hive queries (similar with SQL queries that reads, writes and manages datasets) will automatically be transferred into a Hadoop job. These three benchmarks were selected as they are frequently used for straggler evaluation, such as in the Google paper [46], in LATE [162], in Mantri [5] as well as in MCP [38]. In addition, a number of features of Sort make it a desirable benchmark [25]. For the Sort and the WordCount jobs, the input size is 10GB

Table 4.6: Experimental cluster configurations

| Number of VMs | 5 | 10 | 30 |
|---|---|---|---|
| Default VM | 3 | 15 | 22 |
| I/O injected VM | 0 | 1 | 1 |
| CPU injected VM | 0 | 1 | 2 |
| Mem injected VM | 1 | 1 | 2 |
| Combined VM | 1 | 2 | 3 |

and 5GB, respectively; while for the Hive query, the Group By is conducted on a table with more than 10 million rows.

For the cluster environment, we evaluate the modified system in three sizes: with 5 data nodes (VMs), 10 data nodes (VMs), and 30 data nodes (VMs). We injected faults and extreme resource contentions into the cluster to create a complex environment for experiments. An I/O intensive program, a memory intensive tool, and a CPU intensive program are deployed on specific VMs. The first program consists of mainly the "*dd*" and the "*rm*" command to create and delete files which take up most I/O throughput of the machine; the second tool intensively creates new array data to occupy memory; the last program continuously calculates the $\pi$ value until a certain accuracy (actually it can be programmed as an infinite loop to consume the CPU computation power).

The VM number settings are listed in Table 4.6, with "Default VM" representing VMs without injected faults. "I/O injected", "CPU injected" and "Mem injected" VM refer to VMs with a certain interference program running on it, while "Combined VM" indicates VMs with all three (I/O intensive, CPU intensive and memory intensive) applications deployed.

### 4.4.4 Experiment Results

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

**Execution Time Performance Improvement**

Three versions of Hadoop platform are deployed in order to compare their efficiencies in job completion time with straggler presence: Hadoop YARN with 1) the static and 2) the dynamic straggler threshold, and 3) Hadoop YARN without speculation. For any two platforms $P_A$ and $P_B$, we calculate the improvement of job response time $I_{AB}$ for $P_A$ versus $P_B$ by

$$I_{AB} = \frac{P_B(D_{J_j}) - P_A(D_{J_j})}{P_B(D_{J_j})}, \tag{4.6}$$

where $P_A(D_{J_j})$ and $P_B(D_{J_j})$ denote the duration of benchmark jobs running on platform $P_A$ and $P_B$, respectively. To note that, throughout the whole thesis, when referring to execution time improvements under two different platforms, we all follow this calculation.

Table 4.7 shows the results for all three workload types. Each experiment case is executed three times in order to get the average and standard deviation value. We keep the total input size constant throughout different cluster sizes. For example, for Sort, the files of 10GB in size have been sorted, which means that, in the 5 VM cluster, each node holds a 2GB file while for the 10 VM cluster, each node only holds 1GB.

Table 4.7: Results for different threshold performance

| Job Type | Cluster Size | Response Time (seconds) | | | | | |
|---|---|---|---|---|---|---|---|
| | | 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 |
| 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 |
| 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 |

It is shown that, the average job response time for WordCount using the dynamic straggler threshold is $87.34s$, while the static threshold and no speculation is $93.67s$ and $97.67s$, indicating an improvement of 6.76% and 10.58%, respectively. For Sort and Groupby, the average durations on corresponding platforms are $P_{dyn}(D_{J_{sort}}) = 686.67s$, $P_{sta}(D_{J_{sort}}) = 763.34s$, $P_{no\_spec}(D_{J_{sort}}) = 831s$, and $P_{dyn}(D_{J_{hive}}) = 61.33s$, $P_{sta}(D_{J_{hive}}) = 72s$, $P_{no\_spec}(D_{J_{hive}}) = 73.67s$. Figure 4.6 summarizes the improvement for the dynamic and static thresholds versus no speculation in different cluster sizes for different jobs.

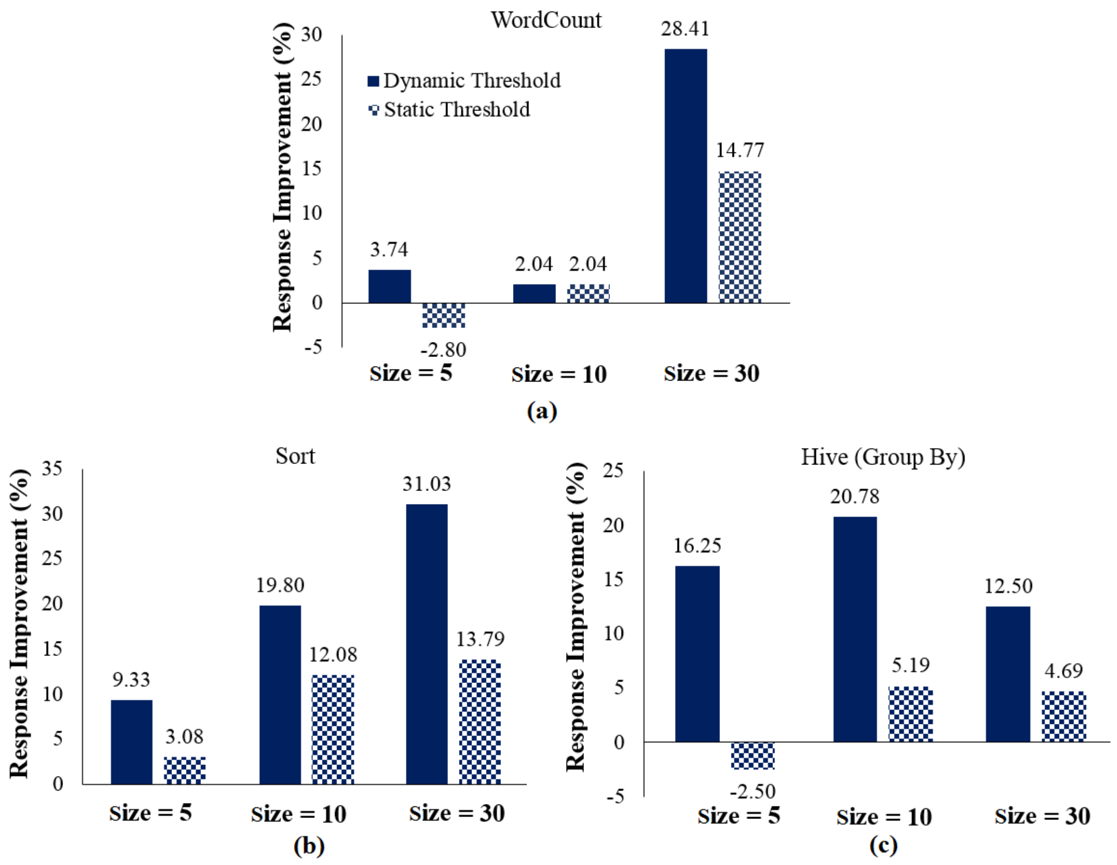


Figure 4.6: Job response time improvement of the dynamic threshold and the static threshold comparing to no speculation for jobs (a) WordCount, (b) Sort, and (c) Hive.

It is observable that, in the five node cluster, static speculation performs worse than no speculation with WordCount and Hive Groupby, leading to a negative improvement value. This is a result of the system experiencing an extremely high utilization state. Additional replication under this case further burdens the system instead of making performance improvements. The dynamic threshold, on the contrary, captures this contention information and makes appropriate adjustments, therefore obtaining a better performance.

In addition, we observe that the execution time performance improvement varies not only with cluster size, but also with different workload types. The improvement is more apparent for Sort 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 the static threshold. This is consistent with the algorithm design of creating more replicas to reduce job execution time when utilization is low. While in busy states, such as cluster sizes of 5 or 10, although the dynamic threshold identifies fewer replicas, it reduces the chance of a task to become a straggler due to contention, therefore still performs better than the static method.

To validate that the dynamic threshold really does make an improvement toward job completion time, independent *t*-tests between different methods are conducted. Table 4.8 details the significance value $p$. In the table, wd5 (ws5, and wn5) represents the $W$ordcount workload with the $D$ynamic straggler threshold ($S$tatic threshold, and $N$o speculation) running in a 5 node cluster, while sd5 and hd5 represents workload $S$ort and $H$ive under the same condition (dynamic threshold, and 5 node cluster). The null hypothesis of the tests are $mean(dyn) = mean(sta)$ and $mean(dyn) = mean(no\_spec)$, while the alternative hypothesis are $mean(dyn) < mean(sta)$ and $mean(dyn) < mean(no\_spec)$, representing dynamic threshold performs better than static threshold and no speculation as it gives a shorter response time.

Table 4.8: The *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 | $p = 0.013$ |

From the $T$-test results it is observable that, for Sort jobs, the difference in job execution performance 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 is further analyzed in the next subsection.

Table 4.9: 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% |

**Speculation Overhead and Effectiveness**

Besides the execution time performance, a further comparison regarding the speculation overhead is conducted between static and dynamic threshold methods as well, primarily measuring the number of replicas generated under each algorithm. The detailed results are listed in Table 4.9, from which we see that, when the cluster size is small, the dynamic threshold can save resources through creating fewer replicas compared to the static method. In contrast, the dynamic threshold in a larger cluster size generates more replicas as a result of trading resources for time to achieve better response performance.

In other words, in some cases, the dynamic threshold seems generated extra overhead, for example, when WordCount was running on the 30 VM environment, the proposed algorithm cost 58.3% more resources on creating replicas $((12 - 5)/12)$. However, because this overhead is incurred when the system is in its idle state, the influence it brought toward improving job response time is positive. This auto adjustment is important, especially for jobs with QoS timing constraints. In addition, the extra resources on replication number also help in improving the speculation effectiveness indicator.

We measure speculation effectiveness $E_{spec}$ by comparing the number of successful speculations, replicas that successfully over-pace the straggler and contribute to the response time improvement, with the total number of speculations launched, as shown in Equation 4.7.

$$E_{spec} = \frac{\#Spec_{success}}{\#Spec_{total}} \tag{4.7}$$

From the result listed in Table 4.9, the speculation success rate is much higher for Sort jobs compared to WordCount and Hive, which indicates that more replicas outpace the identified stragglers for Sort, and contribute towards improving the final job execution time. The reason behind this phenomenon is a limitation of speculative-based mechanism: when stragglers are caused by uneven input data sizes, the speculative copy will still suffer from the imbalanced workloads and ended up being killed, and this type of stragglers appears more often under WordCount and Hive compared with Sort.

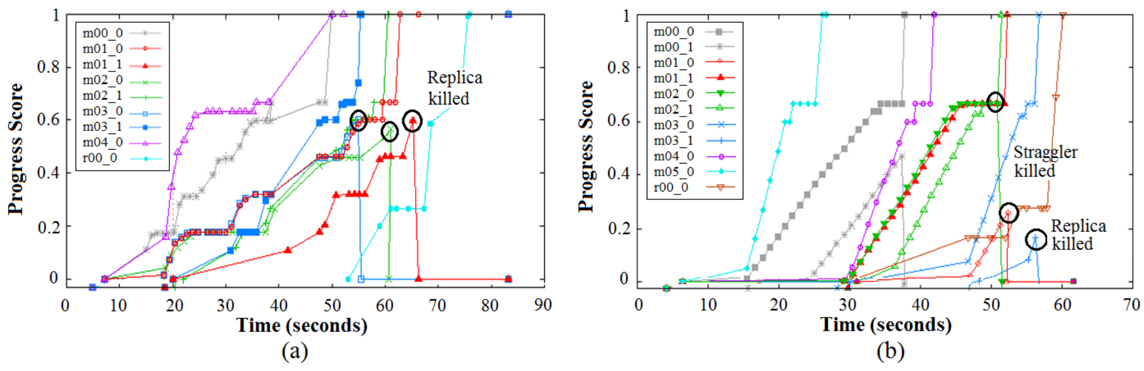


Figure 4.7: WordCount task progress in (a) no fault injection, (b) I/O contention injected cluster

The successful speculations for WordCount are mainly found when the testbed has been injected with resource interference, where stragglers are caused by contention reasons. Figure 4.7 shows a comparison example when running the WordCount job both using dynamic threshold in different testbed settings. Figure 4.7 (a) contains only default VMs and Figure 4.7 (b) has three VMs injected with I/O contention. Each color in the figure represents a task: the ones with suffix zero indicate original task attempts and suffix one means it is a speculative attempt for this specific task. In Figure 4.7 (a), all speculative replicas are killed because the stragglers in this case are caused by data skew, therefore they still finish before the "also lagged" speculation that process identical input. While in Figure 4.7 (b), two out of three speculations manage to over-pace the stragglers and succeed in the end, only task $m03$ has a failed speculation $m03\_1$. This limitation faced by speculative-based methods can be improved by complementary techniques that specifically target at skew-caused stragglers, such as the work detailed in chapter 6.

### Parameter Setting Sensitivity

The efficiency of the proposed algorithm is dependent on selecting the appropriate value for the configurable system parameters. This section studies how the threshold value

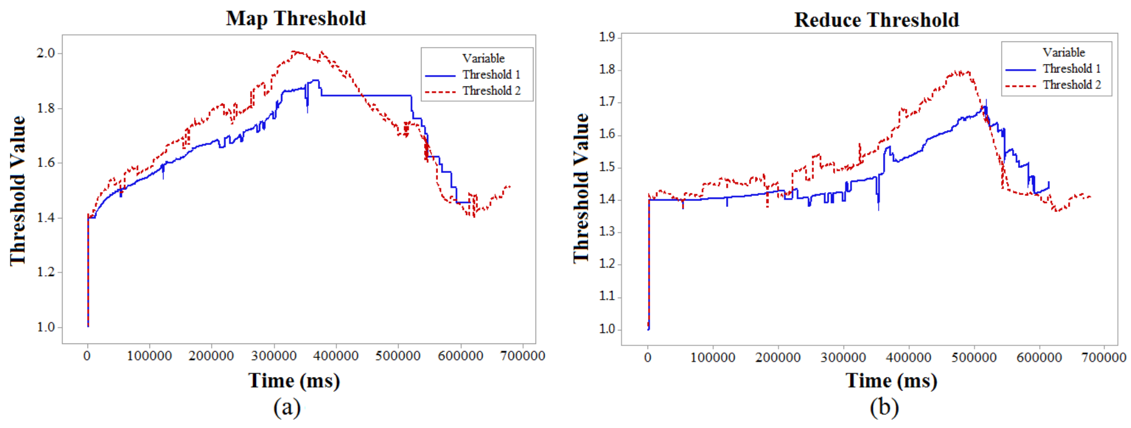


Figure 4.8: Threshold changing pattern for (a) Map and (b) Reduce tasks applying different parameter configurations

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 lead to different strictness of speculative replica creation. The system administrator can also impose different emphasis toward progress adjustor and resource adjustor. Figure 4.8 plots the changing pattern of two thresholds for Map and Reduce tasks of the Sort job as an example, with different parameter settings. The $\alpha$ and $\beta$ for threshold_1 are both 0.5, representing an equal weighting towards task progress phase and resource utilization level. $\mu$ is set with value 0.5, indicating the halfway progress point, and $\omega$ is set to be 0.7, indicating that any utilization below 70% will be treated as "idle". $\phi$ is not used in the experiments as the RM in YARN 2.5.2 only focuses on memory for now. According to the official website [66], CPU will be considered in later versions.

For threshold_2, $\alpha$ and $\beta$ are set to values of 0.4 and 0.6, respectively. This reveals that more emphasis has been put on the influence of resource utilization rather than progress. We decrease the value of $\omega$ from 0.7 to 0.6 for threshold_2 as a comparison, indicating a stricter utilization standard for additional speculation compared with threshold_1.

From the figure we notice that both curves exhibit a similar trend: the threshold value increases in the beginning due to the raising utilization caused by newly started tasks, then followed by a relatively flat period as a result of the neutral effect brought by the progress adjustor. Afterwards, a decreasing trend is observed 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 value increases again to avoid needless speculation.

Despite the similarity in general changing trend, some differences are witnessed between the two. For example, the turning point of the threshold value from increase to decrease for threshold_2 is slightly earlier than threshold_1. This is due to the utilization fall caused by task completion generates a larger effect than late phase increase for threshold_2 ($\beta_{threshold\_1} = \alpha_{threshold\_1}$ while $\beta_{threshold\_2} > \alpha_{threshold\_2}$). And the highest threshold value for threshold_2 is greater than threshold_1 due to it being more sensitive to utilization ($\omega_{threhold_2} < \omega_{threhold_1}$).

### 4.4.5 Simulation Results

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 [60] that can simulate Cloud datacenter operations such as the creation of jobs (comprising multiple tasks) onto a set of machines for execution.

SEED is implemented using the C#.Net 4.0 language (and compiled using the Mono framework for platform portability). The simulation is composed of several SEED instances, each providing the facility of simulating a system partition consisting of *nodes, links,* and *tasks*. The most basic instance can be compiled from two XML and text-based configuration files, with examples shown below:

*Example Network Specification*

```
<xml version = “1.0” encoding = “utf-16”>
<Network Clock_Port = “0”>
   <Nodes>
      <Node ID = “VN0_0.0.0.1” IP_Address = “0.0.0.1” />
      <Node ID = “VN0_0.0.0.2” IP_Address = “0.0.0.2” />
      <SwitchNode ID = “SW0_0.0.0.0” IP_Address = “0.0.0.0” />
   </Nodes>
   <Links>
      <Link ID = “Link 1” NodeA = “SW0_0.0.0.0” NodeB = “VN0_0.0.0.1” />
      <Link ID = “Link 2” NodeA = “SW0_0.0.0.0” NodeB = “VN0_0.0.0.2” />
   </Links>
</Network>
```

*Example Task Specification*

```
#Task Specification#
$ Task duration, Number of Tasks
150, 500
200, 10000
```

Different from other simulators, SEED provides an automated service-oriented process for simulation configuration and deployment, and supports execution across a heterogeneous distributed environment with no assumptions concerning the underlying hardware, as well as minimal user configuration. Importantly, SEED is also capable of managing event synchronization. A detailed comparison between SEED and some representative simulators is summarised in Table 4.10.

Table 4.10: Simulator comparison table

| Feature | Environment | Target domain | Distributed | Slowdown | Synchro | Task Types |
|---|---|---|---|---|---|---|
| SST+ gem5 | HPC | HPC | YES | N/A | N/A | Real |
| Emulab | HPC | Generic | YES | Vary with time slice | Config dependent | Config dependent |
| Graphite | Cluster | Multi core | YES | 41× | Lax | Real |
| YANS | Desktop | Network only | NO | Thread model dependent | Event-based | Models |
| Network CloudSim | Desktop | Generic | NO | Scalability issues | config dependent | Models |
| SEED | Desktop, Cluster | Generic | YES | Between 6× and 15× | Event-based | Models Real |

Besides the general SEED configuration, the design of our simulation adopts the following assumptions:

(1) All speculative replicas created are allocated with identical CPU and memory requirements with the straggler task [162];

(2) The speculative replicas need to re-execute the same work from the beginning, instead of continuing the unfinished work of the straggler from the detected point;

(3) The scheduler creates the replica immediately after a task has been defined as a straggler, and will schedule the replica as a normal task;

(4) The maximum resource capacity of the cluster remains the same, i.e. no addition / removal of server nodes during threshold calculation;

(5) Replicas can potentially become stragglers as well [5];

(6) Once a certain level is exceeded, higher resource utilization within a system leads to a higher probability of straggler occurrence [107].

For the second assumption, it is adopted because this "from the beginning" policy is used by almost every related literature out there. The reason for not check-pointing the original task may be that it requires additional status and complexity. So far, only one paper is observed within which the replica is starting from where the straggler is identified, and this Coworker paper [69] is discussed in Section 2.4.1.

When simulating the task progress, a simple model of a probabilistic function is adopted: tasks follow a linear progress with straggler probability defined in Equation 4.8. We assume the straggler probability is dependent on system utilization.

$$P(_{Straggler}) = \begin{cases} 0.1 & \text{if utilization} \in \text{(0,0.6]} \\ 0.2 & \text{if utilization} \in \text{(0.6,0.8]} \\ 0.3 & \text{if utilization} \in \text{(0.8,0.9]} \\ 0.4 & \text{if utilization} \in \text{(0.9,1)} \end{cases} \tag{4.8}$$

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 [5] and in chapter 3. Other sophisticated progress functions (namely the straggler probability function and the straggler tailing duration function) such as the Pareto and the Zipf distributions [36] can easily be implemented to replace current linear progress if needed.

We construct a simulated cluster with 100 servers and 500 tasks, and another environment with 800 servers and 10,000 tasks. The server nodes in the simulations are configured with 4096MB memory capacity, and tasks for the former cluster are configured with 512MB memory requirement while 256MB is set for the latter environment. For the dynamic threshold calculation algorithm, an equal weighting to progress and resource adjustor is adopted, and the value of 0.5 is assigned to both standard parameters. The detailed results are shown in Table 4.11.

Table 4.11: Simulation results for different thresholds

| Threshold Method | #Nodes | #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 it is observable that, in the cluster with 100 node, at the cost of an additional 2.6% $((72 - 59)/500)$ replica numbers , the dynamic threshold can reduce job execution time by a factor of 20.25% $((163 - 130)/163)$. And among all replicas launched, 66.67% managed to catch up with the corresponding stragglers using the dynamic threshold, while only 30.5% replicas are effective using the static threshold. In the case of 10,000 tasks, the statistics follow the same trend with the results from the previous environment: job execution time has been reduced by 23.94% $((213-162)/213)$ with 3.75% $((1861-1486)/10000)$ more replicas when adopting dynamic straggler threshold, with a 30.3% $(77.54\% - 47.24\%)$ improvement in speculation effectiveness.

## 4.5 Summary

The straggler threshold is a key concept used in current speculative methods, defining to what extent shall a slow task be identified as a straggler in the detection process. This section details an adaptive straggler threshold calculation method that dynamically adapts to different job types and system conditions to improve the efficiency of straggler mitigation. A brief summary that remarks the contribution of this chapter is given as follows.

- *The dynamic threshold proposed is effective in improving job completion time.* While some methods identify stragglers using a pre-defined static threshold for straggler identification, such as 50% greater than average execution, our approach allows for an adaptive threshold calculation that automatically captures job Quality of Service (QoS) timing requirement, task progress, and system resource utilization level. Experiment results demonstrate that, the dynamic technique can improve job completion by a factor up to 20% compared to the static method, while simulation results indicate the same trend, achieving an improvement up to 23.94% in a large scale environment.

- *Replica number trade-offs under different levels of resource utilization are made 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 utilization, the dynamic threshold method proactively generates more replicas to achieve a quicker response time.

- *An enhanced speculation success rate and effective quality assurance is achieved.* Not all replicas generated can successfully outpace the identified stragglers; to increase this percentage is important in improving speculation efficiency. The dynamic threshold is capable of choosing 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.

Refer back to the system model outlined in Section 3.5, this method provides the task-level detection for the Adaptive Speculator component.

# Chapter 5

# Server-level Prediction and Dynamic Blacklisting

Current datacenter environments often consist of thousands of server nodes with different physical capacities (including CPU, memory, disk, etc.), operational age, architecture, and performance [33]. These physical heterogeneities, as well as the dynamic resource utilization and multi-tenancy, result in diverse task execution performance for each node [106], which forms an important reason for the straggler occurrence. In this chapter, node execution performance modeling, ranking, and prediction are discussed, followed by a performance-aware dynamic blacklisting technique in order to avoid stragglers and improve speculation efficiency.

## 5.1 A Google Case Study

One straightforward idea of analyzing a node's ability in terms of parallel job completion is through the measurement of history data, specifically, the execution trace of the tasks that used to run on this machine. Therefore, the first attempt I try is to build a statistical model leveraging historical trace data when modeling node execution performance. A case study is conducted using the Google trace to demonstrate this statistical-based node performance modeling.

A filter is designed to get the target historical data out of the massive traces. Namely, in this analysis, I focus on MapReduce jobs - a representative job type that containing subtasks which exhibit similar completion time. For example in the Hadoop system, the Map tasks are automatically generated based on the input data determined by the Hadoop Distributed File System (HDFS) block size, running the same piece of Map function code. Therefore they normally have similar designed completion time. This equal designed duration is an important assumption supports the following analysis.

Details of the data semantics, formats, and schema are introduced in [120]. Among the trace there are four tables that directly relate to our research objective. The ***Machine_events*** table details server status (i.e. whether it has been added, removed, or modified within the cluster); the ***Job_events*** and ***Task_events*** tables record information pertaining to job/task status (un-submitted, pending, running, dead) expressed through recorded events (submit, schedule, kill, evict) at specific timestamps; the ***Task_usage*** table gives information of the start/end time of each individual task as well as the specific placement to servers. Three filtering conditions are applied in order to extract the MapReduce job:

(1) *Identify parallel jobs:* The first filtering condition is to identify parallel jobs according to its task numbers. A SQL query is constructed to select the jobIDs with more than two taskIDs submitted at the same time after studying the timestamps of job and task submissions.

(2) *Determine production jobs:* Tasks within the cluster are assigned with priorities ranging between 0 and 11 for lowest and highest scheduling priority indicated in the *Job_events* table. Production tasks including latency sensitive tasks are with priority from 2 to 9, monitoring is 10 to 11, and gratis tasks are 0 to 1. The latter two priority scales are excluded after this filter.

(3) *Extract MapReduce jobs:* Unlike the above two conditions that have explicit relating attributes, filtering out jobs that exhibit MapReduce characteristics requires two additional hypotheses. Firstly, the attributes "*job name*" and "*logical job name*" in the *Job_events* table are used, both of which are opaque base 64-encoded strings that have been hashed to hide sensitive information, shall be used. Unique job names are generated by automated tools to avoid conflicts, however, the job names generated by different executions of the same program will usually have the same logical

name. MapReduce is an example of this kind of application that frequently generates unique job names with identical logical names. Secondly, the *"username"* can assist towards identifying MapReduce jobs as well. Usernames in this trace represent services run on top of the Google cluster, and jobs executing under the same username are likely to be part of the same service. When a single program runs multiple jobs, such as master job and worker job spawned by the same MapReduce, those jobs will almost always run as the same user. Corresponding jobs are selected after running a SQL query that captures these two characteristics.

The filtered target MapReduce dataset is consisting of 92,848 jobs with 10,894,461 tasks. Importantly, since no biased selection towards node type is conducted, theoretically, the influence brought by eliminating additional tasks applies equally to all machine nodes. In other words, no imbalanced interference would be introduced after the task data filtering that could lead to unreliable node-related results. The statistical-based framework utilizing this target data to evaluate node performance is detailed in algorithm 5.

**ALGORITHM 5:** Workflow of the Statistical Based Node Analyzer

```
Inputs:
        {tasks}: A task set with "Task" elements
        {machines}: A machine set with "String" elements
1  while True do
2      set $\Omega = \emptyset$, set $\Psi = \emptyset$, set $\Gamma = \emptyset$;
3      for each task $\in \{tasks\}$ do
4          $\mu$ = NormalizedExecutionValue(task, {tasks});
5          $\omega = \langle$ task.tID, task.jID, task.mID, $\mu \rangle$;
6          $\Omega = \Omega \cup \{\omega\}$;
7      for each $\omega \in \Omega$ do
8          for each mID $\in \{machines\}$ do
9              if (mID == $\omega$.mID) then
10                 $\psi = \langle$ mID, $\omega.\mu \rangle$;
11         $\Psi = \Psi \cup \{\psi\}$;
12     for each mID $\in \{machines\}$ do
13         init CI = $\langle$ Low, High $\rangle$;
14         CI = MachineExecutionPerformance(mID, $\Psi$);
15         $\gamma = \langle$ mID, CI $\rangle$;
16         $\Gamma = \Gamma \cup \{\gamma\}$;
17     set $\Delta$ = IntervalBasedRank($\Gamma$);
18     Sleep(TimeWindow);
```

For algorithm inputs, the "*Task*" element is a user defined data structure that contains attributes of *tID*, *jID*, *mID*, and *duration*. The notion of *tID* is short for taskID, and is a unique identification string. Similary, *jID* is short for jobID, indicating which job this task belongs to, and *mID* is short for machineID, showing the machine this task runs on. The notion of *duration* represents the execution time of the task.

Key phases of the statistical-based method include following procedures: calculating normalized execution value for tasks, building up machine execution performance model, calculate target indicators, and interval based ranking. The first step captures key features to represent node execution performance while the second and the third steps build up the distribution model and generate statistical attributes such as Confidence Interval (CI). The last procedure ranks the nodes according to the attributes. The following subsections introduce the algorithms developed for each step.

### 5.1.1 Normalized Task Execution

When leveraging historical job execution behavior to analysis node performance, raw task durations cannot be used directly to generate comparable results because there are multiple workloads with different designed length co-exist in the Cloud environments. For example, $T_{A1}$ with designed duration of 10 seconds is assigned to $M_1$ while $T_{B1}$ with the designed length of 100 seconds is assigned to $M_2$, and both tasks finish after 100 seconds. If the raw duration is used to do the analysis, then the conclusion of equal node performance would be generated because the tasks exhibit equal duration after running on these two nodes. However, $M_1$ actually performs much worse than $M_2$, extended $T_{A1}$ 10 times compared with its normal execution.

In order to solve this problem, normalized execution values of tasks are used to construct the probabilistic model of nodes in order to ascertain the likelihood of straggler occurrence. The value is calculated following Equation 5.1 using Z-score normalization

$$\widetilde{D_{T_{ji}}} = \frac{D_{T_{ji}} - \overline{D_{J_j}}}{\sigma_{J_j}} \tag{5.1}$$

where $\overline{D_{J_j}} = avg\{D_{T_{ji}}\}, T_{ji} \in J_j$. Through this normalization, the duration variation brought by job types can be eliminated. And because of the assumption that tasks from the same job have similar designed duration, $\widetilde{D_{T_{ji}}}$ reveals the relative speed of $T_{ji}$ compared to its sibling tasks. A positive $\widetilde{D_{T_{ji}}}$ value represents a slower execution because the duration of $T_{ji}$ is larger than the job average, and the increment of the positive $\widetilde{D_{T_{ji}}}$ indicates an aggravated straggler behavior $T_{ji}$ exhibits. Vice versa, a negative $\widetilde{D_{T_{ji}}}$ indicates a shorter response, and the smaller the negative value, the quicker $T_{ji}$ performs. The detailed calculation procedure is given in algorithm 6, with *targetT* representing the task that needs the normalized value calculation.

**ALGORITHM 6:** Normalized Execution Value

```
Inputs:
        targetT: The target task for normalized value calculation
        {tasks}: A task set with "Task" elements
Output:
        NormalizedValue: The calculated normalized value
1  double AvgJobD = 0, int tNum = 0, double StDevD = 0;
2  for each task ∈ {tasks} do
3      if (targetT.jID == task.jID) then
4          tNum++;
5          AvgJobD += task.duration;
6  AvgJobD /= tNum;
7  for each task ∈ {tasks} do
8      if (targetT.jID == task.jID) then
9          StDevD += math.pow((task.duration - AvgJobD),2);
10 StDevD = math.sqrt(StDevD / tNum);
11 NormalizedValue = (targetT.duration - AvgJobD) / StDevD;
12 return NormalizedValue;
```

The key idea supporting the statistical analysis is that, for each machine $M_k$ within the cluster, the collected normalized execution values of all tasks that are assigned to it within a certain time period can be used to analyze its execution ability. If the majority of tasks assigned to $M_k$ are with positive $\widetilde{D_{T_{ji}}}$s, which indicate slower execution compared with their own average job duration, we say that $M_k$ encounters a poor execution performance. In contrast, intensive negative $\widetilde{D_{T_{ji}}}$ values observed from $M_k$ demonstrate a good execution performance, because tasks assigned to it always tend to finish quicker than the other sibling tasks from the same job.

### 5.1.2 Distribution Fit for Node Execution Performance

It may be a coincidence if one task performs slowly on a certain node, or we can infer task-related reasons instead of node-related causes for this slowness. However, the statistical distribution of all tasks' behavior observed on this node tells a different story. Therefore, in order to conduct the node performance analysis, the distribution fit is first conducted.

There exist numerous Goodness of Fit (GoF) tests designed for different data characteristics, including chi-square, Anderson-Darling (AD) and Kolmogorov-Smirnov (KS) [89][119]. Normally a GoF test generates a *p-value*, representing the probability that, it is false to reject the assumption that the sample data is from a certain distribution. In other words, assume the null hypothesis is that the sample data indeed coming from a certain target distribution, then any GoF with a *p-value* lower than a certain significance level (usually 0.05) can be rejected. In our analysis, AD test is adopted as it places greater emphasis towards tailing data distribution through a weight function [44]. Under the AD test, we focus on the *AD-value* returned rather than the general *p-value*.

Minitab [95], a statistical software similar to SPSS, is used in this case study. In Minitab, a smaller *AD-value* indicates a larger chance that the sample data is coming from a target distribution. Figure 5.1 shows an example result of distribution fit for the node with ID 4820223869 in the Google data. In this case, altogether seven different distributions are tested, including 3-Parameter Lognormal (*3P-LN*), Normal (*N*), 2-Parameter Exponential (*2P-E*), 3-Parameter Weibull (*3P-W*), 3-Parameter Gamma (*3P-G*), Loglogistic (*L*) and

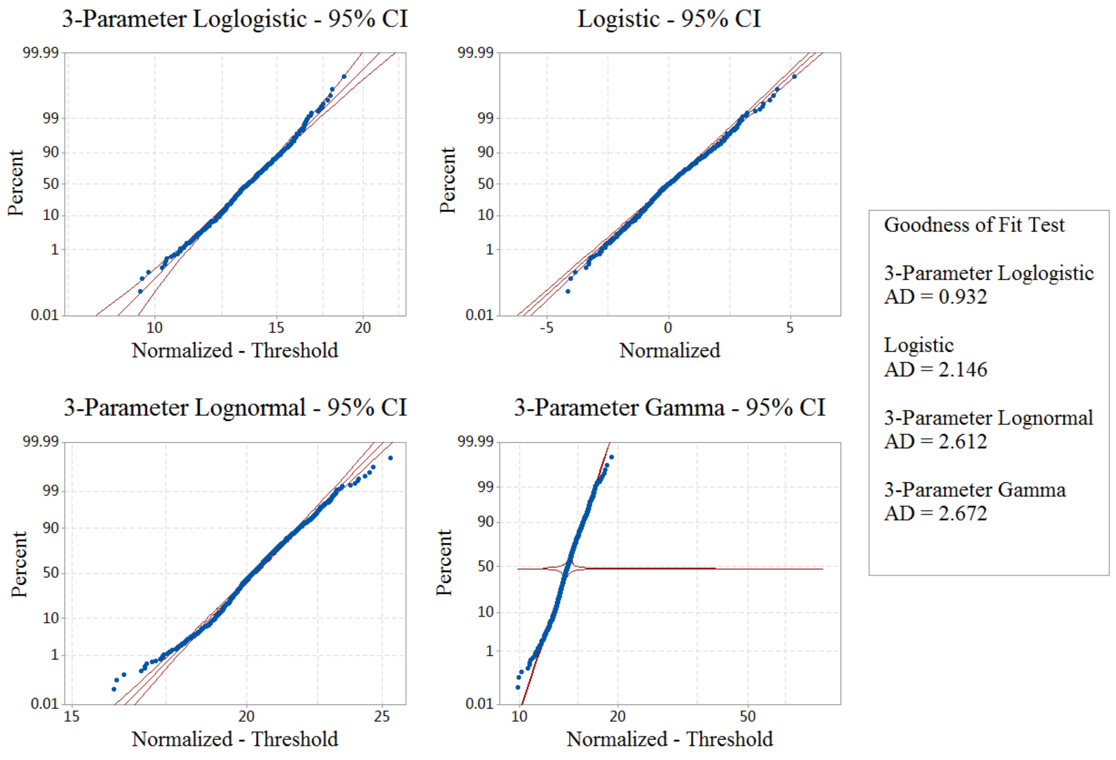


Figure 5.1: Top four best fitting distribution for example node $M_{4820223869}$

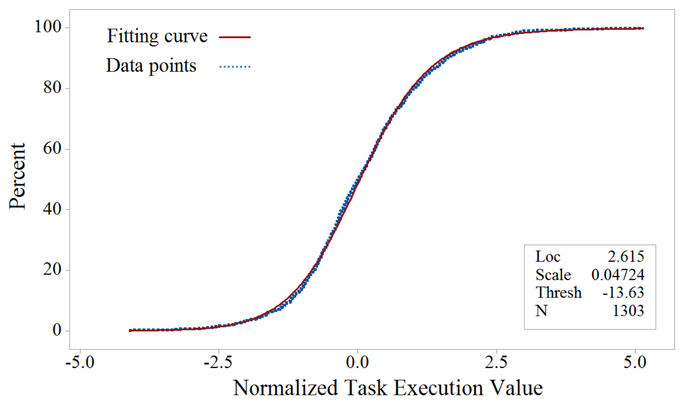


Figure 5.2: The CDF fitting of $M_{4820223869}$ using 3-Parameter Loglogistic distribution

3-parameter Loglogistic (*3P-LL*). The top four best fits are listed in the figure, among which the *3P-LL* distribution represents the best accuracy due to a lower *AD-value*. The detailed Cumulative Distribution Function (CDF) fitting is given in Figure 5.2.

As there are over 12,500 server nodes within the Google cluster, it is beneficial to perform sampling first in order to conduct the in-depth analysis of nodes that accurately reflects the general characteristics of the whole cluster. A subset containing 132 nodes is generated after sampling. The number of 132 is chosen because it is the minimal number that can retain a 5% margin of error to the whole population calculated by the Minitab tool through its sample size calculation function.

In addition, due to the fact that there are mainly four types of servers within the Google system (refer to the shaded types in Table 3.1), with each type reflecting different physical capacities in the dimension of CPU and RAM. A random selection is made within each

Table 5.1: Google node distribution GoF result

| | Distribution | Number | Percentage |
|---|---|---|---|
| Best Fit | 3P-LL | 112 | 84.85% |
| | 3P-LN | 11 | 8.33% |
| | 3P-G | 6 | 4.54% |
| | L | 2 | 1.52% |
| | 3P-W | 1 | 0.76% |
| Second Fit | L | 86 | 65.15% |
| | 3P-LL | 18 | 13.64% |
| | 3P-LN | 17 | 12.88% |
| | 3P-G | 11 | 8.33% |

server type in the sampling process to generate the corresponding number of nodes. This is to make sure that the final sample set consists of nodes that remain the same server type proportion (53.46%, 30.76%, 7.93%, and 6.34%) with the whole population.

Table 5.1 summaries the distribution fitting result for the sampled machines. From the table it is observable that, most of the distributions fit into the 3P-LL model, with 84.85% ranks 3P-LL first and 98.29% include 3P-LL as top two best fits. This finding helps in the following up process of calculating statistical attributes of the node.

## 5.1.3 Target Indicator Choice

The statistical properties derived from the distribution can be used to infer the susceptibility of nodes to the straggler behavior. For example, Figure 5.3 lists the distributions of four nodes from the 132-node sample set. The two nodes with ID (a) 672206 and (b) 554297904 experiences approximately equal positive and negative values, indicating a balanced node performance regarding its ability of executing tasks; node (c)

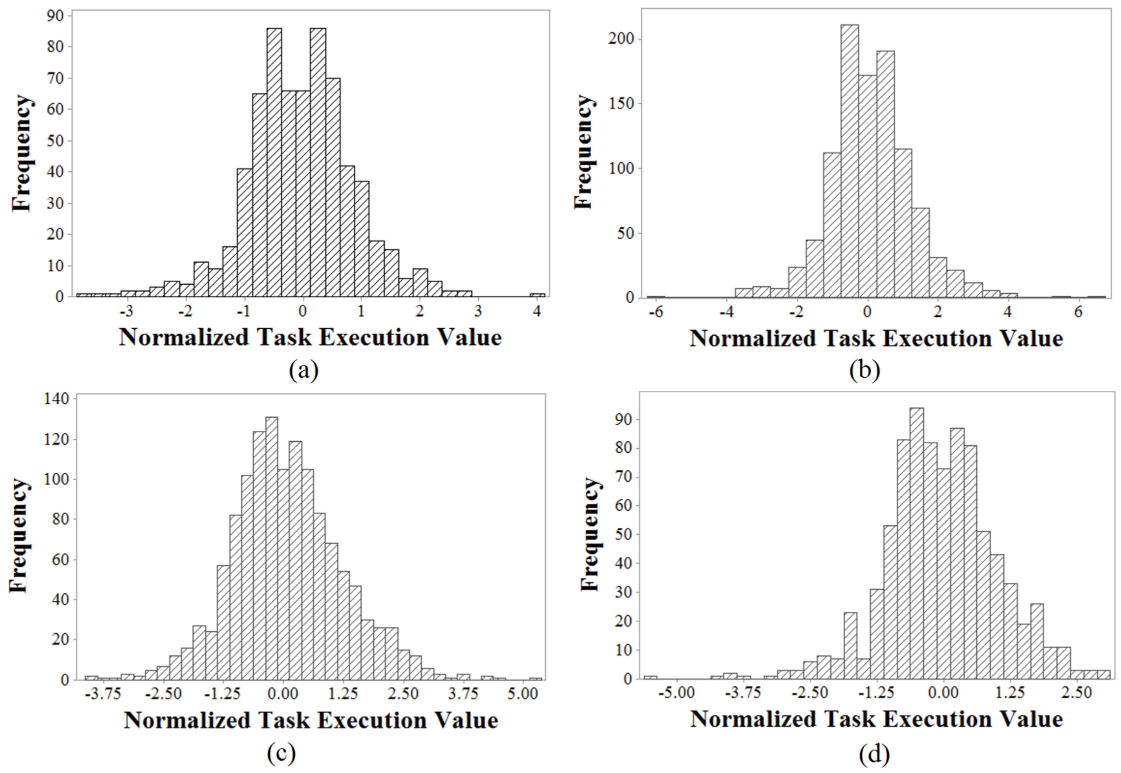


Figure 5.3: Normalized value frequency for machine (a) $M_{672206}$, (b) $M_{554297904}$, (c) $M_{4820223869}$, and (d) $M_{257336015}$ from the Google system

4820223869 has more positive values, indicating a slightly weak performance, while node (d) 257336015 has more negative values, representing a better performance.

Besides the number of positive/negative values as described in the above example, the other indicators that can be used to capture the node performance behavior include the mean value, the standard deviation, the CI, the quantile points, and the extreme value possibility, and etc., each reflects different evaluation objective. For example, if the average $\widetilde{D_{T_{ji}}}$ of all the historical tasks assigned to node $M_1$ is 2, we can infer that $M_1$ is a weak performance node because most tasks assigned on $M_1$ are stragglers in their own job, characterized by $2 * \sigma_{J_j}$ times slower than their own average duration $\overline{D_{J_j}}$. And it is reasonable to assume that, later tasks which are about to be assigned on $M_1$ in the near future will have a possible relative speed around $2 * \sigma_j$ times slower as well. Table 5.2 lists representative attributes and their corresponding meanings for reference.

For example, under cases when CI is chosen as the indicator of interest, it provides an insight that, for all tasks assigned to this node, there is a confidence (e.g. 95%) to believe their normalized duration will fall within a specified interval. For other examples, standard deviation describes the stability of the node execution performance, while extreme value possibility represents the task straggler occurrence probability.

To be more specific, after we get the most suited distribution fit, normally 3P-LL, we can then calculate the statistical attributes such as the 95% quartile value and the percentage for stragglers (normalized value exceeds a pre-defined threshold). Figure 5.4 details the number distribution of (a) 95% quantile value and of (b) probability for normalized value larger than 1 using the 132-node sample set, from which a rough knowledge about node execution performance can be obtained: for Figure 5.4 (a), the four nodes with 95% quantile larger than 2.3 indicate weak performance while for Figure 5.4 (b), the three machines with straggler rate larger than 25% are the weakest ones. This insight is necessary when determining the optimal placement of tasks onto nodes under the presence of stragglers.

Table 5.2: Node performance indicator candidates and corresponding meanings

| Indicator | Meaning |
|---|---|
| Mean | The possible normalized execution value for tasks assigned onto this node |
| StdDev | The normalized task execution value on this specific machine is stable or random |
| CI | The possible normalized execution value assigned will between a certain interval |
| Extreme % | The task straggler possibility for this machine |
| Quantile | Describes the normalized value for most tasks been assigned onto that specific node |

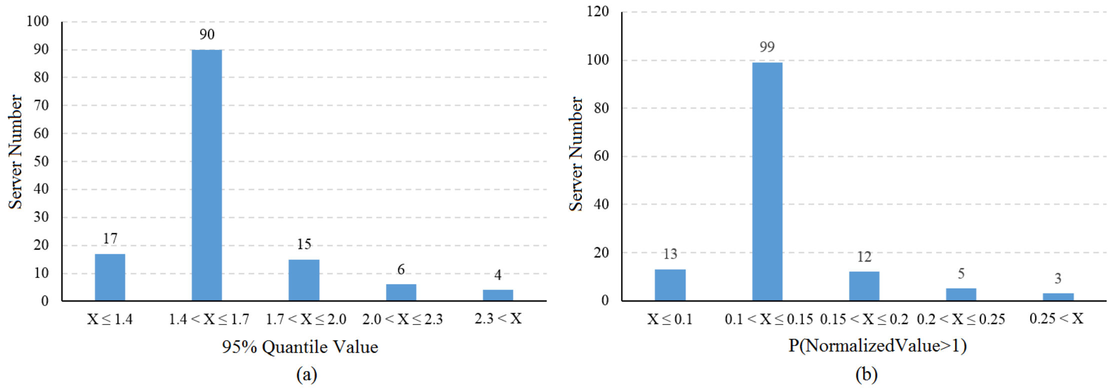


Figure 5.4: Ranking examples for the Google server using different indicator

An interesting finding we observe is that, higher node capacity does not always result in better execution performance, and this contradicts with the assumption adopted by some of the current literature. For example, when exploring the node performance of the aforementioned machine sample set, the node execution performance results are grouped in accordance with the four major server types in Google. Surprisingly, the server type with a stronger capacity tends to exhibit worse execution performance as shown in Figure 5.5.

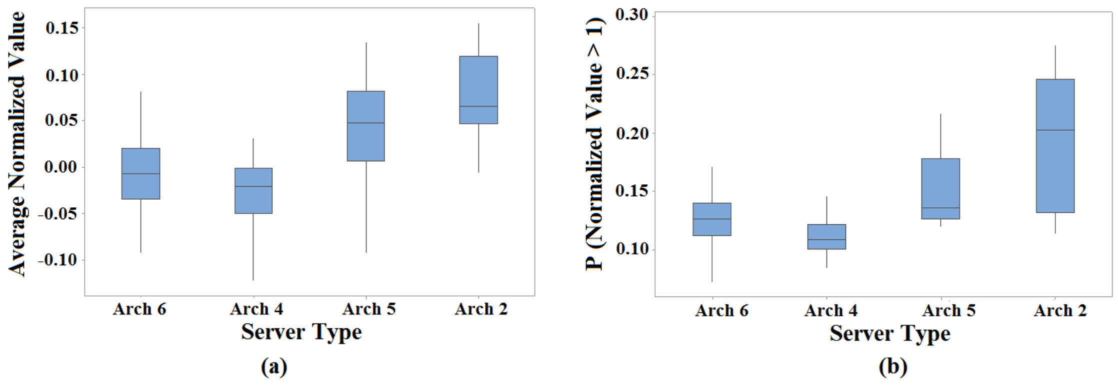
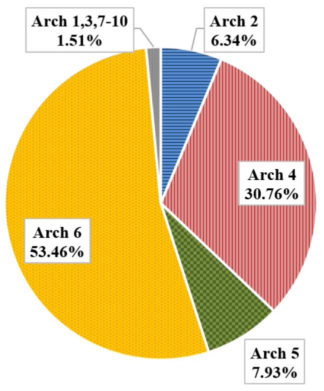


Figure 5.5: Boxplot of (a) mean normalized value and (b) extreme value possibility for each group in the Google cluster

Figure 5.5(a) and Figure 5.5(b) are the boxplots of node performance when mean value and extreme value possibility are chosen as the target indicator, respectively. It is observable that, nodes belong to server type *Arch2* tend to exhibit larger normalized execution values, while values for *Arch4* nodes fall below zero, signifying most tasks run on this server category execute quicker than their average. That is to say, while the server node capacities are $C_{Arch4} < C_{Arch6} < C_{Arch5} < C_{Arch2}$ (detailed capacity of each server type is listed in Table 3.1), the overall nodes execution performance actually exhibit an opposite ranking, with $Arch4 > Arch6 > Arch5 > Arch2$.

There are many possible reasons behind this phenomenon, for example, it may because of that, servers with larger capacity tend to receive more task submissions, hence the larger chance of straggler occurrence. However, a strong correlation between server population and the number of tasks submitted to a specific server type is observed as shown in Figure 5.6 (a) and (b), and this strong correlation is also statistically proved by a high pearson correlation coefficient value (0.994). In other words, the scheduling algorithm deployed within the Google datacenter appears to be load balanced, equally assigns tasks across all servers. Therefore, the reason behind the weaker node performance of the "stronger nodes" is not the higher contention/utilization as guessed.



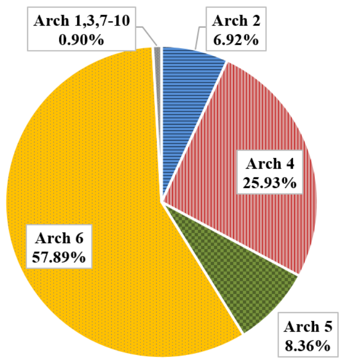


Figure 5.6: Proportions of (a) server population, and (b) task submission per server type

Due to the lack of information from the data provided, we cannot come up with further explanations of why this behavior is observed in the Google system. However, this does prove that node execution performance is not purely dependent on its physical capacity, and this conclusion is consistent with the findings discussed in Section 3.2.3. Therefore, when ranking the nodes with their performance, it is important to use the real historical data to do the modeling first.

### 5.1.4 Ranking and Weak Node Identification

In respect of dealing with the straggler problem, it is important to identify the weak performance nodes and to avoid scheduling tasks to such nodes. A ranking process is needed in order to achieve this goal. Based on above modeling, if value-type indicators are chosen to represent the node execution performance such as the mean and the quantile value, the ranking is relatively straightforward. However, the system administrator has to pre-define the number of $k$ in order to get the top $k$ worst nodes, in which case subjective factors

**ALGORITHM 7:** Interval Based Ranking

```
Inputs:
        {machines}: A machine set of "Machine" elements
Output:
        {mID}: The machine ID set indicating weakest $n$ machines
1  for each machine $\in$ {machines} do
2      init machine.outEdge = $\emptyset$, machine.inEdge = $\emptyset$;
3  for each m1 $\in$ {machines} do
4      for each m2 $\in$ {machines} do
5          if (m1.mID != m2.mID) then
6              if (m1.CI.High $\leqslant$ m2.CI.Low) then
7                  m1.outEdge = m1.outEdge $\cup$ {m2.mID };
8                  m2.inEdge = m2.inEdge $\cup$ {m1.mID };
9  {mID } = $\emptyset$; for each m $\in$ {machines} do
10     if (m.outEdge == $\emptyset$) then
11         {mID } = {mID } $\cup$ {m.mID };
12 return {mID};
```

may be brought in and undermine scheduling / speculation performance. One option that can automatically generate the suitable number of weakest nodes is through a graph-based ranking algorithm utilizing interval-type indicators such as the CI.

The graph-based ranking is detailed in algorithm 7, modified based on P-Cores [21] after constructing a Directed Acyclic Graph (DAG). The dots in the DAG represent servers within the cluster. $[L_{M_1}, H_{M_1}]$ represents the CI of the execution performance of node $M_1$, with $L$ and $H$ represent the lower boundary and the higher boundary, respectiverly. A sequence edge is constructed from $M_1$ to $M_2$ only when the condition of $L_{M_2} \geq H_{M_1}$ stands. CI overlaps will not lead to an edge under this rule. Figure 5.7 shows an example of such DAG following above edge construction principle.

From this design it can be inferred that, if a machine is with no outward edge, it is the

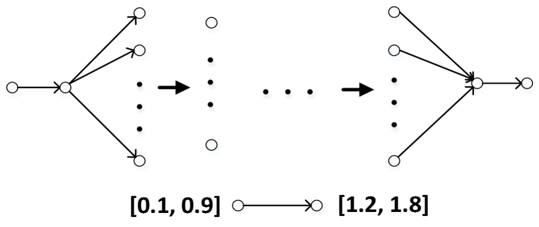


Figure 5.7: An DAG edge example

current weakest node because its CI is larger than the others, indicating a frequent straggler behavior. The input "*Machine*" element in algorithm 7 is a user-defined structure that contains attributes of $mID$ and its performance CI.

The algorithm functions in a way that, dots with no outward edge repeatedly being removed from the DAG along with all the related inward edges, until there are no remaining dots. The iteration time on which the node is been deleted is recorded, and subscribed as the *level* this machine should be classified to. In other words, the level zero nodes represent the ones that are removed at the first iteration, and are the weakest ones among all because they have the largest CI value according to the rule described in Figure 5.7.

In algorithm 7, the default ranking policy which returning all level zero nodes is demonstrated, termed as the *P-Cores without number* policy. After feeding the filtered MapReduce data into the algorithm, the statistical-based modeling classifies Google nodes into five levels depending on their performance of executing tasks. In total, there are 105 nodes identified as the level zero ones (0.83% of total population) in this case study. The numbers for other levels are: 1,772 (14.08%) for level 1; 7,265 (57.74%) for level 2; 3,386(26.91%) for level 3; and 55(0.44%) for level 4.

To note that, the *P-Cores without number* policy works well for large-scale clusters. However, when the cluster size is small, this policy has a risk of hindering system capacity due to no control of the exact machine number classified as unsuitable for launching task. For example, under extreme cases when all performance CIs calculated are overlapped with each other, making the DAG contains no edges but only scattered dots, the *P-Cores without number* policy will rank all nodes as level zero ones, leaving all available machines classified as weak ones.

In order to solve this problem, the *P-Cores with number* policy is introduced as an alternative complement, which generates the top $k$ weakest nodes. It uses standard deviation as the vice indicator to help with the ranking procedure. When the number of level zero nodes surpasses a certain threshold, we further rank them in descending order of the StDev. The heuristic here is that, nodes with weaker performance will result in a more random task execution behavior, characterized by a larger $\widetilde{D_{T_{ji}}}$ StDev. Through this alternative policy, the user can control the desired number of weak nodes returned.

To evaluate the generality of this statistical analysis scheme, another example of using it in an experimental environment consists of three Virtual Machine (VM) clusters is given. The detailed introduction of the involved clusters are given in Section 5.3.2. Within each

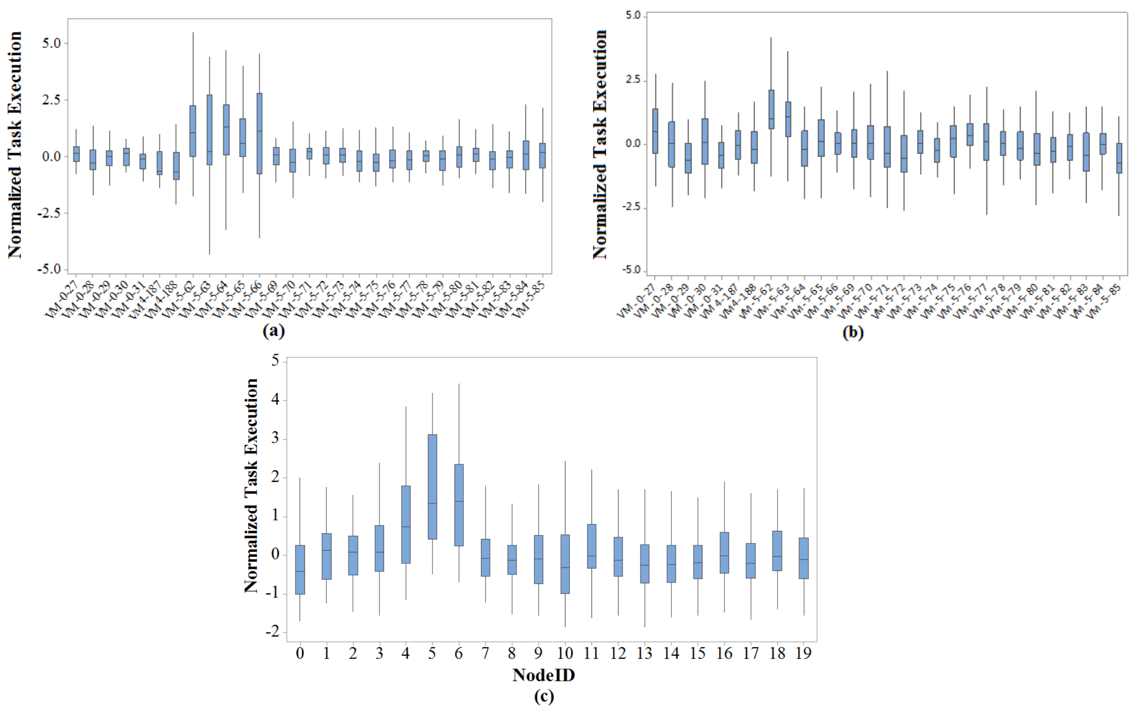


Figure 5.8: The node performance ranking within (a) the OpenNebula cluster; (b) the OpenNebula 2 cluster; and (c) the ExoGeni cluster

cluster, the nodes are heterogeneous in performance due to injected interferences described in Table 5.6. The log data is generated running WordCount and Sort jobs, and the box plot of all tasks' normalized execution value per node for the corresponding clusters after an hour run is shown in Figure 5.8, from which insights of the node performance ranking can be observed.

From the results we see that, in the first OpenNebula cluster, node 10.1.5.62, 10.1.5.63, 10.1.5.64, 10.1.5.65, 10.1.5.66 are with obvious higher performance CI (observed from the normalized value distribution, the average and the standard deviation are both larger than the other nodes). This result precisely flagged out all the nodes with memory interference program running on top. The results in the ExoGENI cluster and the OpenNebula 2 cluster exhibit similar trend: for the former, node4, node5, and node6 are identified as weak performance nodes, while for the latter, the node of 10.1.5.62 and 10.1.5.63 are flagged out. This result is consistent with the real node ranking in the cluster setup (refer to Section 5.3.2), that all the nodes with injected memory fault are successfully ranked as the worst performed ones.

In addition, in Figure 5.8 (b), the nodes with injected CPU fault (namely node1, node2, and node3) are exhibiting the second largest performance CI, ranked weaker than the rest within the cluster. However in Figure 5.8 (a) and (c), this observation is not as clear: for example, in Figure 5.8 (a), node 10.1.0.28, which is injected with CPU contention, performs better than node 10.1.5.71, which is a normally configured node, during the one-hour experiment period. This reveals a fact that, for the YARN system with Word-Count and Sort workloads, the contention for memory is the major cause of the straggler behavior rather than the contention for CPU.

To conclude, the node ranking results demonstrated from this example again shows the ability of the proposed analysis in detecting weakly performed nodes. This result, combined with the Google case study, proved that it is reasonable to model node execution performance through the statistical analysis based on historical behavior of tasks running on the node, especially through the distribution of the normalized tasks' durations. When the analysis is conducted in Cloud-hosted environments, the performance ranking would be VM's execution performance ranking, in which case, VM is mapped to the so-called server node in this type of virtualized system, because the tasks are assigned to VMs and the VM resource is what the cluster scheduler cares about. If the majority of weakly performing VMs are located on the same physical machine, that would generate useful hint for physical node analysis, which is another interesting topic that falls outside the boundary of this research.

## 5.2 Machine Learning based Prediction

Despite the feasibility, one challenge encountered by the statistical modeling is its ability in capturing the most up-to-date node performance when this attribute changes dynamically, and making predictions based on historical patterns. Figure 5.9 (a) shows the data analytics result of the OpenCloud machine behaviors within a 20 day period. The straggler rate for each node is quite balanced: every one of them faces a 3% to 4% straggler rate. However, if the performance is split into daily basis as shown in Figure 5.9 (b), a quite different trend is observed. Each line in the graph represents a machine node in the cluster, on several days such as the $3^{rd}$ to the $7^{th}$ day, the straggler rate across different machines is relatively similar; while for other days such as the $8^{th}$ to the $11^{th}$ day, the performance of each machine varies a lot. The weakest performance reaches almost $30\%$ straggler rate while others remain less than $5\%$. Similar behavior is also seen in the attribute of speculation failure rate as shown in Figure 5.9 (c) and (d), which reflect the dynamic nature of speculation efficiency.

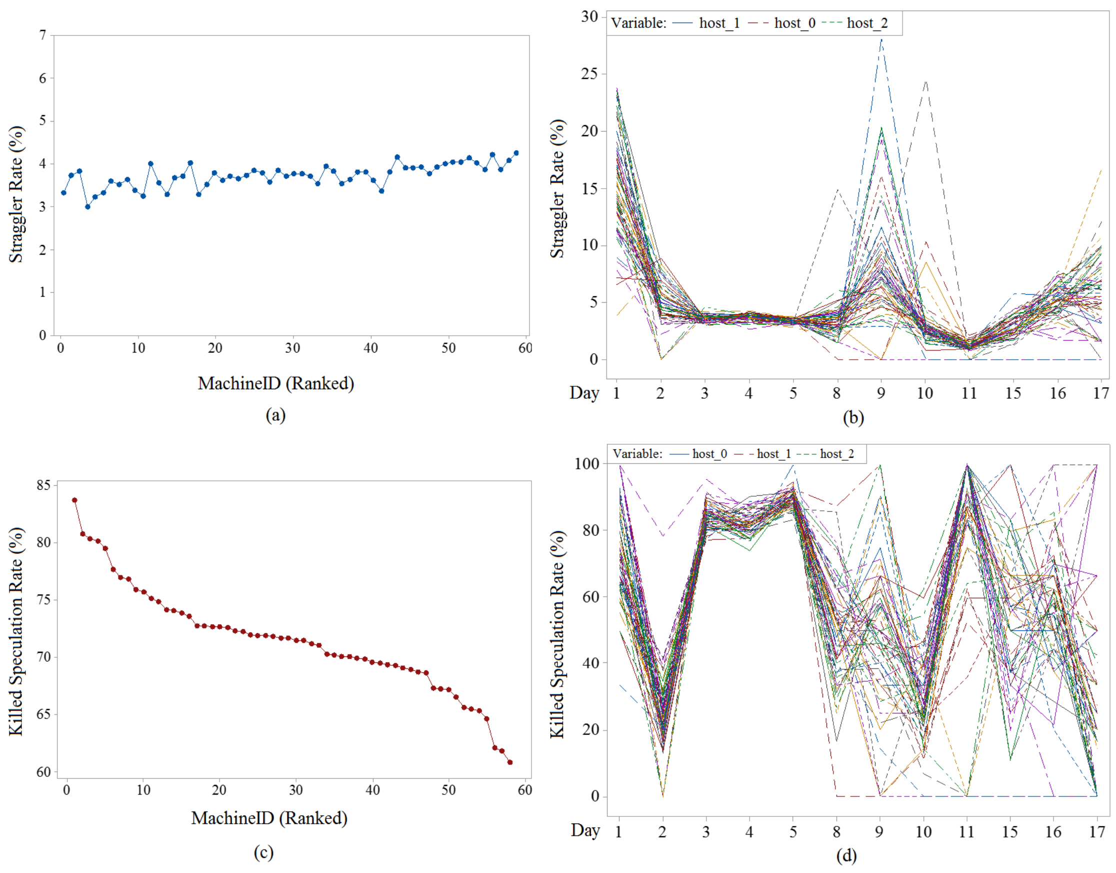


Figure 5.9: Straggler rate per node (a) in a 20-day period; (b) per day changing trend; and killed speculation rate per node (c) in a 20-day period; (d) per day changing trend. Each line in (b) and (d) represents a node, the legend only gives three examples due to the space

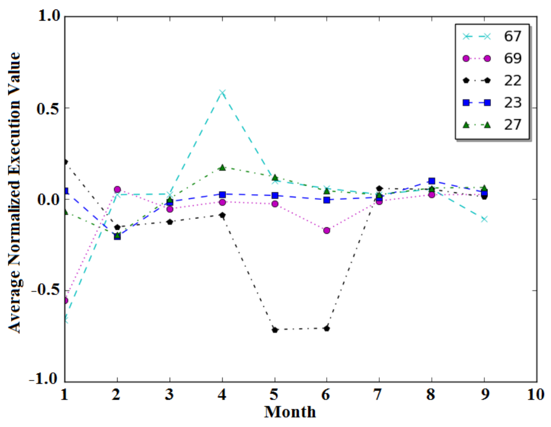


Figure 5.10: Node execution performance changing trend

Figure 5.10 illustrates a clearer example with five nodes from the OpenCloud system, using $\widetilde{D_{T_{ji}}}$ to reflect the quickness or slowness of tasks derived from different nodes, adopting average value as the performance indicator. The $\widetilde{D_{T_{ji}}}$s from tasks assigned to each node within each month is summarized, and the results cover 10 months in total. Again, each line in the graph represents a machine node, with y-axis being the $\widetilde{D_{T_{ji}}}$ average for the specific month. It is observable that nodes tend to exhibit diverse performance in different months. For instance, machine $M_{67}$ outperforms the others in the $9^{th}$ month (though exhibiting a smaller negative average $\widetilde{D_{T_{ji}}}$) after it suffers in the $4^{th}$ month. This is consistent with previous observations shown in Figure 3.9: the weakest nodes (with a significantly larger number of stragglers) change over time, revealing a dynamic nature of straggler occurrence.

It is important to model the evolutionary pattern of node performance and predict possible behavior in the near future due to this dynamic attribute. Machine Learning (ML) techniques can be used to address such challenge. In this thesis, a Machine Learning based Node Analyzer (ML-NA) is proposed accordingly. The following sections introduce the ML-NA design in respects of feature selection, labeling, classification, and prediction.

### 5.2.1 Feature Selection

The first challenge of the ML based analyzer is to select the proper features to describe a machine in the aspect of its execution performance, considering the fact that node performance is typically influenced by multiple factors. There are mainly three key feature groups that are taken into consideration in the ML-NA design: the statistical attributes of the normalized task durations per node, task number per node, and timing attributes.

- *Statistical attributes*: As discussed in Section 5.1, node execution performance can be reflected by the statistical attributes of all tasks running on it within a certain time period. In the following analysis, two statistical attributes are used when featuring a node: the average and the standard deviation of all $\widetilde{D_{T_{ji}}}$s pertaining to each node. The former one sets a rough performance standard while the latter reflects the fluctuation range of the node performance, showing a stable or random possibility of $\widetilde{D_{T_{ji}}}$ in the certain node.
- *Task number*: Apart from the statistical attributes derived from the $\widetilde{D_{T_{ji}}}$ distribution, task number per node is the other important feature that we use to describe the node

performance. It implies the node's contention state and reflects the impact of such contention toward job execution rapidness. In addition, the normalized task number compared with all the other machine nodes in the cluster is used rather than the raw task number. This is because we intend to constrain the selected features into a similar range, which lays the foundation for further operations such as clustering.

- *Timing attributes*: Considering the fact that sometimes performance degradation is caused by time-cumulative impacts, ML-NA adopts another feature dimension, timing attributes, into its design. To be specific, the historical data is divided into groups according to the job submission time. Then, the three basic meta-features proposed above, namely the average and the standard deviation of all $\widetilde{D_{T_{ji}}}$s from tasks per node, as well as the normalized task number, are calculated within each time group to generate new feature sets.

The OpenCloud trace is used to give a concrete example of the features used in ML-NA. For the three meta-features, Figure 5.11 depicts the result of leveraging them to cluster nodes into different categories using $k$-means [94]. In the graph, machine nodes within the same clusterization group have similar execution performance. The meta-feature values in Figure 5.11 is calculated once within the whole period of time, hence the three dimension. This is for visualization purpose: high dimensional image is not straightforward. The real features are more sophisticated after bringing in the timing attributes.

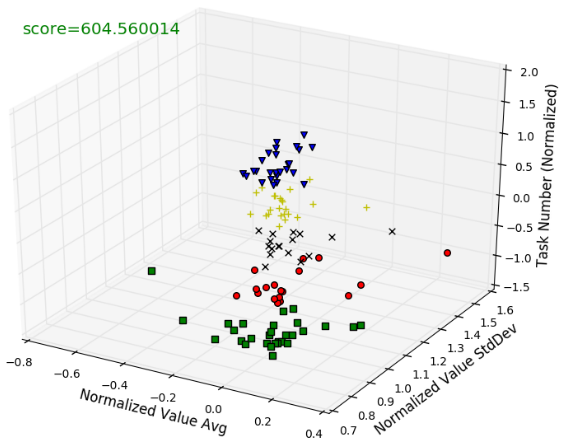


Figure 5.11: Clustering results with three features ($k = 5$)

One way to introduce the timing attribute is to split the trace according to days. For example, if all the OpenCloud traces within the first month is in hand, containing 30 days' data, the input can be constructed into a dataset consists of following 91-tuples, with each of them representing a node characterized by corresponding features.

$$\begin{aligned} <M_{id}, avg\{\widetilde{D_{T_{ji}}}_{day1}\}, \sigma\{\widetilde{D_{T_{ji}}}_{day1}\}, norm\{N\_task_{day1}\}, \\ avg\{\widetilde{D_{T_{ji}}}_{day2}\}, \sigma\{\widetilde{D_{T_{ji}}}_{day2}\}, norm\{N\_task_{day2}\}, \cdots, \\ avg\{\widetilde{D_{T_{ji}}}_{day30}\}, \sigma\{\widetilde{D_{T_{ji}}}_{day30}\}, norm\{N\_task_{day30}\} > \end{aligned}$$

Within this 91-tuple, $avg\{\widetilde{D_{T_{ji}}}_{day1}\}$ and $\sigma\{\widetilde{D_{T_{ji}}}_{day1}\}$ represent the average and the standard deviation of all tasks' normalized value assigned onto the machine $M_{id}$ in the $1^{st}$ day; $norm\{N\_task_{day1}\}$ stands for the normalized task number on machine $M_{id}$ compared with all other nodes within the cluster in the $1^{st}$ day. To note that, $avg\{\widetilde{D_{T_{ji}}}_{day2}\}$, $\sigma\{\widetilde{D_{T_{ji}}}_{day2}\}$, and $norm\{N\_task_{day2}\}$ are calculated based on all tasks submitted in both first and second day together, rather than the $2^{nd}$ day itself. In other words, this timing attribute calculation is performed in a cumulative manner. Similarly for the $30^{th}$ day, the results are derived from the whole month's data rather than a single day.

We process the whole OpenCloud trace into 9 subsets according to months to do the feature extraction, each contains 30 days data ignoring the fact that natural months are slightly different in day numbers. This makes it easier to construct training and test sets for later prediction.

### 5.2.2 The Automatic Labeling Algorithm

Labeling is required due to the lack of direct performance indicator of nodes in the tracelog data, while the labeled information is needed in order to train a classifier. Previously, to label a weak node is more of a manual process that depends on the system administrator, which is prone to errors. In ML-NA, we propose an automatic labeling algorithm that utilizes the generated features to objectively discriminate weak performance nodes from the normal ones within the cluster.

• Clustering

To label a node, we first thing we do is to put the nodes with similar performance into the same group. In this scenario, clustering is the most well-known technique that can be used, and $k$-means is one of the simplest whilst very effective algorithm [94].

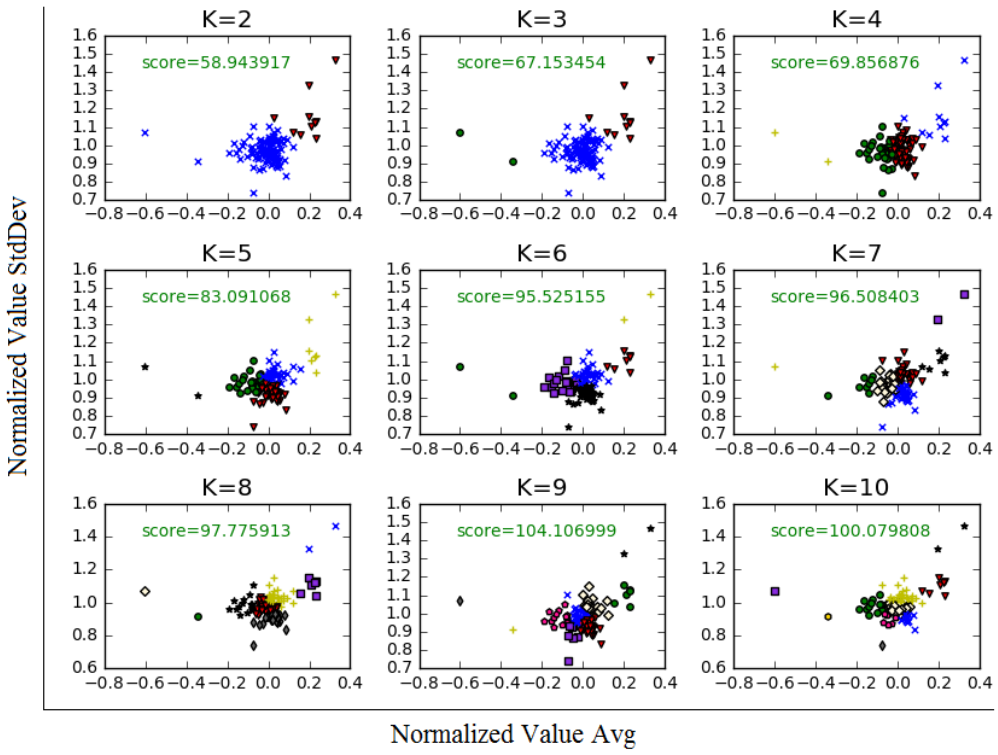


Figure 5.12: Different $k$-clustering results with two features as an example

The key parameter for launching $k$-means is to find the optimal $k$ value: it should be sufficiently large to constrain the number of nodes within each group, so that the minority (weak performance nodes) can be separated from the majority (the normal ones), yet maintaining the best clustering result characterised by a high calinski-harabaz score. The calinski-harabaz score measures the covariance within each cluster and among different clusters. Higher calinski-harabaz score signifies a superior clustering result [131]. Figure 5.12 demonstrates the score variation when $k$ is changing from 2 to 10 using only two features to represent node performance as an example. The reason why only two features are included in Figure 5.12 is simply for the clarity of figure description, while in real ML-NA, 90 features (except $M_{id}$ in the 91-tuple) are used to conduct the clustering.

Table 5.3: The *k* value choices

| | The 1st Month | The 2nd Month | The 3rd Month | The 4th Month | The 5th Month |
|---|---|---|---|---|---|
| Calinski Harabasz Score | 2:252.05; 3:219.60; 4:212.24; 5:241.34; 6:267.97; 7:357.94; 8:389.98; 9:447.72 | 2:1449.31; 3:968.79; 4:778.84; 5:767.33; 6:762.84; 7:762.31; 8:761.51; 9:741.02 | 2:128.80; 3:107.89; 4:110.40; 5:153.44; 6:200.49; 7:275.37; 8:357.01; 9:393.83 | 2:428.35; 3:452.37; 4:446.41; 5:373.91; 6:333.96; 7:302.50 8:288.48; 9:285.85 | 2:345.36; 3:257.99; 4:280.71; 5:296.65; 6:297.33; 7:333.09; 8:324.92; 9:316.74 |
| Optimal K | 9 | 2 | 9 | 3 | 7 |
| Sample(-)% | 18.46% | 13.70% | 3.23% | 8.45% | 17.53% |
| | The 6th Month | The 7th Month | The 8th Month | The 9th Month | |
| Calinski Harabasz Score | 2:124.19; 3:121.24; 4:121.23; 5:124.35; 6:133.90; 7:131.06; 8:124.73; 9:119.51 | 2:68.39; 3:154.48; 4:124.28; 5:113.03; 6:105.02; 7:98.69; 8:97.83; 9:94.52 | 2:90.36; 3:92.16; 4:84.52; 5:77.26; 6:67.87; 7:69.39; 8:64.08; 9:64.63 | 2:107.25; 3:123.62; 4:107.18; 5:97.08; 6:95.62; 7:86.36; 8:83.89; 9:81.18 | |
| Optimal *K* | 6 | *4* | *4* | *4* | |
| Sample(-)% | 21.43% | 13.21% | 12.50% | 10.34% | |

Table 5.3 details the calinski-harabaz scores when $k$ is ranging from 2 to 10 for each month. In this thesis we specify the maximum clustering number to be 10 when exploring the optimal $k$, based on the observation that desirable proportion of weak performance nodes can be sorted out. The maximum $k$ number can be easily customized according to different purposes. The *optimal k* in the table represents the final $k$ used when labeling the data within each month. Most optimal $k$ is the one with the highest score. However, for the $7^{th}$, $8^{th}$ and $9^{th}$ month, the optimal $k$ used is slightly different. This is due to the fact that, in these months, the $k$ with the best score is not big enough to differentiate a proper proportion of "weak" node set, leading to a negative sample percentage above $25\%$. Under this circumstance, the second largest score with a greater $k$ is chosen.

- Labeling

After putting the nodes with similar performance into $k$ groups, we then need to determine which cluster represents the weakest performance group. Conventional labeling is typically conducted manually by the system administrator, which suffers from operational inefficiency, and the subjective may lead to misidentification. To cope with it, an automatic labeling algorithm is proposed adopting two heuristic ranking criteria:

C1 : *The nodes with weakest performance should have the most number of* $N$ *(*$N \in [1, 30]$*) where* $avg\{\widetilde{D_{T_{ji}}}_{dayN}\}$ *is positive* .

The number of positive $avg\{\widetilde{D_{T_{ji}}}_{dayN}\}$ is the primary indicator when judging whether a specific group contains weak performance nodes. According to Equation 5.1, a positive $\widetilde{D_{T_{ji}}}$ signifies a straggler. Therefore, for node $M_{id}$, a positive $avg\{\widetilde{D_{T_{ji}}}_{dayN}\}$ indicates a high likelihood of straggler occurance on the $N^{th}$ day. As a result, the number of positive $avg\{\widetilde{D_{T_{ji}}}_{dayN}\}$ can be used to imply the frequency of such slow tendency the node exhibits in a month time.

C2 : *If multiple nodes have the same number of* $N$*, then the one with the smallest average* $\sigma\{\widetilde{D_{T_{ji}}}_{dayN}\}$ *suggests the worst performance.*

The $\sigma\{\widetilde{D_{T_{ji}}}_{dayN}\}$ value implies the confidence when predicting node performance, because it represents a stable or random status. A small $\sigma\{\widetilde{D_{T_{ji}}}_{dayN}\}$ indicates a concentrated $\widetilde{D_{T_{ji}}}$ distribution for node $M_{id}$. Therefore, for nodes that already shown a slow tendency, e.g. with maximum $N$ where $avg\{\widetilde{D_{T_{ji}}}_{dayN}\}$ is positive according to [C1], smaller average $\sigma\{\widetilde{D_{T_{ji}}}_{dayN}\}$ evidence a higher chance of weakness for this node group.

The center point of each clustering group is compared and all the nodes from the same group will be given the same label. The detailed algorithm is presented in algorithm 8 following the above two heuristics, with label “1” represents the negative sample (weak nodes), and “0” indicates the positive sample of nodes that exhibit normal performance. To note that, the primary purpose of this algorithm is to predict the weakest performed nodes to avoid the straggler behavior, therefore binary labels are adopted in align with this goal. According to different usage, a set of labels corresponds to multiple performance levels can be adopted with only a minor modification.

**ALGORITHM 8:** Labeling Algorithm

```
Inputs:
        Training sets $data$ = {⟨ M$_{id}$, Avg$_{day_1}$,..., Num$_{day_{30}}$⟩}
        Optimal $K$ from $K$-means process
Output:
        Labelled sets{⟨ Label, M$_{id}$, Avg$_{day_1}$,..., Num$_{day_{30}}$⟩}
1  Categories = kmeans(n_clusters = $K$, data = $data$);
2  for each center in Categories do
3      for Avg$_{day_j}$, StDev$_{day_j}$ in center.AttributeList do
4          pos_counts = count the number of j, Avg$_{day_j}$ $> 0$
5          stdev_avg = calculate the average of StDev$_{day_j}$
6  WeakIndexList = Categories.indexof($max$(pos_counts));
7  WeakIndex = WeakIndexList.indexof($min$(stdev_avg)) ;
8  for each node in data do
9      if Category(node) == Categories.indexof(WeakIndex) then
10         Label = 1;
11     else
12         Label = 0;
13     node = node.insert(Label);
14 return data
```

### 5.2.3 Boosting Based Classifier

ML-NA is a multi-stage learning procedure that predicts node performance based on classification while labeled data is fed into the classifier as input. There are a lot of mature classification algorithms [94] such as SVM, Boosting, Decision Tree, Random Forest, and Naive Bayes, etc. Each algorithm emphasizes specific attributes from the training data to get the optimal performance. For example, the Bayesian classifier requires all the attributes to be independent of each other: the attributes $x_k$ should fulfill Equation 5.2, where $C_i$ represents a given condition.

$$P(X|C_i) = \prod_{k=1}^{n} P(x_k|C_i) = P(x_1|C_i) \times ... \times P(x_n|C_i) \tag{5.2}$$

Table 5.4 illustrates the precision, recall and accuracy when adopting different prevailing classification algorithms to the OpenCloud datasets with automatic labeling to predict performance category. Parameters used to generate these results are the default values in the Python scikit-learn library [131], and the cross-validation portion is 1/3.

Table 5.4: Algorithm comparisons

| | Precision | Recall | Accuracy |
|---|---|---|---|
| Random Forest | 89.47% | 58.62% | 92.86% |
| SVM | 100% | 6.9% | 86.22% |
| Ada Boosting | 78.95% | 51.72% | 90.82% |
| Decision Tree | 62.96% | 58.62% | 88.78% |
| Naive Bayes | 16.67% | 27.59% | 68.88% |
| XGBoost | 82.61% | 65.52% | 92.86% |

It is observable that, the Naive Bayes classifier performs significantly worse than the others. This is because, in our training set, the features (elements in the 91-tuple except $M_{id}$) are correlated, i.e., they are generated in an incremental manner according to time. This is consistent with the aforementioned limitation of the Bayesian classifier. On the contrary, boosting based methods outperform the others from the table. In ML-NA, XGBoost [41] is adopted in the classification process before prediction.

### 5.2.4 The Node Performance Prediction

Different parameter settings are tested when training the XGBoost model, and the main parameters tuned are *learning rate* $\eta$, *evaluation metrics*, and *gbtree depth*. Table 5.5 details the optimal prediction result from all testing cases, with $\eta$ being 0.1 and the maximum depth of the gbtree booster being 12. The *logloss* value calculates the negative log-likelihood is adopted as the evaluation metrics.

Table 5.5: Prediction results with parameter sets of $\eta$ = 0.1, max_depth = 12, eval_metric = logloss

| | (1,2) | (2,3) | (3,4) | (4,5) | (5,6) | (6,7) | (7,8) | (8,9) | Average | StDev |
|---|---|---|---|---|---|---|---|---|---|---|
| Accuracy | 68.49% | 93.55% | 91.55% | 79.38% | 78.57% | 45.28% | 87.50% | 86.21% | 78.82% | 0.15 |
| | - | (1+2,3) | (2+3,4) | (3+4,5) | (4+5,6) | (5+6,7) | (6+7,8) | (7+8,9) | Average | StDev |
| Accuracy | - | 64.52% | 91.55% | 82.47% | 78.57% | 41.51% | 83.93% | 87.93% | 75.78% | 0.16 |
| | - | - | (1+2+3,4) | (2+3+4,5) | (3+4+5,6) | (4+5+6,7) | (5+6+7,8) | (6+7+8,9) | Average | StDev |
| Accuracy | - | - | 69.01% | 81.44% | 75.00% | 50.94% | 73.21% | 91.38% | 73.50% | 0.12 |
| | - | - | - | (1+...+4,5) | (2+...+5,6) | (3+...+6,7) | (4+...+7,8) | (5+...+8,9) | Average | StDev |
| Accuracy | - | - | - | 94.37% | 100% | 100% | 92.45% | 98.21% | 97.01% | 0.03 |
| | - | - | - | - | (1+...+5,6) | (2+...+6,7) | (3+...+7,8) | (4+...+8,9) | Average | StDev |
| Accuracy | - | - | - | - | 98.59% | 100% | 98.21% | 94.33% | 97.78% | 0.02 |
| | - | - | - | - | - | (1+...+6,7) | (2+...+7,8) | (3+...+8,9) | Average | StDev |
| Accuracy | - | - | - | - | - | 77.36% | 82.14% | 94.83% | 84.78% | 0.07 |
| | - | - | - | - | - | - | (1+...+7,8) | (2+...+8,9) | Average | StDev |
| Accuracy | - | - | - | - | - | - | 85.71% | 89.66% | 87.69% | 0.02 |
| | - | - | - | - | - | - | - | (1+...+8,9) | Average | StDev |
| Accuracy | - | - | - | - | - | - | - | 92.86% | 92.86% | 0 |

The numbers in Table 5.5 are prediction accuracies calculated following Equation 5.3, where *TP* stands for true positive, *TN* is short for true negative. Similarly, *FP* and *FN* are abbreviations of false positive and false negative, respectively.

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN} \tag{5.3}$$

The results of different data sizes are compared in the form of a sliding window to test the sensitiveness towards training size. In Table 5.5, (1,2) represents the prediction result of node performance category in the $2^{nd}$ month through using training data in the $1^{st}$ month, and (1+...+8,9) represents the prediction result for the $9^{th}$ month by using a combined training data from the $1^{st}$ to the $8^{th}$ month, indicating a much larger training size.

In addition, the average and the standard deviation of accuracies for each training size are recorded in the table. Besides the horizontal comparison of training size, the vertical comparison of node performance category within each month is summarised in Figure 5.13. Figure 5.13 (a) concludes the minimal, the average and the maximum accuracies with the optimal parameter setting, i.e. the parameters used in Table 5.5, while Figure 5.13(b) shows a comparison result from another parameter setting, with $\eta = 0.3$, *gbtree depth* $= 9$, and *evaluation metrics* $=$ *error*, which represent the classification error rate.

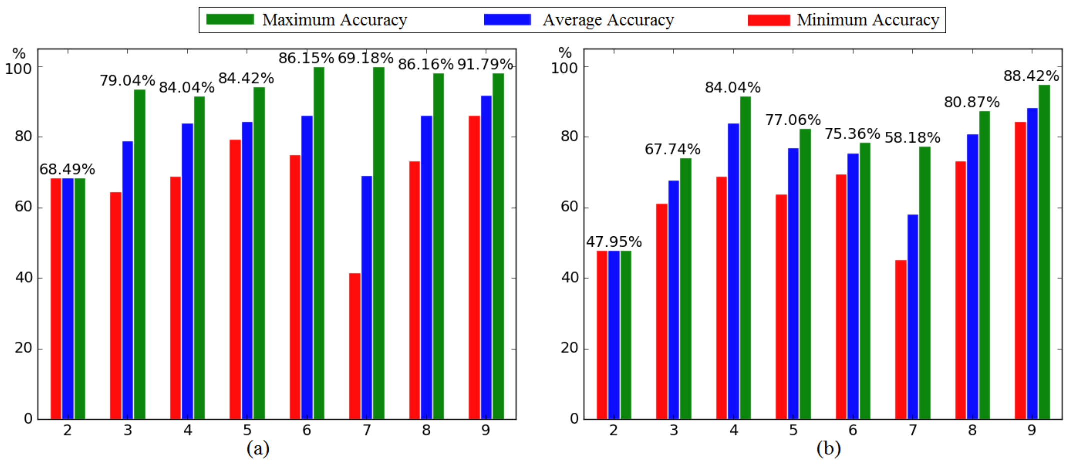


Figure 5.13: Node classification prediction accuracy for each month, with (a) parameters used in Table 5.5, and (b) a comparable parameter setting

The numbers listed in the figure are the average values across different training sizes for each month. It is observable that, the parameter settings in Table 5.5 surpasses the other testing case with much higher prediction accuracy. With proper parameter tuning, the prediction results for most months exceed 85%, and with proper training size, the highest prediction results for most months are above 90%. The highest accuracy when predicting next month's node performance category even reaches 100% in some cases.

Despite the peak accuracy ML-NA can achieve, there are still some low accuracy results under extreme cases. The worst cases occur when predicting node classification for the $2^{nd}$ and the $7^{th}$ month in the result table. The reason behind the low accuracy for the $2^{nd}$ month is relatively straightforward - the insufficient training data. When predicting the node categories for the $2^{nd}$ month, we can merely collect data from the $1^{st}$ month. In fact, the prediction accuracy based on only one month's training data tends to be limited for most months. The numbers are shown in the first row in Table 5.5. Most accuracies are below 80% such as the prediction pair of (1,2), (4,5), (5,6), and (6,7).

For the low accuracy occur in the $7^{th}$ month, it is due to the special characteristic of the data in the $6^{th}$ month. Figure 5.14 shows the first ten lines of the data collected from month six, in which, features of $avg\{\widetilde{D_{T_{ji}}}_{day1}\}$, $\sigma\{\widetilde{D_{T_{ji}}}_{day1}\}$, and $norm\{N\_task_{day1}\}$ to $avg\{\widetilde{D_{T_{ji}}}_{day3}\}$, $\sigma\{\widetilde{D_{T_{ji}}}_{day3}\}$, and $norm\{N\_task_{day3}\}$ from the 91-tuple are *NaN*. This is due to the fact that, there are no tasks been submitted to the syste on the first three days of the $6^{th}$ month, leading to a blank value for those feature columns. These unexpected *NaN*s form a noticeable different pattern, leading to a low prediction accuracy.

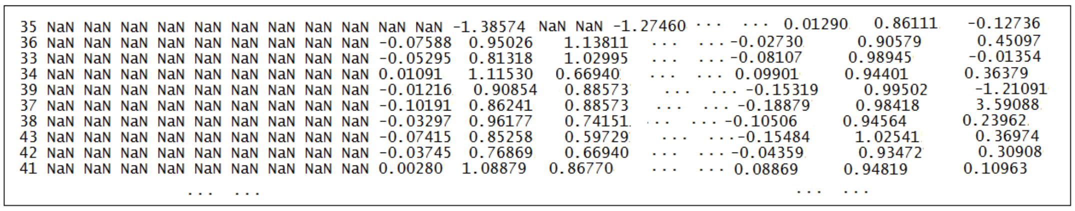

```
35 NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN -1.38574 NaN NaN -1.27460 ...   ...  0.01290   0.86111    -0.12736
36 NaN NaN NaN NaN NaN NaN NaN NaN NaN -0.07588  0.95026   1.13811   ...   ...-0.02730     0.90579     0.45097
33 NaN NaN NaN NaN NaN NaN NaN NaN NaN -0.05295  0.81318   1.02995   ...   ...-0.08107    0.98945     -0.01354
34 NaN NaN NaN NaN NaN NaN NaN NaN NaN 0.01091   1.11530  0.66940   ...   ... 0.09901    0.94401    0.36379
39 NaN NaN NaN NaN NaN NaN NaN NaN NaN -0.01216  0.90854   0.88573    ...   ... -0.15319     0.99502     -1.21091
37 NaN NaN NaN NaN NaN NaN NaN NaN NaN -0.10191  0.86241   0.88573   ...   ... -0.18879    0.98418     3.59088
38 NaN NaN NaN NaN NaN NaN NaN NaN NaN -0.03297  0.96177   0.74151  ...   ...-0.10506    0.94564    0.23962
43 NaN NaN NaN NaN NaN NaN NaN NaN NaN -0.07415  0.85258   0.59729   ...   ...-0.15484     1.02541     0.36974
42 NaN NaN NaN NaN NaN NaN NaN NaN NaN -0.03745  0.76869   0.66940   ...   ...-0.04359     0.93472     0.30908
41 NaN NaN NaN NaN NaN NaN NaN NaN NaN 0.00280  1.08879  0.86770    ...   ... 0.08869    0.94819    0.10963
                      ...   ...                                                    ...   ...
```

Figure 5.14: Training / evaluation segment for the $6^{th}$ month with *NaN* attributes

This result reveals a limitation of the proposed ML-NA algorithm: it can only predict node performance with high accuracy when there are jobs running in the system. Sudden reduce in task numbers may influence the algorithm performance. However, we believe this is a loose assumption that most production cluster can achieve.

The node performance classification based on the developed features reveals non-trivial correlations between task execution duration and node-level attributes along with time, and the prediction gives insights into the performance changing trend of a node in the near future, which can be utilized in designing and implementing efficient scheduling algorithm to mitigate stragglers. One of such attempts, the straggler-aware dynamic blacklisting method is developed.

## 5.3 Dynamic Blacklisting

Besides the naive speculative execution, another popular straggler mitigate technique is through avoidance. Blacklisting [166] is the representative method of this kind, avoid scheduling tasks onto known faulty nodes. However, blacklisting may be insufficient when stragglers are not restricted to a small set of machines [6], and when this node performance changes dynamically. In addition, current blacklisting is often through manual configuration, for example, to configure the *mapred-site.xml* file in Hadoop, which requires input from the system administrator. This practice is inflexible especially when the system scale increases. NEPAB, a Node Execution Performance Aware Blacklisting method, is proposed to cope with these challenges and to improve speculation efficiency through the straggler aware placement.

### 5.3.1 Implementation

In the NEPAB design, the nodes are ranked according to their susceptibility of straggler occurrence at each timestamp, and the set of machines that are with the weakest performance will be temporarily isolated from the available resource pool to improve parallel job execution time. The whole process is automated, administrators do not need to predefine parameters such as the blacklist size if they prefer not to, the algorithm can still prohibit all weak nodes to decrease straggler possibility.

We implement NEPAB based on the YARN platform, and to minimize overhead, existing interfaces provided are leveraged. Two key components of the NEPAB system are the *Node Performance Analyzer* and the *Node Health Checker*. For the former, the *History Server Rest API* is used to collect job execution log details in order to generate the task normalized value within a certain time frame. Attributes of interest that can be returned from the API call include *JobID*, *TaskID*, *AttemptsID* (indicating whether this attempt is a speculation or an original task), *SubmitTime*, *EndTime*, *Status* (success or been killed), *MachineID*, etc. The relevant syntax of the History Server Rest API can be found in [10]. In the current implementation, a file consisting of following 6-tuples is held by every node after the calculation:

$$< J_j, T_{ji}, M_k, (T_{ji})_s, (T_{ji})_e, \widetilde{D_{T_{ji}}} >$$

where $(T_{ji})_s$, $(T_{ji})_e$ and $\widetilde{D_{T_{ji}}}$ represent the start time, the end time, and the calculated normalized execution value of task $T_{ji}$ that runs on machine $M_k$. This file provides input that enables the node performance prediction.

For the latter component, we utilize the *Node Healthy Checker* mechanism YARN provided, which functions through specifying a user-defined script. When the given condition is fulfilled, the script generates a message with an “ERROR” heading to report the unhealthy status of the node to the Node Manager (NM). One example script is as follows.

*The weak node report script*

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

In this example, when the script detects the existence of the flag file, the reported “ERROR”-heading message can then be detected by the YARN NM and Resource Manager (RM), triggering the built-in mechanism that adds this specific node into a blacklist. In other words, no new tasks will be assigned to this node with *machineID* until the next iteration when the script is called again.

Through this implementation, we make sure that additional modifications to the default YARN system are minimized. The major overhead comes from the algorithm complexity itself rather than data processing or operational costs such as restart a node. The time window, between the start time parameter in the REST API and the current timestamp, can be adjusted to control the number of history tasks as inputs to inhibit computational overhead. Besides, the prediction component can be deployed in other nodes rather than the name node to further reduce the computing burden.

To further explore the NEPAB overhead, we run a set of Sort jobs in a Virtual Machine (VM) cluster with two settings: (1) the default YARN, and (2) the YARN with NEPAB configured as no node to be blacklisted. The other configurations such as the VM capacities, the Hadoop Distributed File System (HDFS) settings, the job inputs, etc. all remain the same across the whole experiment. The job execution time result for the default YARN is 299.67 seconds in average, with a StDev of 13.9 (301s, 316s, and 282s), while for the NEPAB system, duration results are 307s, 283s, and 294s (Avg: 294.67s; StDev: 9.8). These two sets of result have no difference from a statistical perspective, which indicates minimal overhead generated by the NEPAB modification.

## 5.3.2 Evaluation Results

The effectiveness of the NEPAB framework is evaluated through measuring two key performance indicators: the execution time results to demonstrate the performance in improving job response time, and the successful speculation rate results to illustrate how it benefits straggler mitigation.

• Experiment Setup

Two testbeds are set up in order to evaluate the NEPAB performance: the first one is a 30-VM cluster built on top of the OpenNebula platform [104], with typical VM configurations to be 1GB of memory, 1 virtual core with 2.34 GHz capacity, and 10GB disk space on potentially shared hard drive. The VMs use KVM virtualization software and run the Ubuntu 12.04 x86_64 operating system. Another environment is a 20-VM cluster build on top of the ExoGENI infrastructure [52] with 2 XOLarge VM and 18 XOMedium VM, both running the CentOS 6.7 operating system. Detailed configurations of each VM type can be found in [53].

In all experiments, the HDFS is configured to maintain two replicas for each data chunk. The job types mainly include WordCount and Sort, with the same reasons as given in Section 4.4. In addition, we configured the container size for both Map and Reduce tasks to be 1GB of memory, and the node capacity to be 2GB. In the ExoGENI cluster, we configured an Ambari [9] system to monitor and to manage the cluster utilization.

Table 5.6: Cluster VM configurations

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

Contention "faults" are injected into the system to simulate a complex environment with node performance heterogeneity. We test three cases with different numbers of weak nodes through creating extreme resource contention situations on certain VMs. The CPU/memory intensive tools are the same as the ones used in Section 4.4. The detailed deployment configuration for these three environments is shown in Table 5.6. In the OpenNebula cluster, VMs are referred with their private IP address while in the Exo-

Table 5.7: Job execution time results with NEPAB and YARN speculator

| | Without Injected Faults | | | | | | |
|---|---|---|---|---|---|---|---|
| | NEPAB | | | YARN$_{\text{speculator}}$ | | | Improvement |
| Sort | 183 s | Avg | StDev | 151s | Avg | StDev | -7.09% |
| | 160s | 166s | 12 | 154s | 155s | 4 | |
| | 155s | | | 160s | | | |
| WordCount | 74s | Avg | StDev | 75s | Avg | StDev | 4% |
| | 72s | 72s | 1 | 73s | 75s | 1 | |
| | 71s | | | 76s | | | |
| | **With Faults Injected** | | | | | | |
| | NEPAB | | | YARN$_{\text{speculator}}$ | | | Improvement |
| Sort | 468s | Avg | StDev | 529s | Avg | StDev | 15.46% |
| | 451s | 443s | 25 | 531s | 524s | 9 | |
| | 409s | | | 511s | | | |
| WordCount | 89s | Avg | StDev | 141s | Avg | StDev | 22.66% |
| | 100s | 99s | 8 | 137s | 128s | 16 | |
| | 109s | | | 106s | | | |

GENI cluster, VMs are referred with their hostname.

- Execution Time Performance

The final job execution time is dependent on the duration of its last parallelized task. When a subset of parallelized tasks is assigned to nodes with poor execution performance, they have a larger chance to become stragglers, which lead to an extended job response. For applications that emphasize timing constraints, this response extension may result in a Quality of Service (QoS) breakdown and cause late timing failures. After applying the NEPAB technique in the OpenNebula cluster, we get an improved job response time result as shown in Table 5.7.

The improvement value is calculated following the same general principle introduced in Equation 4.6. In this case, it is represented as $(D_{Yarn\_speculator} - D_{NEPAB})/D_{Yarn\_speculator}$, with $D_{NEPAB}$ and $D_{Yarn\_speculator}$ stand for the average job duration under the NEPAB framework and in the original YARN system with the default speculator, respectively. The execution averages and standard deviations are calculated based on three experiment runs for each test case listed in Table 5.7.

From the results it is observable that, for system configuration consists of only homogeneous default VMs without any injected faults, the response time improvement is limited: only 4% on average for the WordCount job. And for the Sort job, the NEPAB system sometimes even results in deteriorated execution: -7.07% on average. This is because when nodes are exhibiting similar execution performance, such as the example of the $3^{rd}$ to the $5^{th}$ day as shown in Figure 5.9, and when cluster size is limited, 30 VM in this case, to blacklist node may hinder system capacity. Under these circumstances, capacity is more important for job execution performance compared with the negative impact of stragglers caused by weakly performed nodes.

On the contrary, NEPAB functions well for the cluster with heterogeneous node performance: 15.46% and 22.66% improvement for Sort and WordCount, respectively. These results in Table 5.7 are generated under the blacklisting policy that all weak nodes identified by the automatic ranking are blacklisted. During the experiments, this number is ranging from 2 to 5 in the 30 node OpenNebula cluster, indicating a 7% to 20% weak node percentage.

We have tested the additional *top-k* policy through controlling the parameter of the prohibited node number as well. The job response time in the ExoGENI cluster is evaluated, with the number of blacklisted nodes ranging from 0 to 5. Zero blacklisted node represents the comparison standard performance of the original YARN. The results are detailed in Figure 5.15, from which it is observable that, 3 is the optimal number of $k$ (15% in proportion) when determining the number of weak nodes to be blacklisted in this system configuration, with an average improvement for job response time being 55.43%.

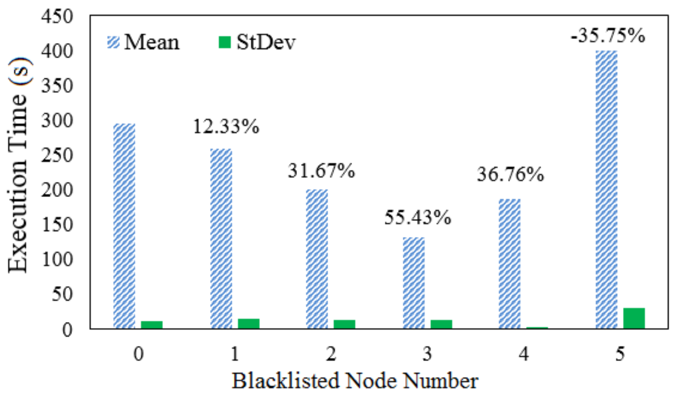


Figure 5.15: The average job execution time (with standard deviation) with different numbers of blacklisted nodes in the ExoGeni cluster

However, if 5 nodes (20% in proportion) are blacklisted from this 20-node cluster, the execution time will be increased by 35.75%. This is observed due to the fact that, if the number of the blacklisted nodes is too small, such as 1 (5% in proportion) in Figure 5.15, there is still a possibility that some of the machines with high straggler occurrence continue to hinder the job execution, while on the other hand, if this number is too large, it will make the system suffer from the capacity loss. And in this experiment case, the number of three covers all VMs with injected memory fault (weak nodes), which is consistent with the node performance configuration.

- Speculation Efficiency Performance

Another important improvement of the NEPAB framework is its effectiveness in reducing the speculation failure rate. Bearing the principle of assigning speculative replicas to fast nodes to enlarge their chance of surpassing the stragglers, NEPAB is effective in improving the performance of successful speculation. Detailed results are listed in Figure 5.16: the average successful speculation rate is 63.8%, 65.33%, and 62.83% for 1 (3%), 2 (7%), and 3 (10%) blacklisted nodes, respectively. The efficiency performance is doubled compared to the current successful speculation rate in real-world systems, which is less than 30% as analyzed in Section 3.4.1.

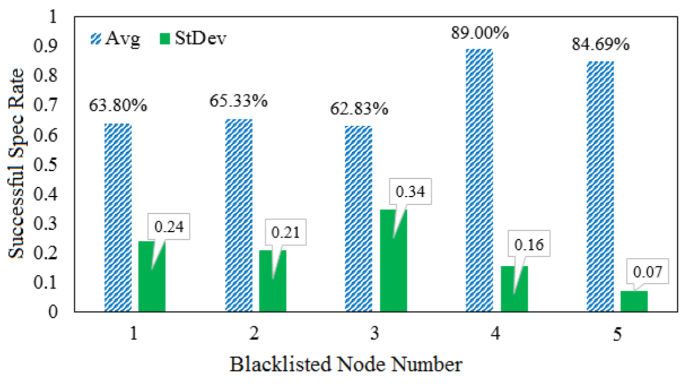


Figure 5.16: The successful speculation rate with different numbers of blacklisted nodes in the the OpenNebula cluster

We also notice that, the successful speculation rate in the NEPAB system increases to almost 90% after we further blacklist 4 to 5 nodes. This is because that, for situations when only a part of weak nodes are prohibited, there is still a chance for the speculations to be assigned to servers with poor execution performance. And this problem is eliminated after the number of blacklisted nodes covers the majority of the weak ones.

All the average and the standard deviation values listed in Figure 5.16 are calculated based on three execution runs for each case to eliminate randomness. In addition, the evaluation of situation with more than 6 nodes to be blacklisted is not included in the figure, due to the fact that the maximum number of weak nodes identified by the NEPAB method is 5, and there is no need to blacklist nodes that behave normally.

## 5.4 Summary

Parallelized tasks with similar designed duration may end up with varied execution time after assigning on different machine nodes. Weakly performing nodes influence parallel job execution by enlarging the possibility of the straggler behavior, and can limit speculation efficiency by hindering successful replications. This chapter targets at the influence of node performance when conducting straggler mitigation, and the main contributions are summarised as below:

- Proposed a node execution performance modeling and ranking algorithm, which analyze node ability in terms of job execution through the statistical pattern draw from historical task behaviors on the node. The Goodness of Fit (GoF) result demonstrates that, most machine nodes follow the same distribution when measuring the normalized task durations, and the normalization process made this approach can be applied to the general workload types. Leveraging the Google trace as a case study, this algorithm manages to identify 0.83% weakest nodes out of the whole cluster, and reveals a fact that node execution performance is a dynamic attribute of node changes over time, and it does not depend solely on physical capacity.

- Developed a Machine Learning (ML) based prediction scheme that accurately captures the performance changing trend of a node. A series of features are explored to describe node performance, including normalized task execution times and task number per node values, statistical characteristics and timing attributes. An automatic labeling algorithm to generate objective labels for machines according to their performance category is developed, and the nodes are classified and predicted. Results show that, the ML based method is capable of predicting node performance categories with an average accuracy up to 92.86%.

- Implemented a node execution performance aware blacklisting (NEPAB) framework on top of YARN that improves both job execution performance and speculation efficiency. The periodically updated straggler possibility per node analysis accurately reflect the newest system state, and the speculation can be enhanced via blacklisting, in which nodes with the weakest performance in the next scheduling window will be temporarily prohibited. The overhead is minimized by leveraging information that already collected by the default YARN system and the built-in node healthy checker mechanism. Results show that, NEPAB can improve job completion time up to 55.43% compared to the default YARN speculator, and is capable of increasing successful speculation rate up to 89%.

Refer back to the system model outlined in Section 3.5, this chapter provides the server-level prediction for the Node Performance Analyzer component.

# Chapter 6

# Coping with Skew-caused Stragglers

Current speculative execution scheme has an unavoidable limitation when dealing with data skew caused stragglers. One biggest hypothesis assumed by the speculative execution is that, the redundant copy will behave as a quick task like other normal ones that do not fall behind. Therefore, even it is launched after the straggler, it still has a chance to catch up. However, when mitigating skew caused stragglers, because the speculative copy needs to process identical input with the original task, itself will again become a straggler caused by the uneven input distribution. Figure 6.1 illustrates this scenario with a WordCount example. A job $J_j$ processes 50 documents with 50 Map tasks $T_{jM_1}$ to $T_{jM_{50}}$, the default documents are 10MB in size, with 0, 1, 3, 5, and 7 expensive files that are 50MB in size to mimic the uneven input distribution. Number 0 on the x-axis indicates no skew inputs, while number 1 to 7 represent lightly skewed data towards more severe skewed inputs. It is observable from the example that, the speculation failure rate increases with the number of skewed inputs.

In order to prevent such speculation breakdown, it is necessary to develop a skew mitigation method as a complementary component for the designed straggler mitigation system.

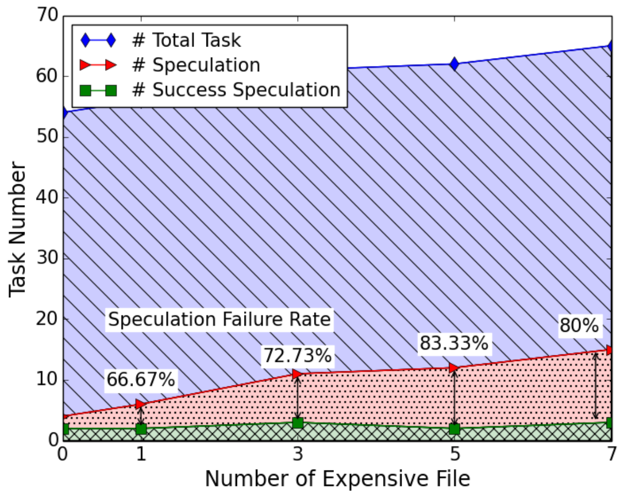


Figure 6.1: Speculation failure rate with different input skews

## 6.1 Skews in MapReduce Framework

Skews are often witnessed under the MapReduce framework. Before further analysis, we first extend our notion expressions to further differentiate Map tasks and Reduce tasks with the general parallel tasks we formulated in the previous discussions.

### 6.1.1 Refined Notions

For a MapReduce job $J_{MR_j}$, $T_{jM_i}$ and $T_{jR_i}$ represent the $i_{th}$ Map and Reduce task, respectively. $< K_i, V_i >$ stands for the key/value pair processed and generated, where $K_{Im_i}$ is the intermediate key. The MapReduce workload can then be represented as: (input) $< K_i, V_i > \rightarrow$ Map $\rightarrow [< K_{Im_i}, V_{Im_i} >] \rightarrow$ combine $\rightarrow < K_{Im_i}, [V_{Im_i}] > \rightarrow$ partition $\rightarrow < K_{Im_i}, [V_{Im_i}], T_{jR_i} > \rightarrow$ shuffle $\rightarrow < K_{Im_i}, [V_{Im_i}] > \rightarrow$ Reduce $\rightarrow [< K_{Im_i}, V_{Im_k} >]$ (output). The Map tasks transfer the input file into key value pairs with the customized keys defined by the application developer.

For example, in the WordCount job that counts the frequency of each word in a document, the keys are defined as the independent words, with the value of “1” indicating one appearance of the key. Refresh our memory from Section 2.2.2, the *combine* phase is an optional optimization that combines the value of the same key within each $T_{jM_i}$ to reduce the network traffic for later shuffle phase. Still use WordCount as an example, the *combine* process will generate one record of $< word, 5 >$ out of 5 $< word, 1 >$ pairs. The *partition* phase is responsible for marking the keys with Reducers, which determines the Reduce input distribution.

Once the partition is done, $T_{jR_i}$ will copy the output marked with its own ID from each $T_{jM_i}$ using HTTP across the network (the *copy/shuffle* operation), and then the *sort* operation merges and sorts the intermediate keys for $T_{jR_i}$. This is necessary because different $T_{jM_i}$s may output the same intermediate key. The *shuffle* and *sort* operations often occur simultaneously, i.e. while the Map outputs are being fetched, they are merged and sort.

### 6.1.2 Different Skew Types

The skews in MapReduce stem from various reasons. For Map, the most common skew is caused by uneven input file size [79]. For example, if a 150MB size input is processed by the application running on a Hadoop cluster with 128MB HDFS block size configuration, the input will be divided into two files with 128MB and 22MB in size. The Map skews can be addressed by splitting the expensive file or adjusting the HDFS parameters, which is relatively straightforward. In contrast, skews in the Reduce phase are more complicated.

There are mainly two types of skew Reduce tasks can encounter: the *expensive key group* skews and the *partition* skews. For the former, the MapReduce framework requires that all tuples sharing the same key should be dispatched to the same Reducer. Key groups refer to the sequence of $< K_{Im_i}, [V_{Im_i}] >$ pairs. Many real-world datasets exhibit skews in nature. Figure 6.2 shows some examples, among which, (a) is the word frequency from the Shakespeare collection [140] and (b) is from the wiki English dataset [116].

Reduce task can easily encounter the expensive key group skew if WordCount is run on such data: for Figure 6.2 (a), there are altogether 67,056 words with the most frequently used one appears 23,197 times, while the average word count is 13; for Figure 6.2 (b), there are 21,433,355 words with the most frequently used one appears 46,134,908 times, while the average word count is 43. Another example would be the PageRank application, a link analysis algorithm that assigns a weight to each node in a graph by iteratively aggregating the weight of its inbound neighbors. If a graph contains nodes with a large degree of incoming edges such as Figure 6.2 (c) and Figure 6.2 (d), PageRank will suffer from Reduce skew. Figure 6.2 (c) is the Google web dataset and Figure 6.2 (d) is the Facebook social circles dataset [135]. The x-axis refers to the web pages (represented as nodes in the PageRank graph) and the y-axis is the number of hyperlinks in each page

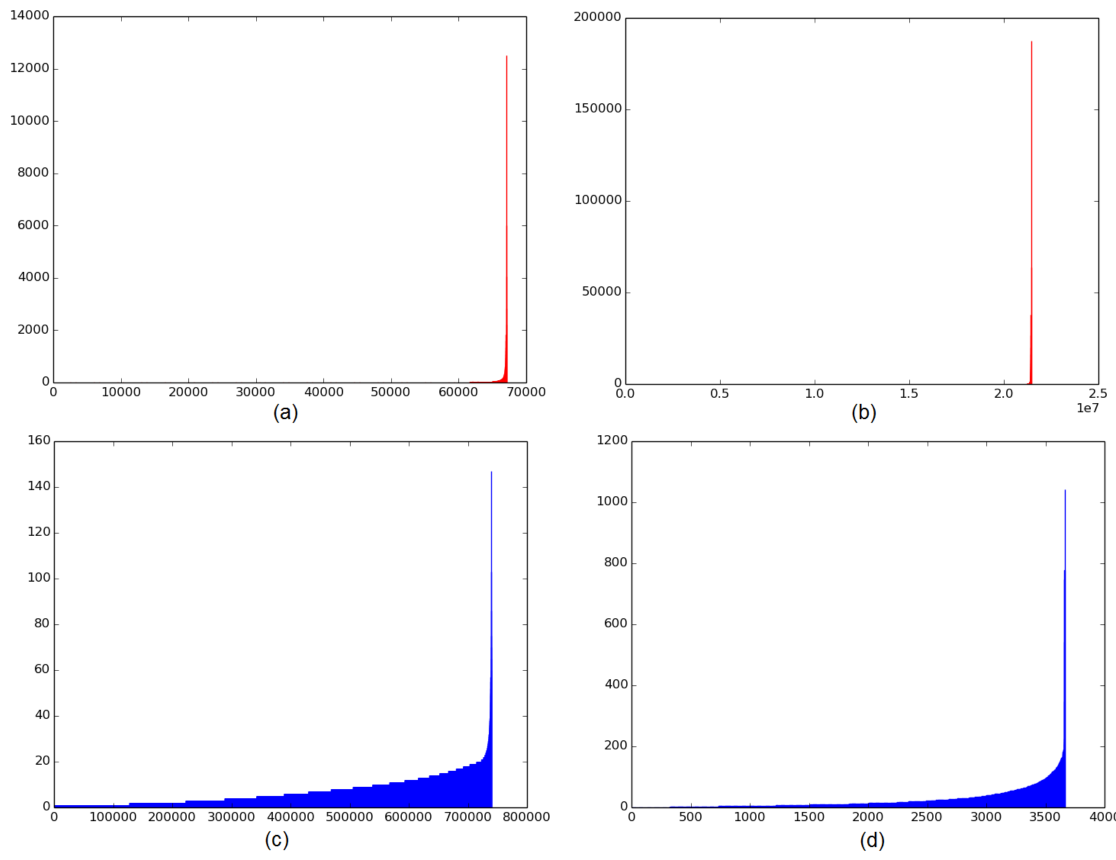


Figure 6.2: The word distribution of (a) the Shakespeare collection, (b) the English wiki dataset, and the edge number distribution of (c) the Google web dataset, (d) the Facebook social circles dataset

(represented as edges in the PageRank algorithm). In Figure 6.2 (c), there are 739,454 pages, and the biggest graph node contains 456 linked edges, however the average number of edges per node is only 7; in Figure 6.2 (d), there are 3,363 pages included in the figure, the largest page contains 1,043 edges while the average edge number per node is 24.

The other type of skew is exclusive to Reduce tasks. It is called the partition skew because it is caused by unreasonable partition decisions, such as the example job given in Figure 6.3 with two Reducers. If the hash function categorizes the intermediate key of "A","B", and "F" into a group and "C", "D", and "E" to another, the *Reducer1* will have to process 1.9 times of key-value pairs compared to *Reducer2*, where a partition skew occurs. On the contrary, a smart partition algorithm in this example can actually achieve possible "no-skew" solution also shown in Figure 6.3.

The severity of the intermediate data skew varies with different Reducer numbers. When processing the same data in the above example but with three Reducers instead of only

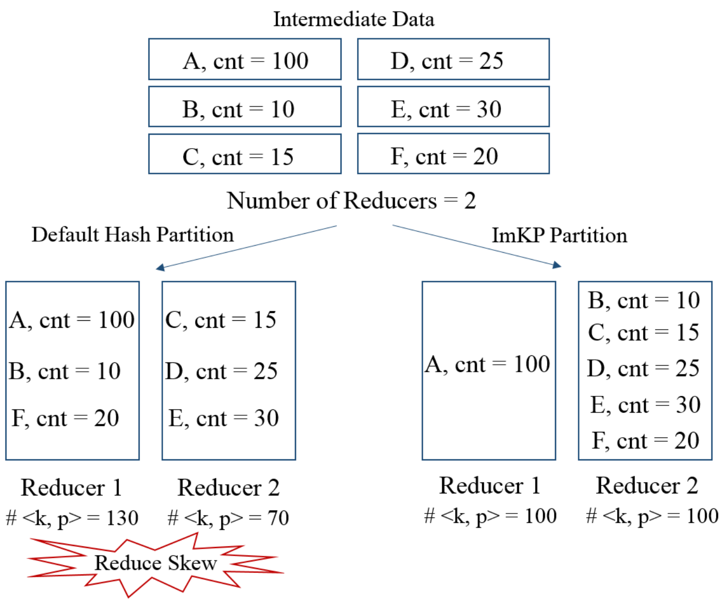


Figure 6.3: Reduce skew and possible improvement the ImKP method can achieve

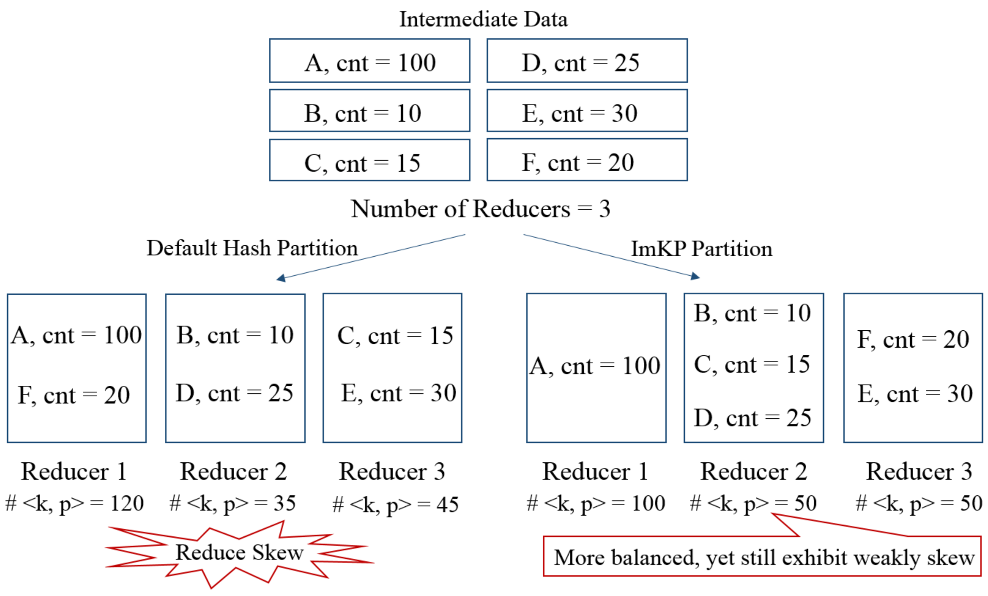


Figure 6.4: ImKP limitation illustration

two, the hash partition can result in a different skew situation, which is slightly better as illustrated in Figure 6.4. The degree of skew can sometimes be alleviated by enlarging the Reducer number, however, such practice introduces new challenges include overloaded network traffic due to the increased communication, etc. There are some general principles of choosing the reasonable Reducer number, which is not the focus of this research. We care about the partition policy, and any potential achievement accordingly. In the following discussions, improvements are discussed under the same Reducer configuration.

For an arbitrary application, the distribution of the intermediate data cannot be determined ahead of time. This is the main reason why partition skew happens, and to estimate this distribution is the key thing for most partition-skew mitigation schemes.

## 6.2 Mitigate Reduce Skews with ImKP

In order to minimize the input skew in MapReduce framework, especially the Reduce skews caused by the bad partitioner, an Intermediate Key Pre-processing framework (ImKP) is proposed to improve the overall straggler mitigation efficiency.

### 6.2.1 The ImKP Framework

There are some general design requirements / goals that a good skew mitigation system should accomplish. Examples among them include minimal developer burden, mitigation transparency, flexibility, and minimal overhead.

For minimal developer burden, the MapReduce application developers should be able to migrate their code into the proposed skew mitigation platform with no requirement of learning new techniques. That is to say, the new system should adopt uniform APIs with existing MapReduce platforms such as Hadoop to minimize development complexity.

For skew mitigation transparency and flexibility, the former requires the proposed technique to be transparent to the end users. For normal users, when they launch MapReduce applications on the new platform, there should be no need for them to manually conduct additional configurations regarding skew mitigation if they prefer not to, and they do not need to get into the algorithm details such as the parameter tuning for the partition policy, etc. The latter principle, on the other hand, is for expert users who emphasize certain performance or some level of control to the system. The new framework should provide the possibility for them to insert alternative information to generate flexible partition results.

The requirement of minimal overhead asks for the additional overhead spent on mitigating the skew phenomenon, including extra computation and resources, to be trivial enough that generates no negative impact toward final application level performance indicators such as job execution time.

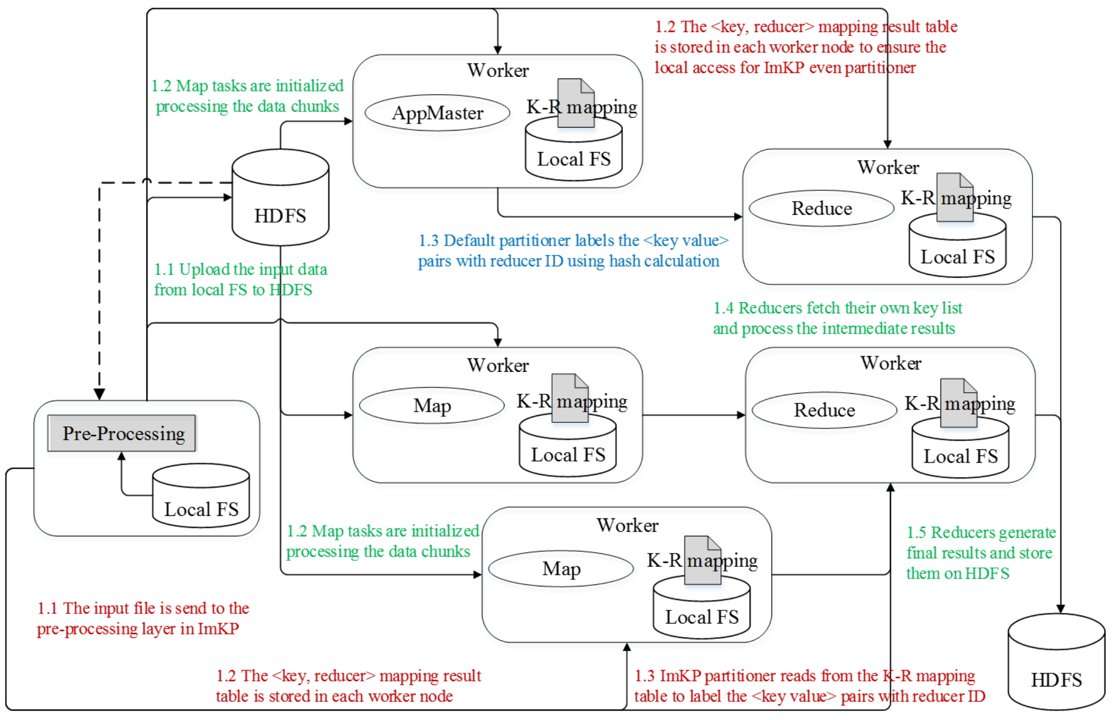

1.2 The <key, reducer> mapping result table is stored in each worker node to ensure the local access for ImKP even partitioner
1.2 Map tasks are initialized processing the data chunks
Worker
AppMaster
K-R mapping
Local FS
HDFS
Worker
Reduce
K-R mapping
Local FS
1.3 Default partitioner labels the <key value> pairs with reducer ID using hash calculation
1.1 Upload the input data from local FS to HDFS
1.4 Reducers fetch their own key list and process the intermediate results
Worker
Map
K-R mapping
Local FS
Worker
Reduce
K-R mapping
Local FS
Pre-Processing
Local FS
1.2 Map tasks are initialized processing the data chunks
Worker
Map
K-R mapping
Local FS
1.5 Reducers generate final results and store them on HDFS
1.1 The input file is send to the pre-processing layer in ImKP
1.2 The <key, reducer> mapping result table is stored in each worker node
1.3 ImKP partitioner reads from the K-R mapping table to label the <key value> pairs with reducer ID
HDFS

Figure 6.5: The system model for the ImKP framework

The proposed ImKP framework fulfills above requirements while enables the even distribution for Reduce inputs. The overall architecture of the ImKP system is presented in Figure 6.5, with the shaded parts to be the components added that exclusively belong to ImKP and the rest are compatible with current Hadoop YARN implementation. Texts in green represent the workflow that both original YARN application and the ImKP job need to go through. The blue texts are exclusive to original YARN, while the red texts solely belong to the ImKP logic. The procedures with the same sequential number indicate the fact that they are executing in parallel. For example, ImKP utilizes the multithreading implementation to parallelize the pre-processing with file uploading to mitigate timing overhead. This is reflected with two 1.1 steps in red and green texts simultaneously.

### 6.2.2 Detailed Workflow and the Pre-processor

Figure 6.6 outlines the workflow comparison for the original MapReduce job and the ImKP framework. The procedures represented in rectangles are the same with current YARN implementation, while the rounded rectangle indicating the changed parts. The major difference between the default hash partitioner and the proposed ImKP even partitioner is that, the later inserts a pre-processing layer before the Map phase to get the

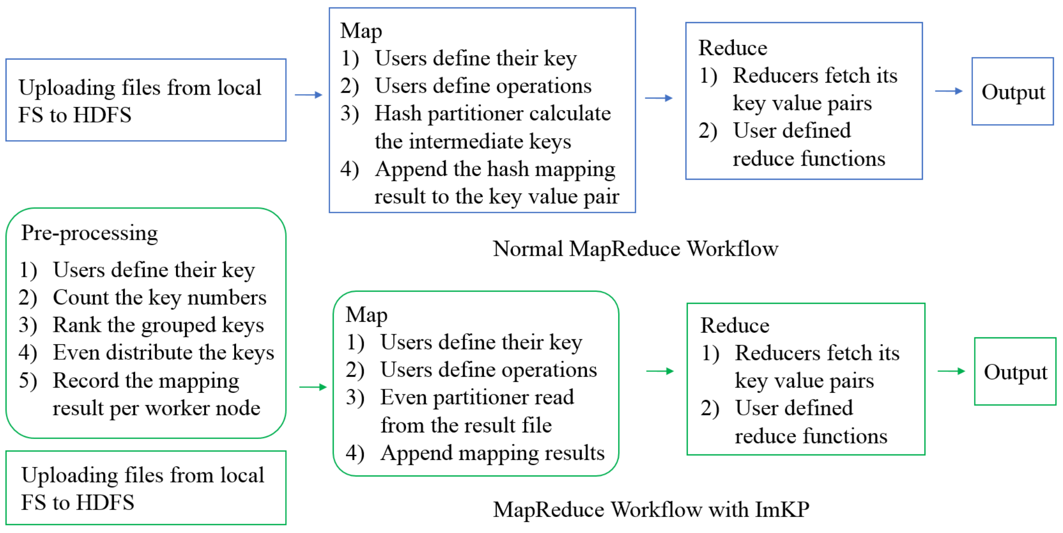

Uploading files from local FS to HDFS
Map
1) Users define their key
2) Users define operations
3) Hash partitioner calculate the intermediate keys
4) Append the hash mapping result to the key value pair
Reduce
1) Reducers fetch its key value pairs
2) User defined reduce functions
Output
Normal MapReduce Workflow
Pre-processing
1) Users define their key
2) Count the key numbers
3) Rank the grouped keys
4) Even distribute the keys
5) Record the mapping result per worker node
Uploading files from local FS to HDFS
Map
1) Users define their key
2) Users define operations
3) Even partitioner read from the result file
4) Append mapping results
Reduce
1) Reducers fetch its key value pairs
2) User defined reduce functions
Output
MapReduce Workflow with ImKP

Figure 6.6: ImKP workflow VS normal MapReduce workflow

accurate intermediate key distribution and to generate a balanced dispatch solution depending on the number of Reducers. In other words, under the ImKP framework, the input data is first sent to the pre-processing component before Map. This additional layer is responsible for generating the *K-R mapping* file that enables the later even partitioner.

We managed to control the pre-processing overhead to a limited level through a group based ranking technique, so that the time spend on pre-processing is constrained to be less than file uploading time. In this way, ImKP guarantees trivial timing overhead toward job execution for applications whose input is stored on local file system. For applications whose input is already stored on HDFS, since the *K-R mapping* file is stored in memory ready for reuse once the pre-processing is done, the overhead is not an expensive price to pay, especially for those jobs that have to go through multiple iterations such as PageRank.

Consistent with the original MapReduce framework, Map tasks are generated to handle the input data chunks after the pre-processing and file uploading process. However, once the Map function finishes, unlike the default partitioner which does the hash calculation in order to label the intermediate key generated by Map with Reducer ID, the even partitioner in ImKP directly look up to the *K-R mapping* table. The table is stored on every machine node within the YARN cluster to ensure the local access for the even partitioner, regardless of the Mapper position.

In addition, the *K-R mapping* mapping table is extremely small in size because of the group based ranking optimization, containing only $\#Reducer \times scale$ rows. The notion of

$scale$ is a user-defined parameter implying the degree of evenness in ImKP pre-processor, with default value to be 50. For example, for applications with 10 Reducers, there will only be 500 bi-tuples in the mapping table. This small size guarantees the promptness of the local read operation. The timing overhead of reading the mapping file from memory and of doing the hash calculation are tested, the average time for the former operation is 10,000ns while the latter is 9,000ns, which is only 1,000ns in difference. In other words, the default hash partitioner and the ImKP even partitioner take approximately same time when conducting the partition operation.

The pre-processor mainly consists of following steps:

- *Define customized keys:* In order to calculate the intermediate key statistics from the input, the definition of the keys must be given. For example, keys are defined as separate terms in a document in WordCount, or as each page node in the PageRank graph. Because the keys required by the ImKP initializer is identical to the keys defined in the Map phase, this step does not require additional developer intervene, the ImKP system can automatically copy the key-define function from the user program given in the original MapReduce framework.

- *Rank the intermediate keys:* The frequencies of the intermediate key occurrence are counted and ranked in this step. The biggest challenge encountered here comes from the fact that the MapReduce framework is designed for big data applications that process a large number of intermediate keys. If the frequency for each key is recorded separately for ranking, it will come at huge computational ($\mathcal{O}(n \log n)$ complexity) and storage costs. A group based ranking scheme is proposed in order to solve this problem. The assumption supports this optimization is that, we believe the number of keys is way beyond the number of Reducers, therefore, one Reducer would have to process multiple keys. Instead of directly rank all the intermediate data, we first map the keys into groups using a hash to decrease the number of items that need to be ranked. A parameter of $scale$ is adopted in this procedure to imply the total number of grouped keys one Reducer will later receive. Altogether $\#Reducer \times scale$ number of key groups are created, and the occurrence frequency of keys will be counted per group for ranking.

- *Even distribute the key groups based on frequency ranking:* This step generates the $< K_{Im_i}, Reduer >$ mapping result for the ImKP even partitioner to assign intermediate keys to Reducers. The best fit policy is adopted in our implementation for this bin-packing problem: the key group with the maximum occurrence frequency in the remaining queue will be mapped to the Reducer with the minimum sum of frequencies. The intermediate keys in the same group will be mapped to the same Reducer. For advanced users, an API is provided so that this default best-fit method can be replaced with more dedicated algorithms. For example, if additional information on the performance diversity among machine nodes is known, this even distribute scheme can be adjusted accordingly. The result mapping file is stored on every worker node within the cluster so that the local access for the ImKP even partitioner can be guaranteed.

## 6.3 Performance Evaluation

The ImKP evaluation focuses on answering following three questions: (1) can it mitigate the Reduce skews by generating a more balanced input size distribution for Reducers; (2) whether the skew mitigation overhead is small enough to be ignored; and (3) whether the overall job response time can be improved.

For each question, either multiple workload types or multiple MapReduce configurations are tested to verify whether the performance improvement remains consistent through different operational situations.

### 6.3.1 Experiment Setup

To evaluate the effectiveness of the ImKP framework, various experiments are run in a 15 Virtual Machine (VM) cluster build on top of the ExoGENI infrastructure [52]. Each VM contains 1 CPU core, 3GB RAM, and 25GB disk, running CentOS 6.7. In all experiments, the container for both Map and Reduce tasks are 1GB in size.

Popular applications including WordCount, PageRank, and Inverted Index provided in the Bespin toolkit [83] are tested, on both synthetic and real-world datasets. We generate 1.6GB synthetic data files following the Zipf distribution with varying $\sigma$ parameters from 0.4 to 1.4 to control the degree of the skew. The larger $\sigma$ value represents a heavier skew. Zipf distribution is commonly observed in real-world datasets, e.g., the word occurrences in natural language, features of the Internet, etc [84]. For real-world data, the Shakespeare collection [140], the English Wiki dataset [116], and the Freebase dataset [58] are used to run the experiments.

### 6.3.2 Skew Mitigation Effectiveness

The number of $< K_{Im_i}, V_{Im_i} >$ pairs processed by each Reducer using the original hash partition and the ImKP even partition algorithm is shown in Figure 6.7. Results are generated after running (a) Inverted Index on the Shakespeare dataset; (b) PageRank on the Freebase dataset; and (c) WordCount on Zipf data set.

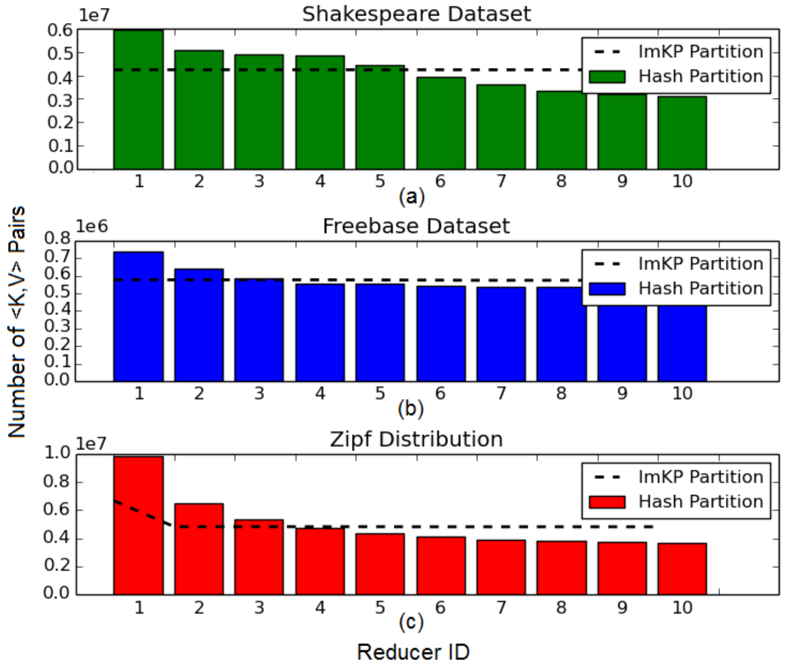


Figure 6.7: Number of inputs per Reducer for (a) Inverted Index on Shakespeare data; for (b) PageRank on Freebase data; and for (c) WordCount on Zipf data

From Figure 6.7 (a) and (b) it is observable that, ImKP achieves an extremely good skew mitigation result: the number of inputs for ImKP Reducers is close to the ideal even distribution, refer to the horizontal line. The *coefficient of variation* defined in Equation 6.1 is used to measure the skewness of the Reduce inputs, where $\# < K_{Im}, V_{Im} >$ represents the number of intermediate key-value pairs processed by each Reducer.

In addition, Table 6.1 details the $C_v$ improvement $((C_v(Hash) - C_v(ImKP))/C_v(ImKP))$ and the $C_v$ times $(C_v(Hash)/C_v(ImKP))$ results with varies degree of skews to show the effectiveness of the ImKP in mitigating Reduce skews.

$$C_v = \frac{\sigma}{\mu} = \frac{std(\# < K_{Im}, V_{Im} >)}{avg(\# < K_{Im}, V_{Im} >)} \tag{6.1}$$

Table 6.1: Reduce input skew mitigation results for different skew degrees

| Reduce Input Size | | $\sigma$=0.4 | $\sigma$=0.5 | $\sigma$=0.6 | $\sigma$=0.7 | $\sigma$=0.8 | $\sigma$=0.9 | $\sigma$=1.0 | $\sigma$=1.1 | $\sigma$=1.2 | $\sigma$=1.3 | $\sigma$=1.4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 reducer | $C_v$ Improvement | 99.76% | 94.94% | 97.52% | 98.92% | 99.98% | 99.75% | 99.87% | 99.94% | 80.09% | 56.28% | 40.68% |
| | $C_v$ Times | 416.27 | 19.75 | 40.33 | 92.25 | 4187.47 | 406.16 | 775.18 | 1665.17 | 5.02 | 2.29 | 1.69 |
| 10 reducer | $C_v$ Improvement | 90.40% | 93.68% | 97.35% | 98.68% | 99.25% | 99.72% | 69.28% | 46.62% | 32.06% | 21.94% | 15.14% |
| | $C_v$ Times | 10.42 | 15.83 | 37.71 | 75.54 | 133.67 | 354.84 | 3.25 | 1.87 | 1.47 | 1.28 | 1.18 |

Table 6.2: Response time improvement for WordCount application on the Zipf data when $\sigma$ changes from 0.4 to 1.4.

| | | $\sigma$ = 0.4 | $\sigma$ = 0.5 | $\sigma$ = 0.6 | $\sigma$ = 0.7 | $\sigma$ = 0.8 | $\sigma$ = 0.9 | $\sigma$ = 1.0 | $\sigma$ = 1.1 | $\sigma$ = 1.2 | $\sigma$ = 1.3 | $\sigma$ = 1.4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 reducer | Improvement | 19.01% | 8.18% | 6.63% | 6.87% | 23.71% | 19.10% | 23.86% | 23.51% | 8.08% | 6.40% | 15.15% |
| | $C_v$(Hash) | 0.04 | 0.01 | 0.01 | 0.12 | 0.01 | 0.03 | 0.01 | 0.05 | 0.01 | 0.17 | 0.20 |
| | $C_v$(ImKP) | 0.21 | 0.02 | 0.06 | 0.13 | 0.04 | 0.07 | 0.17 | 0.13 | 0.04 | 0.15 | 0.05 |
| 10 reducer | Improvement | 12.19% | 0.96% | 15.47% | 26.34% | 11.71% | 14.80% | 12.75% | 4.96% | 13.80% | 29.37% | 15.67% |
| | $C_v$(Hash) | 0.03 | 0.04 | 0.01 | 0.17 | 0.04 | 0.02 | 0.14 | 0.10 | 0.05 | 0.12 | 0.07 |
| | $C_v$(ImKP) | 0.07 | 0.05 | 0.01 | 0.12 | 0.02 | 0.12 | 0.24 | 0.07 | 0.13 | 0.06 | 0.02 |

Meanwhile, it is observed from Figure 6.7 (c) and from the $\sigma = 1.3$, $\sigma = 1.3$ columns in Table 6.1 that, the ImKP algorithm has a limitation dealing with extreme skews. The $\sigma$ value of the Zipf distribution is 1.4 in the Figure 6.7 (c) input, which indicates a severe skew and the existence of extreme expensive keys. ImKP is mainly designed for solving the partition skews, for situations of expensive key skews where the number of intermediate data belongs to a certain key surpasses the sum of the others, ImKP can only achieve a more balanced result, yet still exhibit slightly skew. This is also shown in Figure 6.4 with a straightforward example.

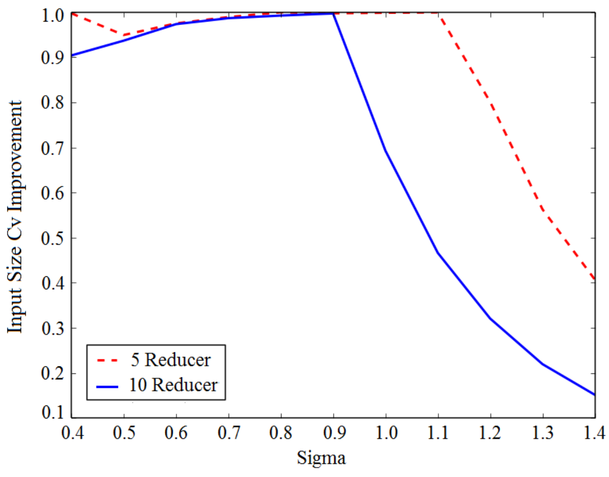


Figure 6.8: The input size improvement for ImKP and hash partition on Zipf data

This limitation explains the $C_v$ improvement changing tendency of the Reducer inputs, which is summarised in Figure 6.8. When there is only a slight skew in the intermediated data, reflected by a small $\sigma$, the original hash partition can generate a fairly reasonable mapping decision that leaves less room for improvement. This improvement gradually increases along with the increase in the skew severity. However, when passing a certain point after the skew becomes extremely heavy, the partition based method cannot solve but only relieve the problem.

### 6.3.3 Skew Mitigation Overhead

The mitigation overhead is one of the biggest concerns for the ImKP system design because it inserts an extra pre-processing layer before the normal Map phase. Different with the literature that estimates the complete intermediate key distribution, ImKP preprocessor adopts a group based ranking scheme that dramatically decreases the number

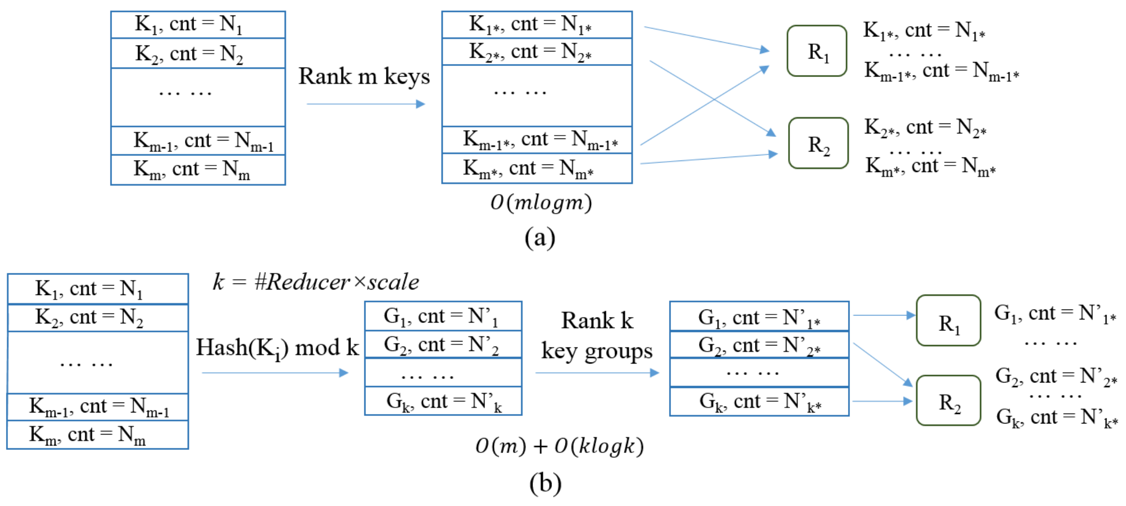


Figure 6.9: Complexity (a) before and (b) after the group based ranking optimization

of elements in the ranking process, based on the fact that one Reducer has to process multiple intermediate keys anyway. The comparison of the algorithm complexity is shown in Figure 6.9, where $m \gg k$.

Besides the complexity analysis, experiments are conducted to test the exact timing overhead of the pre-processing operation. Figure 6.10 illustrates the comparison result of the pre-processing time and the file uploading time for the WordCount application run on various input sizes. From the figure it is observable that, the pre-processing overhead is stable at a low level that remains much smaller than the file uploading time. This overhead is

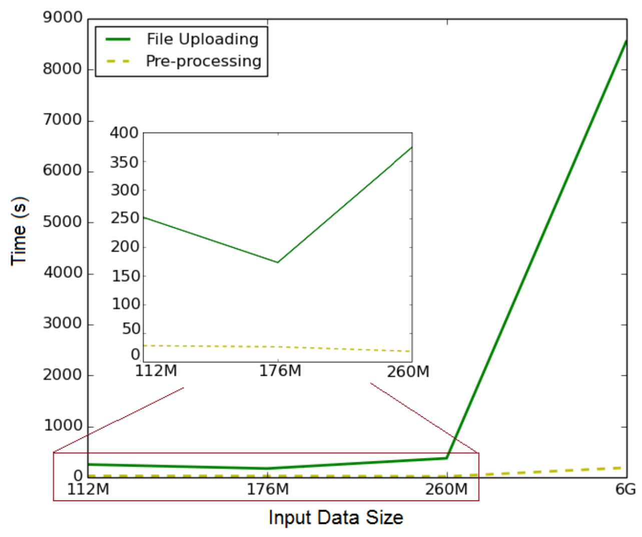


Figure 6.10: The pre-processing overhead

trivial enough even for large inputs such as the 6GB input from the English Wiki dataset. And because of the multithreading parallelization, for applications that store their inputs on local file systems, there will be no extra timing overhead on pre-processing at all. In addition, the pre-processing is only required once for every application, and the mapping file result is stored in memory on every datanode ready for possible reuse. This benefits applications that have to go through multiple MapReduce iterations such as PageRank (the PageRank score updates at each iteration before it convergences). Through this implementation, the overhead of the initial timing overhead is further reduced in ImKP.

### 6.3.4 Job Execution Improvement

According to above analysis, the two modifications made by the ImKP framework compared to the original Hadoop YARN, namely the pre-processor and the different partitioner, both generate no obvious timing difference for overall job completion. Therefore, the execution time is expected to be mainly influenced by the different number of key-value pairs processed by each Reducer. Table 6.2 lists the detailed job execution times under different skew conditions. The *improvement* is again calculated following the same principle in previous chapters (refer to Equation 4.6).

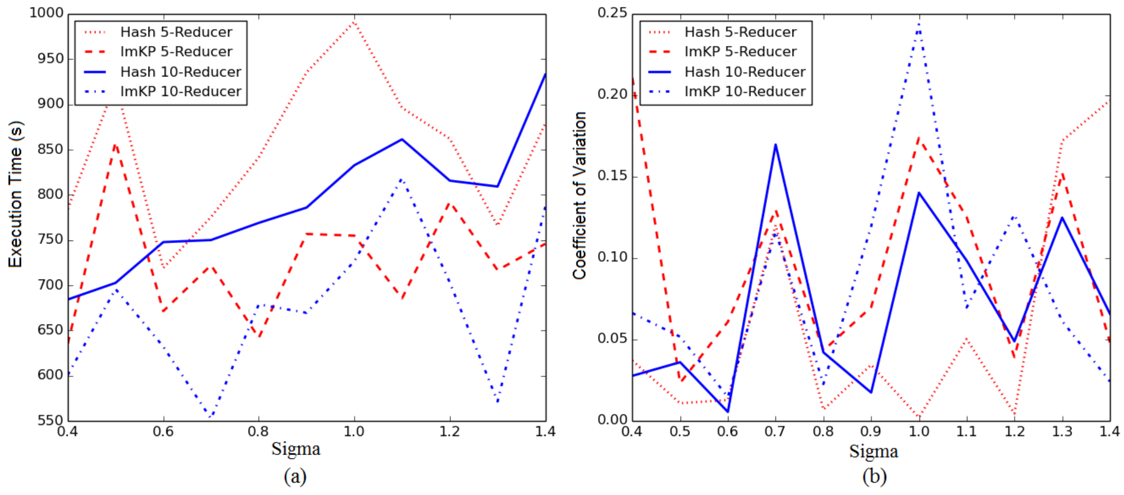


Figure 6.11: The (a) execution time; the (b) execution coefficient of variation for ImKP and hash partition on Zipf data

Figure 6.11 (a) summaries the job execution time improvement for inputs with different level of skews. The results are average values out of three running tests, and the coefficient of variation is used to represent the response time variance for each test case, as shown in Figure 6.11 (b). From the result it is observable that, the ImKP framework is capable of improving average job response time by a factor up to 29.37%. For the number of Reducers, as previously discussed, different configurations result in different levels of skew severance, therefore the improvements discussed in this evaluation are conducted under the same Reduce number configurations.

## 6.4 Summary

An intermediated key pre-processing (ImKP) framework that enables the even partition for Reduce inputs is proposed and discussed in this chapter. ImKP is used to avoid data-skew caused stragglers for Reduce tasks, a special straggler case which cannot be handled by normal speculation-based methods. Main contributions of this chapter are summarised as follows:

- Analyzed the skew behavior with various datasets and illustrated the type of skews within MapReduce framework. The influence of data-skew caused stragglers, especially the Reduce skews, toward efficient speculative execution is discussed, which highlights the necessity of developing a dedicated algorithm targets at skew handling in the straggler mitigation system.

- Proposed ImKP, the Intermediate Key Pre-processing framework that plugged an intermediate key ranking layer before the original Map phase to enable the even partition for Reduce inputs. Results show that, the skewness of input sizes for Reducers can be decreased by 99.8% on average. And overall job response time can be improved by up to 29.37%.

- Developed a group based ranking technique that dramatically reduces pre-processing overhead for the ImKP system. In addition, through parallelizing the pre-processing with the file uploading process, we even managed to eliminate the overhead for workloads that take inputs from local file systems.

Refer back to the system model outlined in Section 3.5, this chapter deals with a special case of stragglers that caused by skewed input data, provides support for the Skew Pre-processor component and functions in a complementary manner.

# Chapter 7

# Conclusion and Future Work

This chapter summarises the work presented in this thesis. The major contributions of the research are outlined and an evaluation of the research is presented. Future directions that this work can be taken are then discussed.

## 7.1 Summary

The work in this thesis proposes an intelligent straggler mitigation framework that can fit into current distributed computing systems and Cloud environments. Specifically, an in-depth data analysis is conducted using multiple real-world system tracelogs to analyze how the straggler problem affects large-scale parallel computing performance. The results of this analysis are leveraged for practical usage in straggler mitigation, and every algorithm proposed is implemented so that experimental comparison can be made against the existing state of the art approaches. The tradeoffs between speculation overhead and execution time improvement are also explored.

Chapter 2 describes the basic background that underpins this research - i.e. the evolution of computing systems from tiered standalone systems to Cloud computing, and onwards to future Edge and Internet of Things (IoT); the concept of parallel computing frameworks and job execution performance; the performance challenge of the straggler problem; and an overview of current straggler mitigation techniques. For the first part, the taxonomy of Cloud computing is illustrated in detail, including identified characteristics, deployment models, and service models. For the second part, the MapReduce architecture and its popular open-source implementations, such as Hadoop and YARN, are presented in detail, including an introduction to its workflow and refined subphases. It is shown that while parallel computing is an active research area, there are challenges in guaranteeing prompt and predictable performance. As a representative example of such challenges, straggler definition and corresponding formulation are presented as the main content of the third part. Finally, state-of-the-art straggler mitigation methods, straggler cause analytics methods, and skew mitigation methods, are introduced in the fourth part, followed by a discussion of the gaps, highlighting the importance of this work.

Given the background concepts of parallel job performance in large-scale computing systems, Chapter 3 presents the quantitative results of how such performance can be influenced by the straggler problem. The three production datasets used for analysis - the Google trace, the AliCloud trace, and the OpenCloud trace - are introduced, with basic scales (machine node, job number, task number, etc) and statistical attributes given in detail. This is then followed by straggler related analysis, which is quantified in terms of straggler pattern statistics, straggler reason analysis, and speculation limitation analysis. Based on the observations made, it is found that current straggler mitigation is far from solving the straggler problem, and requires further in-depth study. This chapter is concluded by presenting and discussing the overall system model of the proposed straggler mitigation system, with responsibilities of each respective components.

Chapter 4 focuses on evaluating the most suitable task stragglers for mitigation through an adaptive threshold. This corresponds to the responsibility of the *Adaptive Speculator* component. Most current speculative-based methods detect stragglers by specifying a predefined threshold to calculate the difference between individual tasks and average task progression within a job. However, such a static threshold limits speculation effectiveness as it fails to capture the intrinsic diversity of timing constraints in Cloud 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 Quality of Service (QoS) for different applications. The presented algorithm improves the execution efficiency of parallel applications by dynamically calculating the straggler threshold, considering key parameters including job QoS timing constraints, task execution progress, and optimal system resource utilization. This dynamic straggler threshold is implemented into the YARN architecture, enables experimental evaluation of its effectiveness against existing state-of-the-art solutions. A simulation is conducted using SEED [60] in order to evaluate the advantages of the proposed algorithm in a larger-scale system.

Chapter 5 focuses on evaluating the most suitable machine nodes for launching tasks to avoid stragglers. This corresponds to the responsibility of the *Node Performance Analyzer* component. The key innovation here is that, instead of simply using physical capacity or contention level to represent node performance, it is believed to be reasonable to use the execution history of a node to represent its ability in fulfilling tasks. The normalized execution time is defined in the chapter, and a detailed probability distribution analysis is presented using the Google trace as a case study. A set of statistical attributes extracted from the distribution is listed to represent the node performance. Besides this statistical-based analysis, this chapter also proposes a Machine Learning (ML) based straggler analyzer which classifies and predicts the performance changing tendency of nodes to avoid straggler occurrence. The feature extraction process and subsequent clustering are introduced in detail, followed by a proposed labeling mechanism that adopts two heuristic rules. In addition, a straggler aware scheduling framework is proposed, combining the node performance analysis result with a blacklisting technique and a node *healthy checker* mechanism. This dynamic node blacklisting scheme is implemented in YARN to experimentally validate its performance.

Chapter 6 focuses on dealing with a special straggler type: stragglers caused by data skew. This is dealt by the *Skew Pre-processor* component. The reason why this complementary method is needed is given in the chapter: although shown to be effective for contention caused stragglers, speculative execution can easily meet a bottleneck when mitigating data skew caused stragglers due to its replicative nature. Identical unbalanced inputs will lead to a slow speculative task, irrespective of what node it is on. The skew types in the MapReduce framework are analyzed, as well as the popular methods used for mitigating each type. In this chapter, we focus on *partition skews* and propose an Intermediate Key Pre-processing framework that enables an even partitioner for Reduce inputs. A group based ranking technique is introduced in detail, which can dramatically decrease pre-processing time. In addition, the proposed algorithm manages to eliminate the timing overhead through parallelizing the pre-processing with the file uploading procedure (from the local file system to Hadoop Distributed File System (HDFS)). The timing overhead for jobs that take input directly from HDFS is minimized through storing the $< GroupedKey, Reducer >$ mapping file on every cluster node for reuse. This algorithm is also implemented into the YARN system, and experiments are conducted on different datasets with various workloads to evaluate its performance.

The proposed intelligent straggler mitigation system can be easily merged into current parallel computing platforms such as Hadoop YARN or Spark, where speculation scheme is already adopted. In other words, in systems where task progress is already monitored and recorded. The framework can be applied to more general production systems as long as corresponding task monitoring component is added. And for Cloud environments where the cluster scheduler focuses on virtualized resources, the speculator only focuses on Virtual Machine (VM) in the current design, which is consistent with the cluster scheduler. Further exploration about VM placement and how that placement would influence stragglers would be further work.

## 7.2 Research Contributions

This research is centered on providing a framework to capture straggler patterns in large-scale systems, to mitigate the negative impact brought by the stragglers and to improve parallel job execution performance. The main contributions are summarised as follows:

i. *The study and quantification of performance inefficiencies within large-scale computing environments caused by the straggler problem.* An in-depth analysis across three different Cloud environments reveals that less than 5% of task stragglers can influence the completion time of more than half of total parallel jobs. To make things worse, current speculation schemes normally come with an extremely high failure rate, typically exceeding 70%. There are many reasons that lead to straggler occurrence, among which resource contention is the dominant one, accounts for frequencies around 80%. This is the first time that there has been literature focusing on quantitative analysis towards straggler related research.

ii. *An adaptive threshold calculation algorithm for enhanced straggler identification and efficient speculation in large-scale computing systems.* This algorithm dynamically adapts to different job types and system conditions. Therefore, it is effective in improving job completion time and reducing late timing failures. The replica number trade-offs for different levels of resource utilization is capable of reducing the speculation failure rate and is beneficial toward effective quality assurance. Results demonstrate that the adaptive approach is capable of reducing job response time by up to 20% compared to a static threshold, as well as a higher speculation success rate, achieving up to 66.67% against 16.67% in comparison to the static method.

iii. *A machine node execution performance modeling and prediction scheme that enables dynamic node blacklisting to avoid straggler occurrence.* The ability for servers to effectively execute tasks varies due to heterogeneous CPU and memory capacities, resource contention situations, network configurations and operational age. Unexpectedly slow server nodes result in assigned tasks becoming stragglers. However, it is currently unknown how slow nodes directly correlate to straggler manifestation. To solve this problem, this research first proposes a method for node performance modeling and ranking in Cloud systems based on analyzing parallel job execution tracelog data, then designs a machine learning based prediction algorithm that classifies cluster nodes into different categories and predicts their performance category in the near future with a high accuracy of up to 92.86%. This information is used as a scheduling guide to support dynamic node blacklisting, which improves speculation effectiveness and minimizes task straggler generation.

iv. *An intermediate key pre-processing partition algorithm dealing with stragglers caused by data skew in the MapReduce framework to act as a complementary part for the straggler mitigation system.* This research analyzes the data skew behavior with various datasets and illustrates the type of skews within the MapReduce framework. The proposed algorithm inserts an intermediate key ranking layer before the original Map phase to enable an even partitioner for Reduce inputs. The group based ranking optimization and the parallelization technique used effectively controls the overhead brought by the pre-processing operation. Results show that compared to the popular hash partition, the proposed algorithm can dramatically decrease Reduce skew, achieving a 99.8% reduction in the coefficient of variation of input sizes in average, and improve performance up to 29.37% in job response time.

## 7.3 Overall Research Evaluation

The success criteria of this research is to see whether the objectives discussed in chapter 1 have been achieved. The overall evaluation is listed as follows:

*Analyzing straggler related statistics within Cloud computing systems.* This thesis has explored three real-world Cloud system tracelogs, with some common principles observed related to the straggler behavior detailed in chapter 3. Besides the general straggler influence analysis, chapter 3 also conducted a straggler root-cause analysis and a speculation limitation analysis, leveraging the AliCloud dataset and the OpenCloud dataset respectively. In addition, how straggler occurrence rate can be influenced by machine nodes is explored in chapter 5, with some new conclusions made - for example, nodes with larger capacity can behave slower than the ones with smaller physical capacity, etc. These observations serve as the solid foundation that motivates the rest of the research.

*Identifying the most appropriate stragglers for mitigation.* Through the usage of the dynamic straggler threshold, chapter 4 details the design of adaptive speculation, which is capable of picking out the target stragglers according to different system conditions and application types. This algorithm is explained theoretically with two numeric examples, and is evaluated both experimentally and through simulation. The simulation set up in respect of straggler rates and straggler length are regular patterns found in chapter 3, which reveals real behaviors from production systems.

*Avoiding straggler occurrence through modeling and predicting machine node performance.* In chapter 5, a comprehensive analysis of how node execution performance can influence straggler behavior on the machine is presented, as well as two node performance analyzer utilizing historical information of stragglers. The first node analyzer is statistical based while the second one is a machine learning based method. These prediction results of the node performance changing tendency have been applied to the straggler aware dynamic node blacklisting approach, which is also proposed in chapter 5. The precision of weak node identification is vital in blacklisting approaches: false positives result in unnecessary capacity loss while false negatives reduce straggler avoidance effectiveness. To further enhance the algorithm, besides the graph-based ranking method which automatically generates the weakest node set, a customized API is also provided that can enables a certain top $k$ worst nodes to be isolated.

*Developing a dedicated algorithm to deal with situations when speculative execution is not appropriate.* This is handled by the research presented in chapter 6. Data skew caused stragglers are the most representative type of straggler that speculation cannot deal with. The pre-processing algorithm proposed learns the distribution of intermediate data before the Reduce phase, and hence makes it possible to build an even partitioner that eliminates Reduce skews. This is very important for mitigating stragglers under the MapReduce framework.

The evaluation of this research is summarised through the comparison against other straggler mitigation approaches listed in Table 2.3. Detailed results are demonstrated in Table 7.1. In the table, *Intelligent* represents the intelligent straggler mitigation scheme proposed in this thesis, compromising the adaptive straggler threshold algorithm, the dynamic server blacklisting algorithm, and the Reduce skew mitigation algorithm. It is observable that, the gaps listed in chapter 2 are improved. In summary, all the four major research objectives have been successfully completed.

Table 7.1: Comparison of my research against other representitive approaches

| Methods | Naive Spec [155] | LATE [162] | Mantri [5] | Dolly [6] | SkewTune [80] | CREST [81] | Grass [7] | Intelligent |
|---|---|---|---|---|---|---|---|---|
| Metrics | PS | ECT | $t_{rem}$, $t_{new}$ | Cloning | $t_{rem}$, $t_{par}$ | PR | ECT | ECT, $t_{new}$ |
| Target Type | General | General | General | Small Jobs | General | General | Approx Jobs | General |
| Node Hetero | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ |
| Dy- NP | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ |
| Dy- Thresh | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ |
| Extra Reso- | ✗ | ✗ | ✓ | 5% | ✓ | ✗ | ✓ | ✓ |
| Spec Cap | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | – |
| Data Skew | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ |
| Bench mark | WC, Sort | WC, Sort | WC, Hive | – | II, PageR | GSA | Hive, Scope | WC, Sort, Hive, II |

## 7.4 Future Work

There are a number of future directions through which this work can be enhanced. Some of these opportunities are highlighted below:

- **Online Root Cause Analytics and Enhanced Straggler Filter**

Root cause analysis of stragglers is challenging in real-world systems because of the stochastic nature of task execution. Current "cause-effect" analysis is often a "backward" one: possible system behaviors that can lead to stragglers are listed and analyzed based on probability. However, the "forward" ones: upon straggler detection, how to determine the major reason that has lead to this specific straggler, are merely seen. If we map stragglers to a special system failure, such as late-timing failure, straggler root cause analysis can refer to the research of fault tolerance as shown in Figure 7.1. The part in the solid square is the root cause analysis of faults [127], while the dotted square is the logical extension of straggler research.

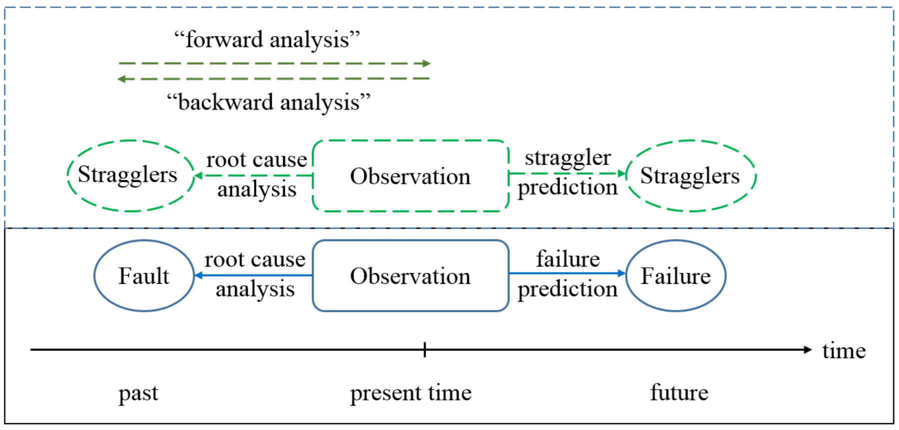


Figure 7.1: Future work on root cause analysis

In addition, read from the timeline within the graph, online failure prediction incorporates measurements of actual system observations during runtime in order to assess the probability of failure occurrence in the near future in terms of seconds or minutes [127]. Mapped to the straggler research, most current literature focuses on analyzing historical data when doing the root cause identification, while practical speculator has to make online decisions. to pick up the suitable method corresponds to the straggler cause and system characteristics during runtime, and to predict whether a straggler will occur in the near future based on an assessment of the monitored system state, require future work on online root cause analytics.

Another aspect when developing enhanced speculation in the future is to further optimize the straggler filters used when detecting slow tasks. Combining different straggler thresholds used in current literature and locating the overlapped common straggler set may help identify the most urgent stragglers, which is worth exploring as the next step of this research.

- **Straggler Mitigation in Approximate Jobs**

Current research is carried under the assumption that the parallel job needs to wait for all its sub-tasks to produce results before generating the final outcome. However, it is possible that some type of job only requires partical results from its sub-tasks. This approximation processing becomes more and more important in the big data era. The rapid growth of data volumes, along with the limitation of cluster capacities and the concurrency of multiple running jobs, has made it inevitable that deadline-bound jobs tend to operate on only a subset of their data in order to meet strict deadline constraints [1].

Due to the fact that the accuracy of these jobs is somehow proportional to the fraction of data processed, the goal of scheduling such jobs and the speculations is to satisfy deadline requirements while trying to process as much data as possible to improve accuracy. Doing some exploration to achieve this goal forms another interesting orientation of this research.

- **Intelligent Scheduling with Machine Learning**

Resource management problems are a natural expansion of straggler research due to the fact that task execution is highly dependent on workload and environments [47]. For instance, the running time of a task varies with data locality, server characteristics, interactions with other tasks, and interference on shared resources such as CPU caches, network bandwidth, etc [61]. While these two factors, workload and environments, are the intuitive focus of resource management systems as well.

Resource management in the real world is challenging: underlying systems nowadays are often complex and hard to model accurately, and practical schedulers have to make online decisions with a huge amount of input data, which is sometimes noisy. In addition, some performance metrics such as the tailing performance we focused in this thesis, are hard to optimize. Most related works target resource management challenges with meticulously designed heuristics using a simplified model of the problem. However, inspired by recent advances in Machine Learning (ML) techniques especially in powerful models such as deep learning, it can be conjectured that, systems which automatically learn to manage resource from experience using Artificial Intelligence (AI) will be the future trend.

There are already quite a few explorations toward utilizing AI-based methods in the resource management domain, translating the problem of packing tasks with multiple resource demands into a learning problem. For example, [75] employs ideas from transfer learning [109] to cope with the problem of generalizing the profiling results from one specific machine to a mathematical model that other machines can use to characterize their sensitivity curve when modeling the task co-locating interference.

Among the many explorations, Reinforcement Learning (RL) is the most popular. RL learns to make decisions directly from the rewards generated through interaction with the environment, and has a long history [138]. However, it becomes extremely popular only until recently when combined with *Deep Learning* techniques such as Deep Neural Networks (DNN), in applications including playing Go games [134], cooling datacenters [51], and etc. The system model that combines RL with DNN is shown in Figure 7.2 [88].

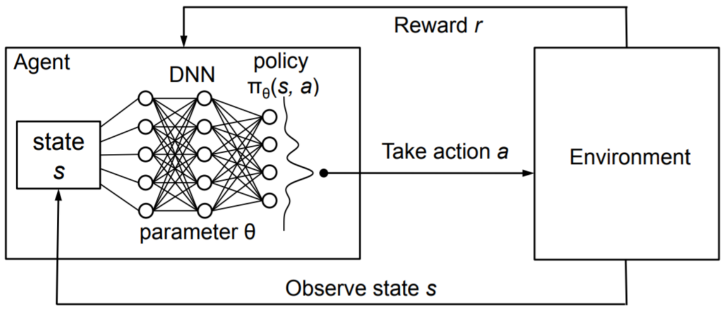


Figure 7.2: The RL system model with policy represented via DNN [88]

From the system model it is observable that RL approaches are especially well-suited to resource management systems. The *agent* that observes *state* $s$ and performs *action* $a$ can be naturally mapped to the cluster scheduler. The state transitions and rewards are stochastic, and the goal of the learning is to maximize the expected cumulative discounted reward $\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t]$, where $\gamma \in (0, 1]$ is a factor discounting future rewards.

In [88], average job slowdown is the training objective, with rewards at each timestep set to be $\sum_{j \in \jmath} \frac{-1}{T_j}$, where $\jmath$ is the set of jobs currently in the system and $T_j$ represents the duration of the job. In this design, the cumulative reward over time coincides with the sum (negative) of job slowdowns, the completion duration divided by job duration. Therefore, maximize this reward mimics minimizing the average slowdown. Other objectives such as to minimize average completion time can be achieved through changing the reward function into $-|\jmath|$. With a properly designed reward function, the RL methods can be adopted to build an intelligent system that mitigates stragglers. This ML-based optimization forms another important future direction of this research.

# Bibliography

[1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013. **Cited on page 157.**

[2] Hussain Al-Aqrabi, Lu Liu, Jie Xu, Richard Hill, Nick Antonopoulos, and Yongzhao Zhan. Investigation of it security and compliance challenges in security-as-a-service for cloud computing. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2012 15th IEEE International Symposium on*, pages 124–129. IEEE, 2012. **Cited on page 14.**

[3] Ahmed Ali-Eldin, Johan Tordsson, Erik Elmroth, and Maria Kihl. Workload classification for efficient auto-scaling of cloud resources. *Tech. Rep.*, 2013. **Cited on page 26.**

[4] AliCloud, 2017. URL `https://www.alibabacloud.com/`. **Cited on page 47.**

[5] 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*, volume 10, page 24, 2010. **Cited on pages 28, 29, 33, 34, 39, 41, 50, 75, 83, 92, 93, and 155.**

[6] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, volume 13, pages 185–198, 2013. **Cited on pages 33, 34, 41, 49, 75, 124, and 155.**

[7] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. Grass: Trimming stragglers in approximation analytics. 2014. **Cited on pages 34, 37, 41, and 155.**

[8] Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. Web services agreement specification (ws-agreement). In *Open grid forum*, volume 128, page 216, 2007. **Cited on page 25.**

[9] Apache-Ambari, 2016. URL `https://ambari.apache.org/`. **Cited on pages 5 and 126.**

[10] Apache-Hadoop-YARN-History-Server-Rest-APIs, 2016. URL `http://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/HistoryServerRest.html`. **Cited on page 124.**

[11] Apache-Hive, 2016. URL `https://hive.apache.org/`. **Cited on page 83.**

[12] Wiki Applications powered by Hadoop, 2017. URL `https://wiki.apache.org/hadoop/PoweredBy`. **Cited on page 21.**

[13] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. Technical report, Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009. **Cited on page 1.**

[14] Kevin Ashton. That internet of things thing. *RFiD Journal*, 22(7), 2011. **Cited on page 16.**

[15] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010. **Cited on page 16.**

[16] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science, 2001. **Cited on page 25.**

[17] 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):11–33, 2004. **Cited on page 24.**

[18] Moussa Ayyash, Hany Elgala, Abdallah Khreishah, Volker Jungnickel, Thomas Little, Sihua Shao, Michael Rahaim, Dominic Schulz, Jonas Hilt, and Ronald Freund. Coexistence of wifi and lifi toward 5g: Concepts, opportunities, and challenges. *IEEE Communications Magazine*, 54(2):64–71, 2016. **Cited on page 16.**

[19] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003. **Cited on page 14.**

[20] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013. **Cited on page 15.**

[21] Vladimir Batagelj and Matjaž Zaveršnik. Generalized cores. *arXiv preprint cs/0202039*, 2002. **Cited on page 108.**

[22] Keith Bennett, Paul Layzell, David Budgen, Pearl Brereton, Linda Macaulay, and Malcolm Munro. Service-based software: the future for flexible software. In *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*, pages 214–221. IEEE, 2000. **Cited on page 11.**

[23] Philip A Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996. **Cited on page 11.**

[24] G Bruce Berriman, Ewa Deelman, Gideon Juve, Mats Rynge, and Jens-S Vöckler. The application of cloud computing to scientific workflows: a study of cost and performance. *Phil. Trans. R. Soc. A*, 371(1983):20120066, 2013. **Cited on page 26.**

[25] 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. **Cited on page 83.**

[26] Peter Bodik, Armando Fox, Michael J Franklin, Michael I Jordan, and David A Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 241–252. ACM, 2010. **Cited on page 26.**

[27] Edward Bortnikov, Ari Frank, Eshcar Hillel, and Sriram Rao. Predicting execution bottlenecks in map-reduce clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 18–18. USENIX Association, 2012. **Cited on page 60.**

[28] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for

cloud-scale computing. In *OSDI*, volume 14, pages 285–300, 2014. **Cited on page 18.**

[29] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010. **Cited on page 24.**

[30] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. The haloop approach to large-scale iterative data analysis. *The VLDB JournalThe International Journal on Very Large Data Bases*, 21(2):169–190, 2012. **Cited on page 24.**

[31] Rajkumar Buyya. High performance cluster computing: Architectures and systems, volume i. *Prentice Hall, Upper SaddleRiver, NJ, USA*, 1:999, 1999. **Cited on page 17.**

[32] Rajkumar Buyya, Chee Shin Yeo, and Srikumar Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on*, pages 5–13. Ieee, 2008. **Cited on page 14.**

[33] 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):599–616, 2009. **Cited on page 97.**

[34] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N Calheiros. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 13–31. Springer, 2010. **Cited on pages 13 and 14.**

[35] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011. **Cited on page 5.**

[36] Maria Carla Calzarossa, Luisa Massari, and Daniele Tessera. Workload characterization: a survey revisited. *ACM Computing Surveys (CSUR)*, 48(3):48, 2016. **Cited on page 93.**

[37] Godwin Caruana, Maozhen Li, Man Qi, Mukhtaj Khan, and Omer Rana. gsched: a resource aware hadoop scheduler for heterogeneous cloud computing environments. *Concurrency and Computation: Practice and Experience*, 29(20), 2017. **Cited on page 18.**

[38] Qi Chen, Cheng Liu, and Zhen Xiao. Improving mapreduce performance using smart speculative execution strategy. *IEEE Transactions on Computers*, 63(4): 954–967, 2014. **Cited on pages 34, 35, 37, 39, and 83.**

[39] Qi Chen, Jinyu Yao, and Zhen Xiao. Libra: Lightweight data skew mitigation in mapreduce. *IEEE Transactions on parallel and distributed systems*, 26(9):2520–2533, 2015. **Cited on page 40.**

[40] 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), 2010 IEEE 10th International Conference on*, pages 2736–2743. IEEE, 2010. **Cited on pages 32 and 35.**

[41] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794. ACM, 2016. **Cited on page 120.**

[42] Dickson KW Chiu, Shing-Chi Cheung, and Sven Till. A three-layer architecture for e-contract enforcement in an e-service environment. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 10–pp. IEEE, 2003. **Cited on page 10.**

[43] VNI Cisco. Cisco visual networking index: Forecast and methodology 2014–2019 white paper. *Cisco, Tech. Rep*, 2015. **Cited on page 1.**

[44] Thomas A De Ruiter. A workload model for mapreduce. *Master Thesis, Delft University of Technology*, 2012. **Cited on page 102.**

[45] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013. **Cited on pages 28, 38, and 52.**

[46] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. **Cited on pages xvi, 5, 17, 19, 20, 59, and 83.**

[47] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014. **Cited on page 157.**

[48] Parkhil F Douglas. The challenge of the computer utility, 1966. **Cited on page 13.**

[49] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM international symposium on high performance distributed computing*, pages 810–818. ACM, 2010. **Cited on page 24.**

[50] Mohamed A El-Refaey and Mohamed Abu Rizkaa. Virtual systems workload characterization: An overview. In *Enabling Technologies: Infrastructures for Collaborative Enterprises, 2009. WETICE'09. 18th IEEE International Workshops on*, pages 72–77. IEEE, 2009. **Cited on page 17.**

[51] Richard Evans and Jim Gao. Deepmind ai reduces google data centre cooling bill by 40%. *DeepMind blog*, 20, 2016. **Cited on page 158.**

[52] ExoGENI, 2017. URL `http://www.exogeni.net/`. **Cited on pages 5, 126, and 142.**

[53] ExoGENI-Wiki, 2016. URL `https://wiki.exogeni.net/doku.php?id=public:experimenters:resource_types:start`. **Cited on pages 5 and 126.**

[54] Anja Feldmann and Ward Whitt. Fitting mixtures of exponentials to long-tail distributions to analyze network performance models. *Performance evaluation*, 31 (3-4):245–279, 1998. **Cited on page 29.**

[55] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation, 2000. **Cited on page 10.**

[56] Klaus Finkenzeller. *RFID handbook: fundamentals and applications in contactless smart cards, radio frequency identification and near-field communication*. John Wiley & Sons, 2010. **Cited on page 16.**

[57] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications*, 15(3):200–222, 2001. **Cited on page 13.**

[58] John Giannandrea Freebase data dumps, 2013. URL `https://developers.google.com/freebase/`. **Cited on page 142.**

[59] Simson Garfinkel. *Architects of the information society: 35 years of the Laboratory for Computer Science at MIT*. MIT press, 1999. **Cited on page 13.**

[60] 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):199–212, 2016. **Cited on pages 5, 91, and 151.**

[61] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2015. **Cited on page 157.**

[62] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996. **Cited on page 21.**

[63] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013. **Cited on pages 1 and 16.**

[64] Haryadi S Gunawi, Thanh Do, Joseph M Hellerstein, Ion Stoica, Dhruba Borthakur, and Jesse Robbins. Failure as a service (faas): A cloud service for large-scale, online failure drills. *University of California, Berkeley, Berkeley*, 3, 2011. **Cited on page 14.**

[65] Alonso Gustavo, F Casati, H Kuno, and V Machiraju. Web services: concepts, architectures and applications, 2004. **Cited on page 10.**

[66] Hadoop, 2016. URL `http://hadoop.apache.org/`. **Cited on pages 80 and 90.**

[67] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011. **Cited on page 18.**

[68] Ming-hao Hu, Chang-jian Wang, and Yu-xing Peng. Meeting deadlines for approximation processing in mapreduce environments. *Frontiers of Information Technology & Electronic Engineering*, 18(11):1754–1772, 2017. **Cited on page 37.**

[69] Sheng-Wei Huang, Tzu-Chi Huang, Syue-Ru Lyu, Ce-Kuen Shieh, and Yi-Sheng Chou. Improving speculative execution performance with coworker for cloud computing. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 1004–1009. IEEE, 2011. **Cited on pages 34, 35, 40, and 93.**

[70] Amazon IoT, 2017. URL `aws.amazon.com/iot`. **Cited on page 16.**

[71] Norm Jouppi. Google supercharges machine learning tasks with tpu custom chip. *Google Blog, May*, 18, 2016. **Cited on page 1.**

[72] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12. ACM, 2017. **Cited on page 1.**

[73] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production mapreduce cluster. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 94–103. IEEE, 2010. **Cited on pages 37 and 38.**

[74] Kyong Hoon Kim, Anton Beloglazov, and Rajkumar Buyya. Power-aware provisioning of cloud resources for real-time services. In *Proceedings of the 7th International Workshop on Middleware for Grids, Clouds and e-Science*, page 1. ACM, 2009. **Cited on page 26.**

[75] Wei Kuang, Laura E Brown, and Zhenlin Wang. Transfer learning-based co-run scheduling for heterogeneous datacenters. In *AAAI*, pages 4247–4248, 2015. **Cited on page 158.**

[76] Gautam Kumar, Ganesh Ananthanarayanan, Sylvia Ratnasamy, and Ion Stoica. Hold'em or fold'em?: aggregation queries under performance variations. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 7. ACM, 2016. **Cited on page 37.**

[77] Umesh Kumar and Jitendar Kumar. A comprehensive review of straggler handling algorithms for mapreduce framework. *International Journal of Grid and Distributed Computing*, 7(4):139–148, 2014. **Cited on page 34.**

[78] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 75–86. ACM, 2010. **Cited on page 39.**

[79] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. A study of skew in mapreduce applications. *Open Cirrus Summit*, 11, 2011. **Cited on page 135.**

[80] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-tune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012. **Cited on pages 40, 41, 49, 73, and 155.**

[81] Lei Lei, Tianyu Wo, and Chunming Hu. Crest: Towards fast speculation of straggler tasks in mapreduce. In *e-Business Engineering (ICEBE), 2011 IEEE 8th International Conference on*, pages 311–316. IEEE, 2011. **Cited on pages 34, 36, 41, and 155.**

[82] 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*, pages 1–14. ACM, 2014. **Cited on pages 29 and 39.**

[83] Jimmy Lin. Bespin: a library that contains implementations of big data algorithms in mapreduce and spark. *https://github.com/lintool/bespin*, 2017. **Cited on page 142.**

[84] Jimmy Lin et al. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce. In *7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, volume 1, pages 57–62. ACM Boston, MA, USA, 2009. **Cited on pages 39 and 142.**

[85] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, and Dawn Leaf. Nist cloud computing reference architecture. *NIST special publication*, 500 (2011):292, 2011. **Cited on page 14.**

[86] Tom H Luan, Longxiang Gao, Zhi Li, Yang Xiang, Guiyi Wei, and Limin Sun. Fog computing: Focusing on mobile users at the edge. *arXiv preprint arXiv:1502.01815*, 2015. **Cited on page 17.**

[87] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P King, and Richard Franck. Web service level agreement (wsla) language specification. *IBM Corporation*, pages 815–824, 2003. **Cited on page 25.**

[88] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56. ACM, 2016. **Cited on pages xix and 158.**

[89] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951. **Cited on page 102.**

[90] DW McKee, SJ Clement, Xue Ouyang, Jie Xu, Richard Romanoy, and John Davies. The internet of simulation, a specialisation of the internet of things with simulation and workflow as a service (sim/wfaas). In *Service-Oriented System Engineering (SOSE), 2017 IEEE Symposium on*, pages 47–56. IEEE, 2017. **Cited on page 14.**

[91] Amardeep Mehta, Jonas Dürango, Johan Tordsson, and Erik Elmroth. Online spike detection in cloud workloads. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 446–451. IEEE, 2015. **Cited on page 26.**

[92] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011. **Cited on page 14.**

[93] Haithem Mezni, Walid Chainbi, and Khaled Ghedira. An autonomic registry-based soa model. In *Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on*, pages 1–4. IEEE, 2011. **Cited on page 13.**

[94] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Science & Business Media, 2013. **Cited on pages 114, 115, and 119.**

[95] Minitab, 2018. URL `http://www.minitab.com/en-us/`. **Cited on page 102.**

[96] Nilo Mitra, Yves Lafon, et al. Soap version 1.2 part 0: Primer. *W3C recommendation*, 24:12, 2003. **Cited on page 13.**

[97] Ismael Solis Moreno, Renyu Yang, Jie Xu, and Tianyu Wo. Improved energy-efficiency in cloud datacenters with interference-aware virtual machine placement.

In *Autonomous Decentralized Systems (ISADS), 2013 IEEE Eleventh International Symposium on*, pages 1–8. IEEE, 2013. **Cited on page 26.**

[98] Ismael Solis Moreno, Peter Garraghan, Paul Townend, and Jie Xu. Analysis, modeling and simulation of workload patterns in a large-scale utility cloud. *IEEE Transactions on Cloud Computing*, 2(2):208–221, 2014. **Cited on page 5.**

[99] Ahsan Morshed, Prem Prakash Jayaraman, Timos Sellis, Dimitrios Georgakopoulos, Massimo Villari, and Rajiv Ranjan. Deep osmosis: Holistic distributed deep learning in osmotic computing. *IEEE Cloud Computing*, 4(6):22–32, 2018. **Cited on pages xvi and 16.**

[100] Nagios, 2017. URL `https://www.nagios.org/`. **Cited on page 56.**

[101] Vishal Ankush Nawale and Priya Deshpande. Minimizing skew in mapreduce applications using node clustering in heterogeneous environment. In *Computational Intelligence and Communication Networks (CICN), 2015 International Conference on*, pages 136–139. IEEE, 2015. **Cited on page 40.**

[102] UDDI Oasis. Version 3.0. 2. *UDDI Spec Technical Committee Draft*, 2004. **Cited on page 13.**

[103] OpenCloud, 2016. URL `http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html`. **Cited on pages 4 and 48.**

[104] OpenNebula, 2016. URL `http://opennebula.org/`. **Cited on pages 83 and 126.**

[105] Simon Ostermann, Alexandria Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. A performance analysis of ec2 cloud computing services for scientific computing. In *International Conference on Cloud Computing*, pages 115–131. Springer, 2009. **Cited on page 26.**

[106] Xue Ouyang, Peter Garraghan, Changjian Wang, Paul Townend, and Jie Xu. An approach for modeling and ranking node-level stragglers in cloud datacenters. In *Services Computing (SCC), 2016 IEEE International Conference on*, pages 673–680. IEEE, 2016. **Cited on page 97.**

[107] 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. **Cited on pages 75 and 92.**

[108] James Padgett, Karim Djemame, and Peter Dew. Grid-based sla management. In *European Grid Conference*, pages 1076–1085. Springer, 2005. **Cited on page 25.**

[109] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010. **Cited on page 158.**

[110] John Panneerselvam, Lu Liu, Nick Antonopoulos, and Yuan Bo. Workload analysis for the scope of user demand prediction model evaluations in cloud environments. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 883–889. IEEE Computer Society, 2014. **Cited on page 26.**

[111] John Panneerselvam, Lu Liu, Nick Antonopoulos, and Marcello Trovati. Latency-aware empirical analysis of the workloads for reducing excess energy consumptions at cloud datacentres. In *Service-Oriented System Engineering (SOSE), 2016 IEEE Symposium on*, pages 44–52. IEEE, 2016. **Cited on page 26.**

[112] John Panneerselvam, Lu Liu, and Nick Antonopoulos. Characterisation of hidden periodicity in large-scale cloud datacentre environments. In *Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 2017 IEEE International Conference on*, pages 496–503. IEEE, 2017. **Cited on page 26.**

[113] John Panneerselvam, Lu Liu, and Nick Antonopoulos. Inot-repcon: Forecasting user behavioural trend in large-scale cloud environments. *Future Generation Computer Systems*, 80:322–341, 2018. **Cited on page 26.**

[114] Michael P Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11), 2007. **Cited on page 11.**

[115] Mike P Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12. IEEE, 2003. **Cited on page 11.**

[116] M Pataki, M Vajna, and A Marosi. Wikipedia as text. *Ercim News - Special theme: Big Data. http://kopiwiki.dsd.sztaki.hu/*, 89:48–49, 2012. **Cited on pages 135 and 142.**

[117] Pankesh Patel, Ajith H Ranabahu, and Amit P Sheth. Service level agreement in cloud computing. 2009. **Cited on page 25.**

[118] B Thirumala Rao and LSS Reddy. Survey on improved scheduling in hadoop mapreduce in cloud environments. *arXiv preprint arXiv:1207.0780*, 2012. **Cited on page 18.**

[119] Nornadiah Mohd Razali, Yap Bee Wah, et al. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of statistical modeling and analytics*, 2(1):21–33, 2011. **Cited on page 102.**

[120] Charles Reiss, John Wilkes, and Joseph L Hellerstein. Google cluster-usage traces: format+ schema. *Google Inc., White Paper*, pages 1–14, 2011. **Cited on pages 4, 17, 27, 46, and 98.**

[121] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012. **Cited on page 26.**

[122] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. *ACM SIGCOMM Computer Communication Review*, 45(4):379–392, 2015. **Cited on pages 34 and 37.**

[123] Klaus Renzel and Wolfgang Keller. Three layer architecture. *Software Architectures and Design Patterns in Business Applications*, 1997. **Cited on page 10.**

[124] Leonard Richardson and Sam Ruby. *RESTful web services.* " O'Reilly Media, Inc.", 2008. **Cited on page 13.**

[125] Josh Rosen and Bill Zhao. Fine-grained micro-tasks for mapreduce skew-handling. *White Paper, University of Berkeley*, 2012. **Cited on pages 39, 49, and 73.**

[126] Michael Rüßmann, Markus Lorenz, Philipp Gerbert, Manuela Waldner, Jan Justus, Pascal Engel, and Michael Harnisch. Industry 4.0: The future of productivity and growth in manufacturing industries. *Boston Consulting Group*, 9, 2015. **Cited on page 1.**

[127] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Computing Surveys (CSUR)*, 42(3):10, 2010. **Cited on page 156.**

[128] Rüdiger Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 101–102. IEEE, 2001. **Cited on page 13.**

[129] Malte Schwarzkopf. Operating system support for warehouse-scale computing. *PhD. University of Cambridge*, 2015. **Cited on pages xvi, 18, and 19.**

[130] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013. **Cited on page 18.**

[131] scikit learn, 2016. URL `http://scikit-learn.org/stable/`. **Cited on pages 116 and 120.**

[132] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016. **Cited on page 16.**

[133] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010. **Cited on pages 17 and 21.**

[134] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016. **Cited on page 158.**

[135] Jure Leskovec Stanford large network dataset collection, 2017. URL `http://snap.stanford.edu/data/`. **Cited on page 135.**

[136] Le Sun, Hai Dong, and Jamshaid Ashraf. Survey of service description languages and their issues in cloud computing. In *Semantics, Knowledge and Grids (SKG), 2012 Eighth International Conference on*, pages 128–135. IEEE, 2012. **Cited on page 13.**

[137] Xiaoyu Sun, Chen He, and Ying Lu. Esamr: An enhanced self-adaptive mapreduce scheduling algorithm. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 148–155. IEEE, 2012. **Cited on page 35.**

[138] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998. **Cited on page 158.**

[139] M Sylvia and B Peterson. Success in the cloud: Why workload matters. *IBM Global Services, Somers, NY, USA, Mar*, 2012. **Cited on page 26.**

[140] William Shakespeare The Complete Works of William Shakespeare, 2017. URL `http://www.gutenberg.org/ebooks/100`. **Cited on pages 135 and 142.**

[141] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009. **Cited on pages 5 and 83.**

[142] Rafael Tolosana-Calasanz, Javier Diaz-Montes, Omer F Rana, Manish Parashar, Erotokritos Xydas, Charalampos Marmaras, Panagiotis Papadopoulos, and Liana Cipcigan. Computational resource management for data-driven applications with deadline constraints. *Concurrency and Computation: Practice and Experience*, 29 (8), 2017. **Cited on page 25.**

[143] Tsar tools, 2017. URL `https://github.com/alibaba/tsar`. **Cited on pages 56 and 59.**

[144] Cloud Computing Testbed School of Computing University of Leeds, 2018. URL `https://engineering.leeds.ac.uk/info/201325/research_and_innovation/133/research_facilities`. **Cited on page 27.**

[145] Google Cluster Data V2, 2016. URL `https://github.com/google/cluster-data`. **Cited on pages 4, 27, and 46.**

[146] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008. **Cited on page 15.**

[147] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth

Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013. **Cited on pages 17 and 70.**

[148] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015. **Cited on page 18.**

[149] Massimo Villari, Maria Fazio, Schahram Dustdar, Omer Rana, and Rajiv Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83, 2016. **Cited on page 17.**

[150] Aaron E Walsh. *Uddi, Soap, and WSDL: the web services specification reference book*. Prentice Hall Professional Technical Reference, 2002. **Cited on page 13.**

[151] Da Wang, Gauri Joshi, and Gregory Wornell. Using straggler replication to reduce latency in large-scale parallel computing. *ACM SIGMETRICS Performance Evaluation Review*, 43(3):7–11, 2015. **Cited on pages 29 and 34.**

[152] Guanying Wang, Ali R Butt, Prashant Pandey, and Karan Gupta. Using realistic simulation for performance analysis of mapreduce setups. In *Proceedings of the 1st ACM workshop on Large-Scale system and application performance*, pages 19–26. ACM, 2009. **Cited on page 25.**

[153] Jiayin Wang, Teng Wang, Zhengyu Yang, Ningfang Mi, and Bo Sheng. esplash: Efficient speculation in large scale heterogeneous computing systems. In *Performance Computing and Communications Conference (IPCCC), 2016 IEEE 35th International*, pages 1–8. IEEE, 2016. **Cited on pages 34 and 36.**

[154] 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*, pages 24–29. ACM, 2011. **Cited on pages 22 and 38.**

[155] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012. **Cited on pages 31, 41, and 155.**

[156] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, James Majors, Adam Manzanares, and Xiao Qin. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Parallel & Distributed Processing,*

*Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–9. IEEE, 2010. **Cited on page 38.**

[157] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *NSDI*, volume 13, pages 329–342, 2013. **Cited on pages 34, 35, and 39.**

[158] Yadwadkar and Wontae. Proactive straggler avoidance using machine learning. *White paper, University of Berkeley*, 2012. **Cited on pages 36 and 54.**

[159] Neeraja J Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. Wrangler: Predictable and faster jobs using fewer resources. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014. **Cited on pages 33, 34, 36, and 57.**

[160] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, and Randy Katz. Multi-task learning for straggler avoiding predictive job scheduling. *The Journal of Machine Learning Research*, 17(1):3692–3728, 2016. **Cited on page 36.**

[161] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. Wireless sensor network survey. *Computer networks*, 52(12):2292–2330, 2008. **Cited on page 16.**

[162] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Osdi*, volume 8, page 7, 2008. **Cited on pages 3, 27, 33, 34, 41, 49, 73, 83, 92, and 155.**

[163] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010. **Cited on page 25.**

[164] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010. **Cited on pages 17 and 23.**

[165] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: reducing the flow completion time tail in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 42(4):139–150, 2012. **Cited on pages 29 and 39.**

[166] 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):1393–1404, 2014. **Cited on pages 18, 34, 56, and 124.**

[167] Xia Zhao, Kai Kang, YuZhong Sun, Yin Song, Minhao Xu, and Tao Pan. Insight and reduction of mapreduce stragglers in heterogeneous environment. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013. **Cited on page 38.**