Energy-efficient Straggler Mitigation for Big Data Applications on the Clouds

Tien-Dat Phan

▶ To cite this version:

Tien-Dat Phan. Energy-efficient Straggler Mitigation for Big Data Applications on the Clouds. Performance [cs.PF]. École normale supérieure de Rennes, 2017. English. NNT : 2017ENSR0008 . tel-01669469v5

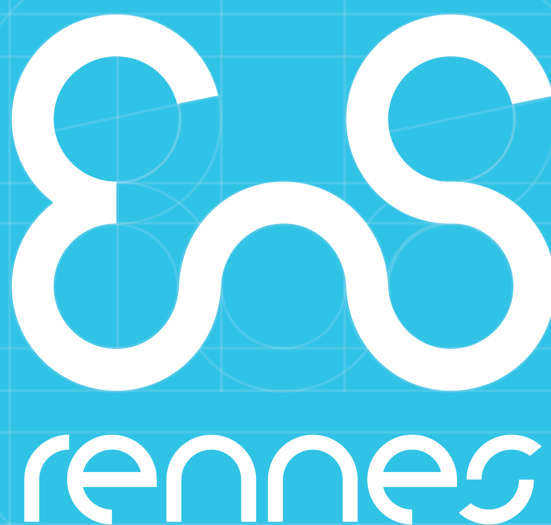
HAL Id: tel-01669469
https://theses.hal.science/tel-01669469v5

Submitted on 5 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE BRETAGNE LOIRE



THÈSE / ENS RENNES
Université Bretagne Loire

pour obtenir le titre de
DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE RENNES
Mention : Informatique
École doctorale MathSTIC

présentée par
Tien-Dat Phan
Préparée à l'unité mixte de recherche 6074
Institut de recherche en informatique
et systèmes aléatoires

Energy-efficient Straggler Mitigation for Big Data Applications on the Clouds

Thèse soutenue le 30 novembre 2017
devant le jury composé de :

Mme PEREZ Maria / *rapportrice*
Professeur des universités, Universidad Politécnica de Madrid, Spain

M. LEGRAND Arnaud / *rapporteur*
Chargé de Recherche, Laboratoire Informatique de Grenoble, France

M. PIERSON Jean-Marc / *examinateur*
Professeur des universités, Université Paul Sabatier de Toulouse, France

M. BOUGÉ Luc / *directeur de thèse*
Professeur des universités, ENS Rennes, France

M. IBRAHIM Shadi / *co-directeur de thèse*
Chargé de Recherche, INRIA Rennes - Bretagne Atlantique, France

M. ANTONIU Gabriel / *co-encadrant de thèse*
Directeur de Recherche, INRIA Rennes - Bretagne Atlantique, France

Contents

1 Introduction **1**
1.1 Context 1
1.2 Contributions 3
1.3 Publications 6
1.4 Implementations 7
1.5 Organization of the Manuscript 7

2 Background: Straggler Mitigation for Big Data Applications on the Clouds **9**
2.1 The Era of Big Data 10
2.2 Big Data Processing on the Clouds 11
2.2.1 Cloud Computing 11
2.2.2 MapReduce Programming Model 12
2.3 Energy Efficiency in Big Data Processing Systems 15
2.3.1 Energy-aware Data-layout Techniques 15
2.3.2 Energy-efficient Big Data Processing Using DVFS 16
2.3.3 Energy-efficient Resource Management 16
2.3.4 Energy-efficient Jobs/Tasks Scheduling 17
2.3.5 Exploiting Renewable Energy 17
2.4 Performance Variability and Stragglers 18
2.4.1 The Causes of Performance Variability 18
2.4.2 The Effect of Performance Variability: Stragglers 19
2.5 State-of-the-art Techniques to Mitigate Stragglers 20
2.5.1 Straggler Detection 20
2.5.2 Straggler Handling 23
2.6 Discussion: Paving the Way to Energy-efficient Straggler Mitigation 25

3 Impact of Straggler Mitigation on Performance and Energy Consumption **27**
3.1 Performance *vs.* Energy Trade-off of Speculative Execution 28
3.2 Understanding the Impact on Performance and Energy Consumption of Speculative Execution 30
3.3 Methodology Overview 31
3.3.1 Platform 31
3.3.2 Benchmarks 31
3.3.3 Hadoop deployment 33

3.4 Performance and Energy Footprints of Speculative Execution 34
3.5 Effectiveness of Speculative Execution 35
3.5.1 On the Performance Penalty of Speculative Execution 35
3.5.2 On the Power Cost of Speculative Execution 38
3.5.3 Zoom in on the Energy Impact of Speculative Execution 40
3.6 Impact of Speculative Copy Scheduling on Performance and Energy Consumption 41
3.6.1 Speculative Copies Are Delayed due to Resource Unavailability 42
3.6.2 Impact of Speculative Copy Allocation on Performance and Energy Consumption 44
3.7 Conclusion 44

4 Measuring and Enabling the Energy Efficiency of Straggler Detection 47
4.1 Energy Inefficiency of Existing Straggler Detection Mechanisms 48
4.2 A Framework to Evaluate Straggler Detection Mechanisms 49
4.2.1 Metrics for Characterizing Straggler Detection Mechanisms 50
4.2.1.1 Lack of evaluation metrics for straggler detection 51
4.2.1.2 Precision, Recall, Detection Latency and Undetected Time . . 52
4.2.2 Linking Straggler Detection Metrics to Performance 53
4.2.2.1 Architectural Models for Performance and Energy Consumption 53
4.2.2.2 On the Impact of Precision and Recall on Energy Consumption and Execution Time 54
4.2.3 Characterizing Straggler Detection Mechanisms via the Proposed Metrics 56
4.2.3.1 Experiment Setup 57
4.2.3.2 Evaluation of Straggler Detection Mechanisms 58
4.3 Hierarchical Straggler Detection: A Green Straggler Detection Mechanism . . 62
4.3.1 Design Principles 63
4.3.2 Architecture 63
4.3.3 Characterizing the Hierarchical Straggler Detection Mechanism 64
4.3.4 Evaluating the Effectiveness of Straggler Detection Mechanisms 65
4.3.4.1 Methodology 65
4.3.4.2 Impact of Straggler Detection Mechanisms with Different Resource Reservation Policies 66
4.3.4.3 Evaluation of Straggler Detection Mechanism Using Proposed Metrics 67
4.3.5 Evaluating *Hierarchical* with Different Applications and Slow-node Thresholds 70
4.3.5.1 Experimental Setup 70
4.3.5.2 Experimental Results 70
4.4 Conclusion 73

5 Energy-aware Straggler Handling for Big Data Processing Systems 75
5.1 Energy-aware Speculative Execution Controller Architecture 76
5.1.1 Allocation Problem Description 76
5.1.2 Copy Allocation Heuristic 77

5.2 Evaluation . . . 78
5.2.1 Experimental Methodology . . . 78
5.2.2 Results with the *WordCount* Application . . . 81
5.2.3 Results with the *Kmeans* Application . . . 86
5.2.4 Results with the *Sort* Application . . . 87
5.3 Conclusion . . . 89

6 Energy-efficient Resource Reservation Mechanism for Straggler Handling 91
6.1 WHEN and WHERE Questions: Impacts of the Answers . . . 92
6.1.1 When to Launch: A Fixed Solution is Not Always Good . . . 92
6.1.2 Where to Launch: Heterogeneity Has to be Considered . . . 93
6.1.3 A Motivating Example . . . 95
6.2 Design Overview . . . 96
6.3 Proposed Techniques . . . 97
6.3.1 Window-based Resource Reservation . . . 97
6.3.2 Heterogeneity-Aware Copy Allocation . . . 99
6.4 Methodology . . . 101
6.5 Experimental Evaluation . . . 104
6.5.1 Comparison of Different Speculative Execution Mechanisms . . . 104
6.5.2 Sensitivity Study . . . 108
6.6 Conclusion . . . 114

7 Conclusion 115
7.1 Achievements . . . 116
7.1.1 Characterizing the Impact of Straggler Mitigation on Performance and Energy Consumption . . . 116
7.1.2 Measuring and Enabling Energy Efficiency of Straggler Detection . . . 117
7.1.3 Bringing Energy-awareness to Straggler Handling . . . 118
7.1.4 Energy-efficient Straggler Handling Mechanism . . . 118
7.2 Perspectives . . . 119
7.2.1 Prospects Related to the *Hierarchical* Straggler Detection Mechanism . 119
7.2.2 Prospects Related to Our Straggler Handling Mechanisms . . . 120

Bibliography 123

Résumé en français 133

List of Figures

2.1 MapReduce processing overview 13
2.2 Energy-efficient techniques for processing Big Data 18
2.3 The presence of stragglers in a production cluster 20

3.1 Potential energy cost of speculative copies 29
3.2 The ratio of successful speculative copies in production clusters 30
3.3 Straggler mitigation in homogeneous environment 34
3.4 Straggler mitigation in heterogeneous environment 34
3.5 Number of successful and unsuccessful speculative copies. 36
3.6 Data skew of the *CloudBurst* application 36
3.7 The longest task execution time in homogeneous environment 37
3.8 Average power consumption in different Hadoop clusters. 38
3.9 Extra slot occupation in homogeneous environment 39
3.10 Total idle time when speculation is enabled. 39
3.11 The longest task execution times in heterogeneous environment 41
3.12 Late speculative copies due to resource unavailability 42
3.13 Average task execution time and power consumption 43
3.14 Distribution of running tasks on a node when launching copies 45

4.1 Energy inefficiency of existing straggler mitigation techniques 49
4.2 Distribution of job sizes in CMU's Hadoop production clusters 58
4.3 Distribution of task execution times of *WordCount* 59
4.4 Impact of speculative lag on straggler detection 61
4.5 *Hierarchical* straggler detection architecture. 64
4.6 Execution time with different straggler detection mechanisms 67
4.7 Energy consumption with different straggler detection mechanisms 67
4.8 Number of speculative copies with different detection mechanisms 68
4.9 The *WordCount* application with different straggler detection mechanisms. . . 71
4.10 The *Sort* application with different straggler detection mechanisms 72

5.1 Straggler ratio in Hadoop production clusters 79
5.2 Speculative execution with different copy allocation methods 81
5.3 Performance with different copy allocation methods 82
5.4 Energy consumption of different copy allocation methods 83

5.5 Energy efficiency of different copy allocation methods 84
5.6 Copy allocation with different copy allocation methods 85
5.7 Execution times of successful speculative copies 86
5.8 Speculative execution with the *Kmeans* application 87
5.9 Performance and energy when running *Kmeans* 88
5.10 Speculative execution with the *Sort* application 89
5.11 Performance and energy when running *Sort* 89

6.1 Early speculative copies do not guarantee better straggler mitigation 93
6.2 Performance variability of speculative copies in Hadoop production cluster . 94
6.3 An example with different speculative execution mechanisms. 96
6.4 Design overview of the reservation-based speculative execution mechansim. 97
6.5 Reservation-based straggler handling mechansim workflow 98
6.6 Normalized execution times with different speculation mechanisms 104
6.7 Energy consumption breakdown 105
6.8 CDF of execution time improvement 106
6.9 CDF of energy consumption improvement 107
6.10 Energy efficiency with different speculative execution mechanisms. 108
6.11 Sensitivity study on resource contention degree. 109
6.12 Sensitivity study on the hardware heterogeneity degree. 110
6.13 Sensitivity study on the straggler ratio: Performance and energy consumption 112
6.14 Sensitivity study on the straggler ratio: Energy efficiency 113
6.15 Sensitivity study on the window size parameter. 113
6.16 Straggler detection and straggler handling latency 114

List of Tables

2.1 State-of-the-art straggler mitigation techniques 21

3.1 Workload characteristics and configurations. 32

4.1 Technical terms and definitions related to speculative execution. 50
4.2 Existing metrics for evaluating straggler detection mechanisms. 51
4.3 Misleading information from existing metrics 52
4.4 Application characteristics and configurations. 58
4.5 Straggler ratio on homogeneous environment 60
4.6 The characteristics of *Default* and *LATE* 62
4.7 Characteristics of the *Hierarchical* straggler detection mechanism 65
4.8 Straggler detection with diverse resource reservation ratios 69

Chapter 1

Introduction

Contents

1.1 Context 1
1.2 Contributions 3
1.3 Publications 6
1.4 Implementations 7
1.5 Organization of the Manuscript 7

1.1 Context

WE have entered the era of Big Data where the size of data, which are daily generated, captured and processed, is increasing at an extreme rate. According to a recent study of International Data Corporation Research [133], the total amount of data generated until 2017 is approximately 16 zettabytes. In more familiar units, this amount equals to 16 trillion gigabytes or 16 billion terabytes. Moreover, the aforementioned study shows that the data size appears to stably double every two years. Accordingly, the total amount of data generated is expected to reach 180 zettabytes by 2025.

To extract useful values from such big volume of data, new processing paradigms have emerged [31, 57]. Amongst those, MapReduce [31] has become the *de-facto* programming model for processing Big Data, due to its scalability and ease of use. Hadoop, an open-source implementation of MapReduce, is currently used for processing Big Data in many academic institutions and companies [91, 113]. For instance, Yahoo! processes hundreds of petabytes of data using Hadoop annually [131]. Recently, Spark is emerging as a low-latency data processing framework by exploiting in-memory data processing [130].

To cope with the high computation and storage demand when storing and processing Big Data, these frameworks are usually deployed over large-scale infrastructures, such as public

clouds and private datacenters [77]. As an example, Facebook operates their datacenters on more than 60,000 machines to process hundreds of terabytes of data daily [105]. These systems, which carry Big Data processing frameworks on the top of large-scale infrastructures to store and process big volume of data, are referred to as *Big Data processing systems*.

At this scale, resource heterogeneity is inevitable and it exists at different levels of the systems. At the level of hardware, several hardware generations coexist in the cloud infrastructures. As a result, users have no control on the hardware components that are allocated to them [83]. At the level of application, the hardware are physically shared between different users. As a result, the resources allocated to an application are not guaranteed to provide consistent performance over the application lifetime [97, 111]. This heterogeneity, in turn, results in an evident *performance variability* [131]. On the other hand, large-scale infrastructures comprise of thousands of energy-hungry machines. These machines collectively consume a huge amount of energy, which results in a high operation cost [46]. As an example, the annual electricity consumption of Google datacenters is over 1,120 GWh, which amounts to a bill of $67 M [92].

In the future, performance variability and energy consumption will continue to be first-order concerns for the design and operation of Big Data processing systems [74, 97]. The scale of underlying infrastructures shall increase to cope with the relentless increment of data size. An increasing scale shall not only even worsen performance variability, but also dramatically increase energy consumption. For reference, the energy requirement for operating such infrastructures is expected to reach the capacity of a typical nuclear power plant [82].

In the context of Big Data, a job usually consists of a very large number of elementary tasks. The *performance* of a job is determined by the completion of its last finished task. Because of the high performance variability, the execution times of tasks can vary over a very large range within the same job. Even though the execution times of a large number of tasks remain close to the average execution time, some of them may exhibit a very large deviation. It is not unusual in practice to observe some tasks with execution times up to 8x longer than the average execution time [6]. This phenomenon is referred to as *heavy-tail* distribution [94]. This heavy-tail has a major negative impact on the job's performance [131]. In this domain, these penalizing tasks are called *stragglers*.

There exists a large body of work dedicated to avoid the occurrence of stragglers [27, 41, 104]. However, performance variability generates unexpected stragglers. In practice, it has been shown that those stragglers have major impact on the performance [131]. As a result, mitigating stragglers is a crucial objective to improve the performance of large-scale Big Data processing systems.

Many *straggler mitigation* techniques have been introduced [4, 6, 31, 131]. Typically, they consist of two phases: *straggler detection* and *straggler handling*. In the detection phase, slow tasks (*e.g.*, tasks with speed or progress below the average) are marked as stragglers [4, 6, 31, 131]. Then, the detected stragglers are handled using either *kill-restart* technique [6] or *speculative execution* technique [31]. In the former case, the straggler is killed and then re-launched in the future with the hope that it can run faster. In the latter case, a *copy* of the detected straggler is launched in parallel with the straggler. As soon as any of them completes, it is marked as *successful* and the other is *killed*. This copy is called *speculative copy* in the sense that there is no guarantee that it can finish before the straggler. Speculative execution has been currently used in many Big Data processing frameworks, such as Hadoop and Spark [130, 131]. For instance, Google reports that speculative execution improves job performance by up to 44% [31].

As we have mentioned, energy consumption is a major concern for operating Big Data processing systems. Unfortunately, speculative execution comes at a high energy cost, even though it can bring significant performance improvement. This is because the energy saved by shortening the execution time of the straggler may not be able to compensate for the extra energy consumed by the speculative copy. Even worse, the speculative copies might not finish before the stragglers and get killed. In practice, existing straggler mitigation techniques still have high ratio of killed speculative copies. In some cases, it can be up to 80% of the total speculative copies [93].

There are several reasons causing this issue. First, current straggler detection mechanisms are equipped with simple filtering algorithms in order to quickly detect stragglers at runtime [4, 6, 31, 131]. It may result in inaccurate detection decisions. For example, they may overly detect normal tasks as stragglers. Consequently, *unnecessary speculative copies* will be launched. These unnecessary speculative copies can negatively impact performance and energy consumption.

Second, a major factor is where to allocate speculative copies across the infrastructure. Performance variability can again impact the performance and energy consumption of different speculative copies allocations. Unfortunately, existing straggler handling mechanisms do not take this into consideration [4, 6, 31, 131]. They may allocate the speculative copies to non-appropriate resources, on which they have poor performance and high energy consumption.

Third, the usual implementation of speculative execution makes a difference between regular tasks and speculative copies. Once there are available resources, regular tasks are considered before taking into account speculative copies. Speculative copies may starve waiting for free resources [113]. Therefore, the speculative copies only have chance to run at the end of the job execution, when all regular tasks are launched. This long delay leaves the speculative copies less chance to be successful, while the stragglers have been running and consuming energy for a long time.

The subject of this thesis is namely to improve straggler mitigation techniques, co-optimizing performance and energy consumption.

1.2 Contributions

In this thesis, we introduce the notion of the *performance-energy efficiency* of a system. It is defined as the pair (P, E), where P stands for the execution time of the system and E stands for its energy consumption. A system is defined to be more efficient in this sense if its execution time P is shorter and its energy consumption E is smaller. In the scope of this thesis, we use the term *energy efficiency* to stand for this notion of efficiency.

This thesis aims to provide a better understanding of the impact of straggler mitigation techniques on both performance and energy consumption and to further propose new solutions to improve the *energy efficiency* of these techniques in Big Data processing systems.

Contributions Roadmap. We start by characterizing the impact of straggler mitigation on the performance and the energy consumption of Big Data processing systems. We observe that the energy efficiency of current straggler mitigation techniques could be much

improved. This motivates a detailed study of its both phases: *straggler detection* and *straggler handling*.

In terms of straggler detection, we introduce a novel framework to characterize and evaluate straggler detection mechanisms comprehensively. Accordingly, we propose a new energy-driven straggler detection mechanism. This straggler detection mechanism is implemented in Hadoop and is demonstrated to have higher energy efficiency compared to the state-of-the-art mechanisms. In terms of straggler handling, we present a new method to allocate speculative copies, which takes into consideration the impact of resource heterogeneity on performance and energy consumption. Finally, we introduce a new energy-efficient mechanism to handle stragglers. This mechanism provides more resource availability for launching speculative copies, by leveraging a dynamic resource reservation approach. It is demonstrated to bring a high improvement in energy efficiency using a discrete-event simulation.

Characterizing the Impact of Straggler Mitigation on Performance and Energy Consumption

A large body of literature has been dedicated to improving straggler mitigation techniques with respect to performance. However, little work focuses on understanding the implications of these techniques on the performance and energy consumption of Big Data processing systems. In this thesis, we rely on Grid'5000 testbed [59], *i.e.*, a highly-configurable infrastructure that supports users to perform experiments at large scale. Using Grid'5000, we conduct a set of experiments to evaluate the impact of straggler mitigation techniques on the performance and energy consumption of Hadoop. Our study reveals that straggler mitigation techniques may sometimes increase, sometimes reduce the energy consumption of Hadoop clusters. In the former case, the increase in energy consumption partially stems from the inaccuracy of existing straggler detection mechanisms used in Hadoop. This leads to a large number of unnecessary speculative copies, which in turn results in lower performance and higher energy consumption. In the latter case, the energy consumption of the system is globally reduced, because the extra energy consumption introduced by speculative copies is compensated by the energy saved by shortening the execution of the applications. Moreover, we show that the extra energy consumption varies across applications. It is contributed to by three main factors: the amount of time that speculative copies run, the idle time of machines, and the allocation of speculative copies. This work led to a publication at the DIDIS '15 conference (see [89]).

Measuring and Enabling Energy Efficiency of Straggler Detection

Despite a large body of studies targeting at improving the straggler detection mechanisms, it is unclear how to evaluate precisely them with the absence of dedicated metrics designed for this purpose. In response, we present an extended framework to characterize and evaluate straggler detection mechanisms. We start with a set of metrics, which were specifically designed to characterize straggler detection mechanisms. We then develop an architectural model by which these metrics can be used to estimate the resulting performance and energy consumption. We further conduct a series of experiments on Grid'5000 to characterize existing straggler detection mechanisms. The results indicate that existing straggler detection

mechanisms [31, 131] could be much improved. In certain cases, only 12% of the detected tasks are actual stragglers. As a result, a large number of unnecessary speculative copies is launched. These copies in turn result in a huge waste of energy. This illustrates the energy inefficiency of existing straggler detection mechanisms.

These results motivate us to introduce an energy-efficient straggler detection mechanism, called *Hierarchical*. It works as a secondary straggler filtering layer on the top of other straggler detection mechanisms. It considers tasks at the node-level while detecting stragglers. The reason for this is the fact that stragglers are mainly caused by node-level problems, *e.g.*, a node with worn-out hardware or resource contentions [6]. Accordingly, *Hierarchical* considers only the tasks on slow nodes, *i.e.*, the nodes with performance below the average. We implement this straggler detection mechanism in Hadoop and evaluate it using representative MapReduce benchmarks [1]. The results show that *Hierarchical* can reduce the energy wasted on killed speculative copies by up to 100%, while maintaining a good performance compared to the state-of-the-art straggler detection mechanisms. This work led to a publication at the Euro-Par '17 conference (see [90]).

Bringing Energy-awareness to Straggler Handling: An Energy-efficient Copy Allocation

Allocating speculative copies to different resources leads to different performance and energy consumption results. Unfortunately, very few of existing works take this into consideration while allocating speculative copies. In this work, we introduce a new straggler handling mechanism, which is equipped with an energy-efficient method to allocate speculative copies. This allocation method prioritizes the most *critical* stragglers (*i.e.*, the stragglers that are expected to have the longest remaining times) to be first handled. The copies of these critical stragglers have better chance to finish before them, compared to the copies of other stragglers. Thus, these copies can shorten the long execution time as well as reduce the high energy consumption of these stragglers. Besides, we present a performance model and an energy consumption model. These two models expose the trade-off between performance and energy consumption when allocating speculative copies on different resources. They are used to guide our speculative copy allocation method in allocating speculative copies to the appropriate resources, which can result in better performance with lower energy consumption. This speculative copy allocation method is implemented in the Hadoop framework. It can run with any straggler detection mechanism provided by Hadoop. We evaluate our speculative copy allocation method on the Grid'5000 [59] testbed using three representative MapReduce applications [1]. Experimental results show that it can reduce energy consumption while guaranteeing performance comparable to state-of-the-art copy allocation methods.

A Reservation-based Approach for Improving the Energy Efficiency of Straggler Handling

The problem of *when* to launch the speculative copies is crucial. Launching a speculative copy too late leaves it no chance to finish earlier than the straggler. However, launching copies as early as possible without considering the question of *where* to launch the copies may also result in bad outcomes. The reason for this again stems from the impact of heterogeneity, as discussed above. Launching a speculative copy on the earliest available resource,

may miss some upcoming resources, which provide better performance with lower energy consumption. Therefore, answering the *when* and *where* questions in harmony is the key to achieve better performance and lower energy consumption.

In this work, we introduce a new straggler handling mechanism, which adopts a reservation-based approach to dynamically provide the relevant resources at the appropriate time. With this straggler handling mechanism, our goal is to optimize both performance and energy consumption at runtime. First, we propose a novel performance model that relies on the execution history to estimate the execution times of new tasks or speculative copies. We also introduce a new power consumption model that takes into consideration the impact of resource contention while collocating different tasks. These two models are used to estimate the performance and energy variations for different task and copy allocation solutions, in order to achieve the performance and energy co-optimization goal. This information helps us select the best locations to launch speculative copies, hence answer the *where* question. Second, we propose a window-based reservation technique to dynamically select the best timing for launching speculative copies, considering the benefits of allocating speculative copies onto the available resources in the current window. This in turn answers the question of *when*. Our proposed solution is evaluated through a set of discrete-event simulations. The results show that it offers significant improvement in both performance and energy efficiency. This work was partially carried out during a 3-month internship at National University of Singapore.

1.3 Publications

Journal Articles

- Shadi Ibrahim, **Tien-Dat Phan**, Alexandra Carpen-Amarie, Houssem-Eddine Chihoub, Diana Moise, Gabriel Antoniu. *Governing Energy Consumption in Hadoop through CPU Frequency Scaling: an Analysis*. In the Journal of Future Generation Computer Systems (**FGCS**), *Vol. 54(C), January 2016*. Impact factor 2016: 3.997.

Papers in International Conferences

- **Tien-Dat Phan**, Shadi Ibrahim, Gabriel Antoniu, Luc Bougé. *On Understanding the Energy Impact of Speculative Execution in Hadoop*. In Proceeding of the 2015 IEEE International Conference on Data Science and Data Intensive Systems (**DSDIS '15**), *Sydney, December 2015.*

- **Tien-Dat Phan**, Shadi Ibrahim, Amelie Chi Zhou, Guillaume Aupy, Gabriel Antoniu. *Energy-Driven Straggler Mitigation in MapReduce*. In Proceedings of the 2017 International European Conference on Parallel and Distributed (**Euro-Par '17**), *Santiago de Compostela, August 2017.* CORE Rank A (acceptance rate 28%).

Posters at International Conferences

- **Tien-Dat Phan**. *Green Big Data Processing in Large-scale Clouds: Towards Energy Efficient Speculative Execution in Hadoop*. In the 2016 IEEE International Parallel & Distributed Processing Symposium (**IPDPS '16**): PhD Forum, *Chicago, May 2016.*

1.4 Implementations

Hierarchical Straggler Detection. It is a novel straggler detection mechanism which was design to work as a secondary detection layer on the top of other straggler detection mechanisms. It processes the list of stragglers detected by the underlying straggler detection mechanism as its input. Stragglers in this list is grouped by the nodes on which they are running. Stragglers that run on slow nodes, *i.e.*, nodes with performance below average, are kept in the list. The other stragglers are removed from the list. The final list is considered as the output of *Hierarchical* straggler detection mechanism. The architecture and the design of this straggler detection mechanism are mentioned in detail in Chapter 4. It is implemented in the Hadoop 1.2.1 and 2.7.3 versions. The important parameters of *Hierarchical* can be customized with Hadoop configuration files. As of now, the Hierarchical straggler detection mechanism is implemented to work on the top of the Default [31] and LATE [131] straggler detection mechanisms.

Size: 2000 lines of codes.

Language(s): Java, XML.

Energy-aware Copy Allocation. This is a copy allocation method which takes into consideration the impact of resource heterogeneity on performance and the energy consumption. The detailed design of this copy allocation method is discussed in Chapter 5. This copy allocation method is implemented as an independent straggler handling module in Hadoop. It can work with any straggler detection mechanism. Its major parameters can be easily tuned using Hadoop configuration files. Currently, we implement this copy allocation method in the Hadoop 1.2.1 version.

Size: 1500 lines of codes.

Language(s): Java, XML.

Discrete-event Straggler Mitigation Simulator. It is a discrete-event simulator which is designed to evaluate straggler mitigation mechanisms. This simulator can reproduce the behavior of a Big Data processing system, which consists of thousands of nodes and runs thousands of tasks concurrently. Additionally, this simulator allows users to easily configure many parameters using a single configuration file. It also can analyze execution traces of production clusters [93] to extract basic parameters, *e.g.*, job arrival rate, number of tasks per job, average task execution time. A detailed description of this simulator's design and implementation is presented in Chapter 6.

Size: 2000 lines of codes.

Language(s): Java, XML.

1.5 Organization of the Manuscript

The rest of this manuscript is organized as follows.

We discuss infrastructures and platforms for processing Big Data in Chapter 2. Then, we mention existing energy-efficient techniques for Big Data processing systems. Later, we discuss in detail state-of-the-art straggler mitigation techniques. Next, we discuss the energy

inefficiency of existing straggler mitigation techniques. Finally, we highlight the directions that lead to our contributions, towards high energy efficiency for straggler mitigation in Big Data processing systems.

Chapter 3 presents an in-depth experimental study to better understand the impact of straggler mitigation on the performance and energy consumption of Hadoop clusters. The findings obtained from this study form a solid base, on which we develop several contributions to improve the energy efficiency of straggler mitigation techniques.

Chapter 4 addresses the straggler detection phase. In this chapter, we introduce a comprehensive framework to characterize straggler detection mechanisms. Using this framework, we characterize and evaluate state-of-the-art straggler detection mechanisms. For instance, we reveal that existing straggler detection mechanisms have low detection precision. This results in high energy consumption due to unnecessary speculative copies. Tackling this aspect, we propose a new straggler detection mechanism. This mechanism reduces the ratio of unnecessary speculative copies. Thus, it reduces the energy wasted on unnecessary speculative copies.

Chapters 5 and 6 focus on straggler straggler handling phase. In Chapter 5, we introduce a new straggler detection mechanism. This mechanism is equipped with an energy-aware speculative copy allocation method. This method takes into account the impact of different speculative copies on both performance and energy consumption. As a result, it allocates speculative copies to resources, on which speculative copies have low energy consumption and high performance. In Chapter 6, we propose a resource reservation mechanism for handling stragglers. This resource reservation mechanism provides timely and appropriate resources for launching speculative copies. The major goal is to bi-optimize performance and energy consumption of speculative execution. Thereby, it improves the energy efficiency of straggler handling in Big Data processing systems.

Chapter 7 concludes our thesis by summarizing our contributions and discussing perspectives.

Chapter 2

Background: Straggler Mitigation for Big Data Applications on the Clouds

Contents

2.1 The Era of Big Data **10**
2.2 Big Data Processing on the Clouds **11**
2.2.1 Cloud Computing 11
2.2.2 MapReduce Programming Model 12
2.3 Energy Efficiency in Big Data Processing Systems **15**
2.3.1 Energy-aware Data-layout Techniques 15
2.3.2 Energy-efficient Big Data Processing Using DVFS 16
2.3.3 Energy-efficient Resource Management 16
2.3.4 Energy-efficient Jobs/Tasks Scheduling 17
2.3.5 Exploiting Renewable Energy 17
2.4 Performance Variability and Stragglers **18**
2.4.1 The Causes of Performance Variability 18
2.4.2 The Effect of Performance Variability: Stragglers 19
2.5 State-of-the-art Techniques to Mitigate Stragglers **20**
2.5.1 Straggler Detection 20
2.5.2 Straggler Handling 23
2.6 Discussion: Paving the Way to Energy-efficient Straggler Mitigation . . . **25**

IN this chapter, we draw a global picture of the advent and development of Big Data processing systems. Then, we dig into details about the key infrastructures and programming models, on which the Big Data applications rely. We later discuss the energy

efficiency of Big Data processing systems. Next, a literature study on the emerging energy-efficient techniques for Big Data processing systems is presented. Subsequently, we discuss the sources of performance variability in Big Data processing systems. Then, we zoom in on stragglers, which are generated by performance variability, and their impacts on both performance and energy consumption. Successively, we present state-of-the-art straggler mitigation techniques and discuss their energy efficiency. Finally, we discuss the challenges for improving the energy efficiency of straggler mitigation techniques.

2.1 The Era of Big Data

Nowadays, the increasing development of information technology facilitates our daily lives. This results in a tremendous growth of data size. This large amount of data needs to be processed to extract valuable information. The processing of these data is referred to as Big Data processing. In this section, we discuss the *major challenges* we have to face when processing Big Data. These challenges, which are abbreviated by V's of Big Data according to their names' common first letter, including: *Volume*, *Variety* and *Velocity* [69, 87, 96].

Volume. This term represents perhaps the most typical attribute which relates to the Big Data concept. This attribute, by nature, is very important as the dataset magnitude of Big Data applications is quickly increasing. This trend is motivated by the exponential growth of devices that increasingly capture and generate more data. For instance, Google is estimated to manage 15 exabytes of data (*i.e.*, 15 billion gigabytes of data) in 2015 [87]. Contributing to this huge amount of data, Youtube is roughly storing from 0.1 to 1.0 exabytes of videos. This volume is expected to increase upto 3 exabytes in 2025 [101]. At such data *Volume*, the processing and managing operations require the power of a huge datacenter, or even multiple datacenters together. To conclude, the large *Volume* of data raises the primary challenge to Big Data processing and managing systems, *i.e.*, the challenge of how to efficiently process large volume of data, across large-scale infrastructures.

Variety. The adoption of Big Data processing appears in many domains, *e.g.*, scientific activities [21, 98, 114], industrial analysis [24], educational schemes [14] and commercial transactions [106]. Besides, the raw input data are usually generated and captured from various devices, in varied formats and forms. This, by default, results in an inevitable data heterogeneity. Precisely, many Big Data applications have to concurrently process input datasets, which consist of: i) structured data (*e.g.*, relational database [30]); ii) semi-structured (*e.g.*, markup languages like XML [16], JavaScript Object Notation (JSON) [2]); and iii) unstructured data (*e.g.*, text-heavy data, audios, videos [18, 118]). As an evident outcome, processing Big Data most likely implies handling a large volume of data without predefined relation structure. At this point, the increasing data *Variety* raises a crucial challenge of how to quickly and adaptively process large amount of input data which are provided in diverse formats.

Velocity. Beside the large *Volume* and high *Variety* of data, Big Data processing systems usually have to handle data at a very fast arrival rate. For instance, a recent study has estimated that the amount of data generated daily is 2.5 exabytes [133]. Facebook receives 350

millions photos daily [78]. Disregarding this fast arrival rate of input data, many Big Data applications require fast response time in the order of sub-seconds [15]. For example, popular multiple-player online game servers, which concurrently handle hundreds of thousands of players, are requested for response time bound of 500 milliseconds [124]. In brief, the response time requirements for Big Data applications appear to get more strict, while the amount of arriving data to be processed is exponentially increasing. As a result, it raises yet another important challenge of how to improve Big Data processing systems to cope with this increasingly high *Velocity*.

2.2 Big Data Processing on the Clouds

Processing Big Data enforces a critical change in both infrastructure (to dynamically provide more computation power) as well as programming model (to efficiently execute Big Data applications across large-scale infrastructures). Hereafter, we discuss two important paradigm shifts in both aspects.

2.2.1 Cloud Computing

With respect to infrastructures, the advent of cloud computing [11] marked a key turning point in the evolution path of distributed computing infrastructures. With this computing model, users can equip their Big Data applications with significantly large computing and storage capacities, thanks to the large-scale infrastructures provided by the cloud. For example, Amazon provides computation as a service, *i.e.*, Amazon Elastic Computing Cloud (EC2) [77], which consists of millions of physical machines [36].

Besides the enormous computing and storage capacities, the cloud computing model provides a high flexibility as it adopts the *pay-as-you-go* model [77]. Accordingly, the users can easily choose the number of Virtual Machines (VMs) as needed. In addition, the specification of a VM instance, *e.g.*, number of CPU cores, memory size, can also be customized. For example, Amazon EC2 provides tens of different instance types, covering thirteen categories[1]. Each category is optimized for a specific purpose, *e.g.*, compute optimized, memory optimized, storage optimized, etc. Furthermore, many cloud providers distribute their infrastructures across the planet, over different continents [77]. Users can customize computing VMs to be geographically close to the data locations. This can reduce the data access latency of Big Data applications, which may need to simultaneously access geo-distributed datasets.

Finally, the cloud computing model offers different control degrees in deploying environments [60, 61]. Hereafter, we discuss each of these cloud service categories individually.

Infrastructure-as-a-Service — IaaS. This is considered as the most basic cloud model. With this category of cloud, users can choose their preferred Operating Systems (OS), the needed platforms and the relevant Graphical User Interfaces (GUI). Thus, they can fully customize the environment to serve their needs. This cloud suits users who have experienced in deploying customized environments. For example, Amazon EC2 provides Infrastructure-as-a-Service to millions of users worldwide [36].

[1] https://aws.amazon.com/ec2/instance-types/.

Platform-as-a-Service — PaaS. This cloud category provides a computing platform with pre-installed Operating Systems, programming language execution environment, web server and database. Users can deploy and run their self-defined software solutions without the cost and complexity of managing the underlying hardware and software layers. As an example, Apache Stratos [88] leverages the Platform-as-a-Service model to provide a cloud service that can be easily extended to run many web services.

Software-as-a-Service — SaaS. With this type of cloud service, users are granted access to application software and databases. Cloud providers take control on the infrastructure and platforms that run the applications. This approach ensures minimum back-end configurations. For instance, Microsoft Office 365 is a well-known Software-as-a-Service which is used by millions of users in academia and enterprises [115].

2.2.2 MapReduce Programming Model

Besides the advent of new computing models, new programming models have been introduced targeting Big Data processing applications. These programming models were designed to work efficiently with large-scale distributed infrastructures [31, 57]. Hereafter, we discuss the most popular programming model for Big Data processing systems, *i.e.*, MapReduce, and the current state-of-the-art Big Data processing frameworks.

MapReduce Programming Model

MapReduce is a programming model which targets efficient data processing across large-scale distributed infrastructures. It leverages the *divide-and-conquer* technique [12] in order to distribute the large amount of work across a distributed infrastructure. Precisely, each MapReduce job is split into multiple tasks. Each task is responsible to process a proportion of the job's input data. These tasks can run concurrently on different machines. The MapReduce tasks, as the name implies, consist of the tasks belonging either *Map* category or *Reduce* category.

MapReduce handles data in *key/value* structure [32]. A *Map* task, written dependingly on the application, takes an input data and processes it to generate the *intermediate* data. Intermediate data are structured as a set of intermediate key/value pairs. The *MapReduce* library groups together all intermediate values associated with the same key. These pairs are sent to the *Reduce* tasks. The execution of all job's *Map* tasks is generally called *Map* phase.

A *Reduce* task is responsible for an intermediate key or a set of keys, depending on the scale of the total key set and the granularity of the *Reduce* phase. Subsequently, it merges together the values, associated to the same key, to produce one output per key. The intermediate values are supplied to the user's reduce function in sequence. When data are too large to fit in memory, they are spilled to disks.

Figure 2.1 shows the overall flow of a MapReduce application. When the user program calls the *MapReduce* function, the following sequence of actions occurs:

1. The MapReduce library evenly divides the input files into M pieces, *i.e.*, *chunks* [31]. The size of these chunks is typically ranging from 16 megabytes to 512 megabytes (MB), depending on the hardware capacity. These chunks are distributed across the cluster in order to reduce the burden of remote data fetching.

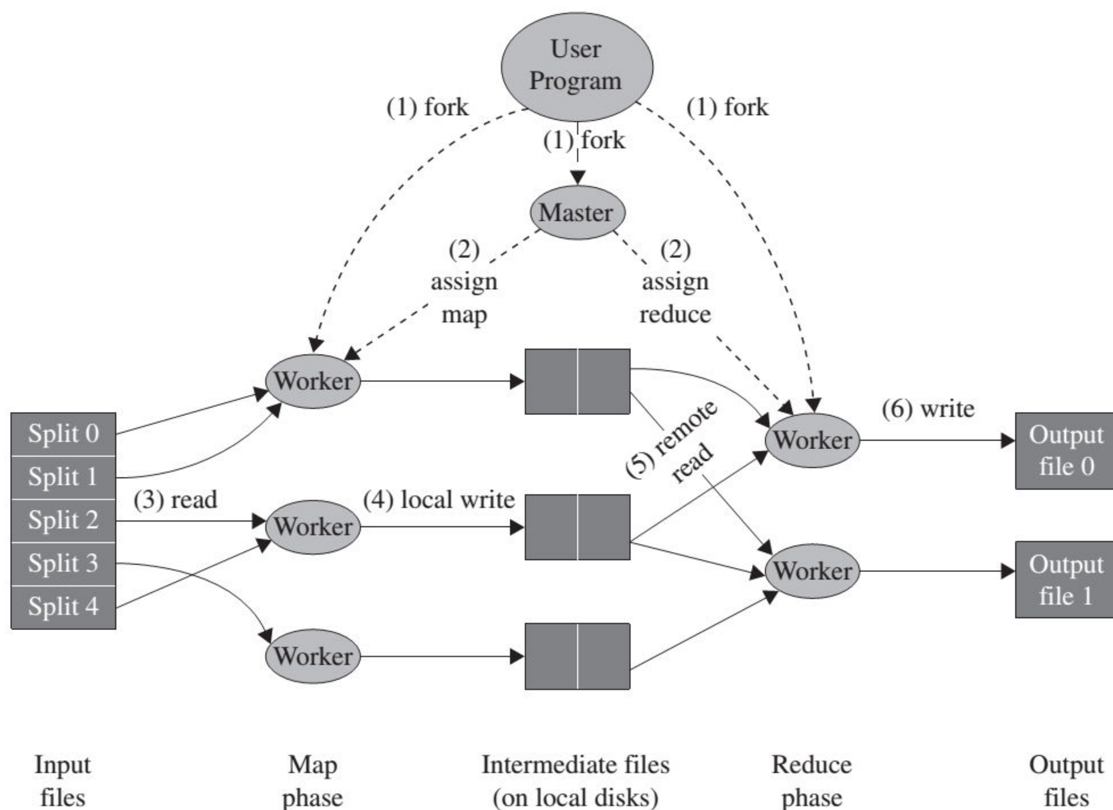


Figure 2.1 – MapReduce processing overview [62].

2. There are management processes running across the cluster of machines. There is one process called master, which is responsible for globally assigning the tasks across the system. The rest of the processes are worker processes. They are responsible for the executions of tasks on corresponding workers. The M input chunks will be respectively processed by M *Map* tasks. The master selects an idle worker node to assign it a *Map* task. This *Map* task will be next initiated on the selected worker.

3. A *Map* task reads the contents of the corresponding input chunk. The contents are parsed into key/value pairs. These pairs are next passed to the user-defined *Map* function, which is written depending on the application purposes. Finally, the output key/value pairs, *i.e.*, the output of the *Map* function, are stored as intermediate data.

4. Periodically, the intermediate key/value pairs are flushed to local disks, if the memory usage exceeds a certain threshold [31]. Typically, the key/value pairs sharing the same key are merged in order to reduce the intermediate data size. Besides, these pairs are sorted according to the keys, in order to facilitate the assignment to *Reduce* tasks. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the *Reduce* tasks. A typical *Map* task ends at this step.

5. There are R *Reduce* tasks. The value of R is controllable by the users. Once a *Reduce*

task is notified by the master with the location of intermediate data, it connects to these locations and reads the data. This reading process continues until all the intermediate data for the *Reduce* task are read. These data, which consist of key/value pairs, are sorted again by keys, in order to group the pairs with the same key.

6. The user-defined *Reduce* function is called in this step to process input data. This function returns output and add it to the final output files.

7. When all *Map* tasks as well as all *Reduce* tasks of the job are completed, the master returns to the user program to continue the execution.

Hadoop

Hadoop is an Apache open-source implementation of MapReduce [45, 49, 50, 79, 113]. Hadoop is optimized for sequential read requests, where the processing involves scanning a large amount of data [113]. Besides, inheriting from the MapReduce programming model, Hadoop is well-known with its massive scalability. Nowadays, Hadoop is widely used for Big Data processing by both academia and enterprises [91].

Hadoop runs Big Data applications on top of large-scale infrastructures. Each node is equipped with a local file system for running MapReduce programs. By default, Hadoop uses Hadoop Distributed File System [48] (HDFS) to manage data files. Hadoop adopts the master/slave architecture of the MapReduce programming model. It dedicates a master process, *i.e.*, *JobTracker*, to globally manage the execution. The node hosting this JobTracker is called master node. In addition, there are a set of worker processes, *i.e.*, *TaskTrackers*, located on every worker node. Each TaskTracker is responsible for managing the task executions on the worker node that hosts it. A TaskTracker i) collects the execution status of all tasks; ii) sends this information back to JobTracker through heartbeat messages; iii) listens to heartbeat responses from JobTracker; and iv) follows JobTracker's indication to take actions (*e.g.*, task deletion, task initiation, data read, data write).

Spark

Spark [130] is a Big Data processing framework implemeted in Scala [84]. Spark targets mainly the iterative jobs [129] (*e.g.*, machine learning jobs) and interactive analysis [130] (*e.g.*, social network streams), where tasks usually use output data as input data in subsequent iterations. Accordingly, Spark leverages the benefits of using memory to store the intermediate data in order to improve the data access speed.

To do so, a new data abstraction is introduced. It is called Resilient Distributed Dataset [129] (RDD), which represents a collection of read-only data objects partitioned across the system. These RDD objects can easily be cached in memory as needed. The cached RDDs stay in the memory and can be reused for multiple MapReduce-like operations concurrently.

As these RDD objects are distributed across large-scale systems, fault tolerance is a must in order to prevent data loss and data unavailability [35]. Therefore, Spark is equipped with a fault tolerance mechanism named *lineage*. This mechanism periodically stores the execution path, which represents the sequence of operations to reproduce the RDD [129]. By using checkpointing technique [67] to frequently store the RDDs in disks, a lost RDD can be

rebuilt by applying the sequence of operations on the most recent checkpointed RDD. With this design, RDDs represent a sweet-spot between the low latency (using memory for data access), on the one hand, and reliability as well as scalability, on the other hand.

Flink

Apache Flink [19] is an open-source project which aims to support both stream processing and batch processing within one single framework. It was designed with the idea that many modules of different data processing applications (*e.g.*, real-time analytics [15], iterative algorithms [71], batch processing [113]) can be executed as pipelined dataflows.

To do so, Flink leverages a processing paradigm that unifies all types of processing (including real-time analytics and batch processing) as one unique data-stream model. This data-stream model can seamlessly handle i) real-time processing; ii) continuous streams; or iii) historical data analysis. The major difference is the starting point along the data stream. By adopting a flexible windowing mechanism, Big Data applications running with Flink can easily compute either early and approximate results or delayed and accurate results using the same operations.

Finally, Flink is equipped with fault tolerance mechanism. Specifically, Flink provides reliable data processing using checkpointing and partial re-execution. The goal of Flink's fault tolerance mechanism is to guarantee a strict exactly-once-processing. The checkpointing mechanism periodically takes a *snapshot* of the current execution state. This reduces the recovery time when a failure occurs, as it only needs to re-execute from the latest snapshot to recover the failed process.

2.3 Energy Efficiency in Big Data Processing Systems

Energy consumption starts to severely constrain the design and the way Big Data processing systems are operated. The energy bill contributes an increasingly significant part to the total operational cost of a datacenter [46]. Moreover, overall energy consumption is continuously increasing as a result of the rapidly growing demand for computing resources to cope with the exponential growth of data.

In response, many studies have been dedicated to evaluate and improve the energy efficiency of Big Data processing systems. These studies adopt different approaches and address several levels of the systems to introduce new energy-efficient techniques. Hereafter, we systematically describe these techniques to draw a global picture of existing energy-efficient techniques for Big Data processing systems.

2.3.1 Energy-aware Data-layout Techniques

There have been several studies on evaluating and improving the energy consumption in datacenters and clouds. Many of these studies focus on power-aware data-layout techniques [3, 63, 64, 74, 103, 107], which allow servers to be turned off without affecting data availability.

GreenHDFS. Kaushik *et al.* [63] separates the HDFS cluster into hot and cold zones. The new or high-access data are placed in the hot zone. Servers in the cold zone are transitioned to the power-saving mode and data are not replicated. Only servers hosting the needed data are woken up upon the arrival of data access requests.

Rabbit. Amur *et al.* [3] introduce an energy-efficient distributed file system that maintains a primary replica on a small subset of always-on nodes (active nodes). Remaining replicas are stored on a larger set of secondary nodes, which are activated to scale up the performance or to tolerate primary failures. Rabbit provides load balancing when reading data.

Sierra. Thereska *et al.* [103] introduce a new power-proportional distributed storage system named Sierra based on cluster-based object storage pattern of Google File System [58] and Windows Azure blob store [44]. Sierra's data replication mechanism guarantees the servers to be turned off without hindering data availability. Therefore, Sierra allows users to dynamically scale in/out the cluster according to the application requirements in flight. Moreover, Sierra provides load balancing and read/write consistency.

2.3.2 Energy-efficient Big Data Processing Using DVFS

Many existing studies focus on achieving power efficiency in Hadoop clusters by using Dynamic Voltage Frequency Scaling technique (DVFS) [29, 99, 116].

TAPA. Li *et al.* [99] discuss the impact of temperature (*i.e.*, machine heat) on performance and energy of Hadoop clusters. This work is based on the observation that higher temperature of CPUs causes higher power consumption even with the same DVFS settings. Accordingly, they propose a temperature-aware power allocation (TAPA) that adjusts the CPU frequencies according to the CPU temperature. TAPA favors the maximum possible CPU frequency, thus maximizing computation capacity, without violating the power budget.

Investigation on Energy Efficiency for Computation-intensive Workloads. Wirtz and Ge [116] present a in-depth experimental study on the energy efficiency of MapReduce computation-intensive workloads. To do so, they compare the energy consumption and the performance of MapReduce applications in three settings: (1) fixed frequencies, (2) setting the frequencies to maximum frequencies when executing Map or Reduce tasks, and minimum otherwise, and (3) performance-constraint frequency settings that tolerate some performance degradation while achieving better power consumption. Experimental results indicate that significant energy savings can be achieved via judicious frequency scheduling for computation-intensive applications.

2.3.3 Energy-efficient Resource Management

There exists a large body of work which addresses the resource management for improving the energy effciency of Big Data processing systems [20, 23, 25, 55, 72].

Energy-aware VM replacement. Cardosa *et al.* [20] present Virtual Machine (VM) replacement algorithms that co-allocate VMs with similar execution characteristics to the same physical machine. The goal of this replacement is to increase the utilization of available resources. Consequently, this increases the number of idle servers that can be deactivated to save energy.

Computation vs. I/O. Chen *et al.* [25] discuss the impact of computation and I/O in MapReduce clusters on energy efficiency. They reveal that energy efficiency can be increased when using data compression to reduce the amount of data transfer, in the case data have high compression ratio (*i.e.*, the ratio between the size of data before and after being compressed). If data have low compression ratio, compressing data can result in low energy efficiency due to the extra energy consumption when compressing data.

Berkeley Energy-efficient MapReduce. Chen *et al.* [23] present an energy-efficient MapReduce workload manager motivated by empirical analysis of real-life MapReduce with Interactive Analysis (MIA) traces at Facebook. They show that interactive jobs operate on just a small fraction of the data, and thus can be served by a small pool of dedicated machines, while the less time-sensitive jobs can run in a batch fashion on the rest of the cluster.

2.3.4 Energy-efficient Jobs/Tasks Scheduling

A large number of studies has been realized to improve the energy efficiency of job/task schedulers for Big Data processing systems [26, 43, 64, 70, 80, 81].

Energy-aware MapReduce Scheduling Algorithms. Mashayekhy *et al.* [80] design a framework for improving the energy efficiency of MapReduce applications, while satisfying the service level agreement (SLA). The proposed framework relies on a greedy algorithm. This algorithm assigns tasks to the machine with the lowest energy consumption as long as their execution times do not violate the service level agreement.

All-In-Strategy. Instead of covering a set of nodes, Lang and Patel propose an all-in strategy (AIS) [70]. AIS saves energy in an all-or-nothing fashion: the entire MapReduce cluster is either on or off. All MapReduce jobs are queued until a certain threshold is reached and then all the jobs are executed with full cluster utilization.

2.3.5 Exploiting Renewable Energy

Recently, many studies exploit the benefit of renewable energy sources in powering Big Data processing systems [42, 43]

GreenHadoop. Goiri *et al.* [43] present GreenHadoop, a MapReduce framework for a datacenter powered by renewable green sources of energy (*e.g.*, solar or wind) and the electrical grid (as a backup). GreenHadoop schedules MapReduce jobs when renewable energy is available and only uses electricity to avoid time violations.

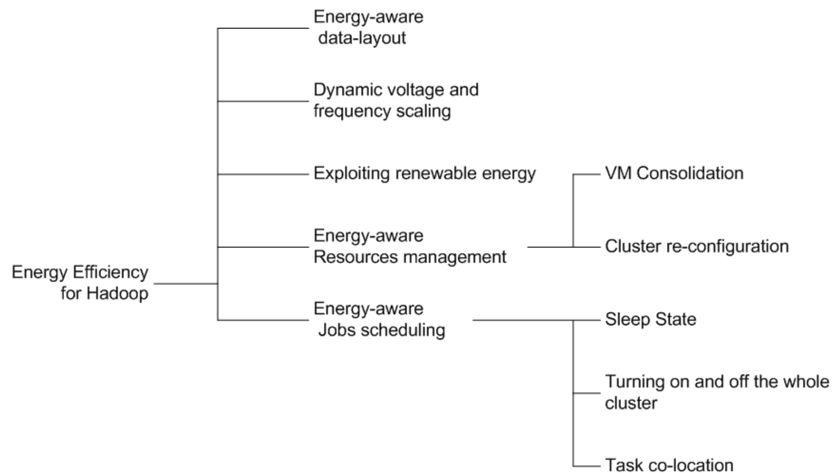


Figure 2.2 – An overview of techniques to improve energy efficiency of Big Data processing systems.

GreenSlot. GreenSlot [42] schedules the workloads to maximize the green energy consumption while meeting the jobs' deadlines. To do so, it relies on an intelligent model to predict the amount of solar energy that will be available in the near future.

Figure 2.2 summarizes existing techniques to improve energy efficiency in Hadoop.

2.4 Performance Variability and Stragglers

Different from the aforementioned energy-efficient techniques, we pay attention to performance variability and its resulting stragglers. These stragglers have high impact on the performance and energy consumption of Big Data processing systems. There exists a large number of works aiming to mitigate these stragglers. These straggler mitigation techniques are widely used in Big Data processing systems. Despite the performance benefits these techniques can bring, they may still increase the energy consumption. Our thesis tackles this issue by (1) providing in-depth understanding on the impacts of these techniques on both performance and energy consumption and (2) proposing new straggler mitigation techniques which have higher energy efficiency. In this section, we discuss in detail the root causes of performance variability. Then, we present stragglers, the emerging outcome of performance variability.

2.4.1 The Causes of Performance Variability

To meet the exponential growth of data and the huge proliferation of Big Data applications [98, 114, 132], cloud infrastructures relentlessly expand with new hardware. For instance, Facebook tripled the scale of their infrastructure in 18 months, increasing from 10,000 servers in April 2008 to 30,000 servers in October 2009 [68]. This expanding process, by nature, brings unavoidable heterogeneity to Big Data processing systems [131].

This heterogeneity exists at several levels of Big Data processing systems, and manifests itself in diverse forms. Regarding the hardware, the gradual upgrading process eventually adds new hardware models to Big Data processing systems [83]. Since typical hardware have

an average lifetime of five to seven years, it is most likely that Big Data processing systems consist of several hardware models at any arbitrary moment [33, 83]. These heterogeneous hardware have differences in either (1) processor architectures, cores and frequencies, (2) memory capacities and interconnect speeds, or (3) I/O capabilities. As different hardware models have heterogeneous performance characteristics, performance variability is a norm while executing on such infrastructures [131].

Moreover, one of the most emerging features of the cloud is *multi-tenancy* where many users are collocated on the same cloud infratructure. This results in dynamic resource allocation between collocated users. This in turn amplifies the performance variability [51]. It is observed that the round trip delay of virtualized network varies within a very large range, up to 100x difference between the minimum and the maximum values [51, 111].

Additionally, each user may concurrently execute multiple Big Data applications. By nature, these applications have different characteristics in resource consumption and performance. As a result, simultaneously executing multiple Big Data applications may lead to a high performance variability. In practice, it is recorded that the difference between the maximum and the minimum throughput can be up to 2.5x [126].

Performance variability may result in performance unpredictability [34, 110], when some tasks unexpectedly take longer time to finish compared to the average task execution time [6, 39, 131]. These long running tasks are called *stragglers*. The execution time of a Big Data application is dominated by the last completed task. Thus, long running stragglers can severely prolong the execution time of Big Data applications and increase the overall energy consumption.

2.4.2 The Effect of Performance Variability: Stragglers

In order to better understand the impact of these stragglers, it is important to know how different the execution time of a straggler compared to the average task execution time. This information is useful to further understand how much they can prolong the execution time of Big Data applications. We analyzed the traces collected in October 2012 from a Hadoop production cluster at Carnegie Mellon University (CMU) [93]. The trace contains the execution information of more than 1500 jobs, consisting of roughly 1,400,000 tasks. We use the following metric to illustrate how much different the execution time of a straggler can be, in comparison with the average task execution time [4]. This metric is calculated as follows:

$$\frac{\max_{i=1}^{N} t_i}{\frac{1}{N} \times \sum_{i=1}^{N} t_i} \tag{2.1}$$

where t_i is the execution time of the ith task and N is the number of all tasks. The value of this metric is calculated per job. Thus, there are more than 1500 values of these, corresponding to 1500 jobs. Figure 2.3 depicts the Cumulative Distribution Function (CDF) of these values. As we can observe, stragglers occur in most of the jobs. Furthermore, more than 10% of the jobs have stragglers which take up to 8x longer to finish compared to the average task execution time. These stragglers can seriously prolong the execution times. As a result, they increase the resource consumption of Big Data applications. This extra resource consumption in turn results in additional energy consumption.

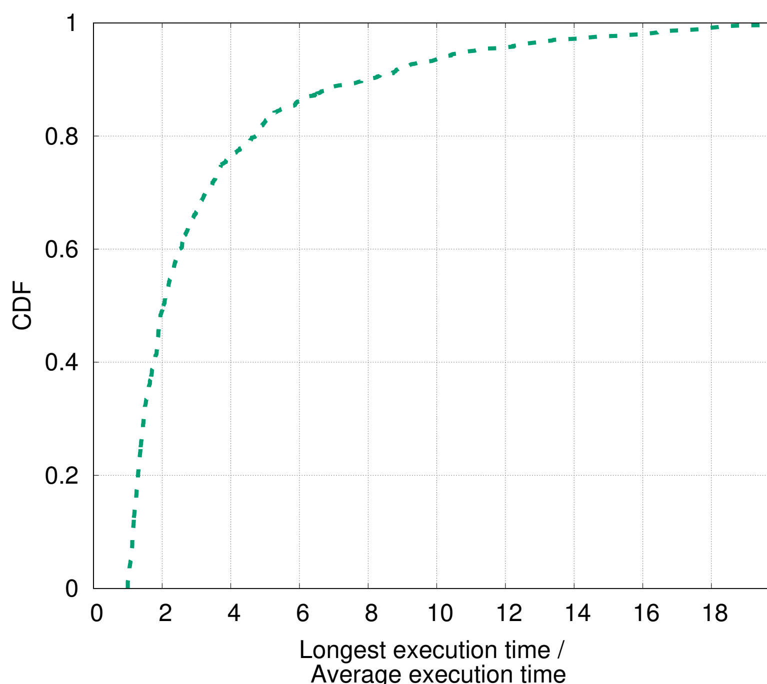


Figure 2.3 – Analyzing the traces of CMU production cluster in October 2012: We present the ratios between the job's longest task execution time compared to the job's average task execution time. The figure depicts the cumulative distribution function of this ratio for more than 1500 jobs. We observed that stragglers present in almost every job's execution.

2.5 State-of-the-art Techniques to Mitigate Stragglers

Much attention has been paid to address and mitigate stragglers. In general, a straggler mitigation technique consists of two phases: *straggler detection* and *straggler handling*. In detection phase, slow tasks, *i.e.*, tasks have progress or speed well below average, are labeled as stragglers (see Table 2.1). Upon the detection, stragglers are handled depending on the adopted technique. For instance, it can be either launching copies of these stragglers (*i.e.*, speculative execution) or killing and restarting these stragglers later (*i.e.*, kill-restart technique). These techniques are summarized in Table 2.1.

Hereafter, we discuss in detail the state-of-the-art straggler mitigation techniques, considering separately the straggler detection and straggler handling.

2.5.1 Straggler Detection

There existed a large number of studies dedicated to improve straggler detection [4, 6, 22, 31, 121, 122, 131]. Hereafter, we discuss these works in detail.

Default Straggler Detection Mechanism. Dean *et al.* [31] present a straggler detection mechanism based on *progress score*. The progress score (*i.e.*, PS) is defined as a 0-to-1 number,

Table 2.1 – Straggler mitigation techniques: State-of-the-art straggler detection and straggler handling mechanisms.

Phases	Description	Approaches/Techniques	Related works
Straggler detection	This phase searches for stragglers amongst the running tasks	Progress score based approach	[58, 86]
		Progress rate based approach	[6, 117, 122, 131]
		Progress rate and progress bandwidth at sub-task level	[22]
Straggler handling	This phase adopts a specific technique to handle detected stragglers	Speculative execution technique	[6, 58, 117, 131]
		Task cloning technique	[4, 119, 121]
		Kill-restart technique	[6]
		Cause-aware straggler handling	[6]

which represents the ratio of processed data over the total input data of a task (see Equation 2.2).

$$\mathrm{PS} = \frac{\mathrm{Size}_{\text{processed data}}}{\mathrm{Size}_{\text{total data}}} \tag{2.2}$$

where $\mathrm{Size}_{\text{processed data}}$ represents the size of data have been processed and $\mathrm{Size}_{\text{total data}}$ is the total input data size. A task is marked as a straggler, if and only if its progress score satisfies Inequation 2.3:

$$\mathrm{PS}_j < \left(\frac{\sum_{i=1}^{N} \mathrm{PS}_i}{N} - 0.2\right) \tag{2.3}$$

where PS_j is the progress score of the considered task and N is the number of total tasks within the same category (*i.e.*, Map tasks or Reduce tasks). It is important to note that the detection threshold, which is by default set to 0.2, is customizable via the configuration file. This straggler detection mechanism has shown to bring significant performance improvement, as it can reduce the job execution times by up to 44% [31].

LATE Straggler Detection Mechanism. Zaharia *et al.* [131] noticed that the progress score alone does not accurately reflect how fast a task runs as different tasks start at different moments. Therefore, they presented a new detection mechanism (*i.e.*, *LATE*) which takes into consideration both the progress score and the elapsed time (*i.e.*, the amount of time during which a task has been running). These two parameters are used to calculate the *progress rate* PR of a task, as shown in Equation 2.4:

$$\mathrm{PR} = \frac{\mathrm{PS}}{\mathrm{t}_{\text{current}} - \mathrm{t}_{\text{start}}} \tag{2.4}$$

where $\mathrm{t}_{\text{current}}$ represents the current time and $\mathrm{t}_{\text{start}}$ specifies the starting time of a task. The progress rates of all running tasks are collected at runtime. Next, these values are used to

calculate the *mean progress rate* $\overline{\text{PR}}$ and the *standard deviation* SD (as shown in Equations 2.5 and 2.6).

$$\overline{\text{PR}} = \frac{1}{N}\sum_{i=1}^{N}\text{PR}_i \tag{2.5}$$

$$\text{SD} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\text{PR}_i - \overline{\text{PR}})^2} \tag{2.6}$$

Using these values, LATE detects a task as straggler if and only if its progress rate satisfies the following inequation:

$$\text{PR}_j < \overline{\text{PR}}(1 - \alpha \times \text{SD}) \tag{2.7}$$

where PR_j denotes the current progress rate of a task and α is called the *slow task threshold*. A high value of α means that the detection mechanism considers only tasks with remarkably low progress rates as stragglers, and vice versa. The configuration file allows users to customize this value. By default, this value is set to 1.0. With this setting, LATE is expected to detect 16% of the running tasks as stragglers (assuming that the execution times of running tasks follow normal distribution [8]). It is shown that using LATE can help reduce the job execution times by up to 50%, compared to the case when straggler mitigation is disabled [131].

Recent studies [6, 22, 52–54, 121, 122] have shown that there still exist several reasons that lead to inaccurate straggler detection. For instance, Reduce tasks within the same job may have different input data sizes [22]. As a result, a task with larger input data size requires, by nature, more time to execute compared to other tasks. This unbalance in input data between tasks is called *data skew*. Detecting a task as straggler without considering the data skew will likely lead to inaccurate straggler detection [6].

Mantri Straggler Detection Mechanism. Ananthanarayanan *et al.* [6] proposed a *cause-aware* straggler detection mechanism, named Mantri. It detects stragglers using the expected execution time $\text{ET}_{\text{expected}}$ of a task. The $\text{ET}_{\text{expected}}$ of a task is the sum of that task's elapsed time and its remaining time $\text{t}_{\text{remaining}}$, which is calculated as follows:

$$\text{t}_{\text{remaining}} = \frac{1 - \text{PS}}{\text{PR}} \tag{2.8}$$

It considers a task as straggler if its expected execution time $\text{ET}_{\text{expected}}$ satisfies the following inequation:

$$\text{ET}_{\text{expected}} > 1.5 \times \text{ET}_{\text{average}} \tag{2.9}$$

where $\text{ET}_{\text{average}}$ represents the average execution of all tasks. Moreover, Mantri keeps monitoring the performance and resource consumption of running tasks. Then, it uses this information to infer the causes, which result in the slow executions of detected stragglers. Accordingly, it classifies the detected stragglers into different categories, including i) *resource contention* (*i.e.*, the competition for resources between concurrent tasks); ii) *hardware heterogeneity* (*i.e.*, different hardware provides different performance); iii) *input unavailability* (*i.e.*, the unavailable input data file prevents the task from progressing); and iv) *data skew* (*i.e.*, the unbalance in input data size between tasks). Based on this classification, Mantri introduces a cause-aware straggler handling mechanism. We will discuss this straggler handling mechanism in the next section.

Smart Straggler Detection Mechanism. Chen *et al.* [22] proposed a new straggler detection mechanism which considers different execution stages of the tasks. Typically, Map tasks consist of three stages, *i.e.*, *Read* stage, *Map* stage and *Merge* stage. By nature, each stage has different characteristics. Thereby, considering the progress rate of a running task at the stage level may improve the straggler detection precision. Besides, this work also introduces an Exponentially Weighted Moving Average (EWMA) algorithm to predict the remaining execution time of a task. Finally, these expected execution times are used to detect stragglers as Mantri (see Inequation 2.9).

Discussion. The aforementioned studies aim to improve the straggler detection. The major goal of these straggler detection mechanisms is to detect more stragglers, thus reduce the long execution tail caused by stragglers and improve the performance of Big Data applications. However, very few of these studies consider the straggler detection problem from another perspective, namely energy efficiency. Specifically, none of these works pays attention to the impact of the inaccurately detected stragglers, despite the fact that these stragglers may lead to a high number of killed speculative copies. These copies in turn result in a high cost of extra energy consumption. In this thesis, we introduce a comprehensive framework to characterize straggler detection mechanisms. The information obtained using this framework provides a detailed picture of existing straggler detection mechanisms, regarding the strengths and shortcomings of them as well as their impacts on performance and energy consumption of Big Data processing systems. Furthermore, we propose an energy-driven straggler detection mechanism. This mechanism targets a high detection precision to reduce the energy cost caused by the killed speculative copies of inaccurately detected stragglers.

2.5.2 Straggler Handling

Straggler mitigation has been drawing much attention during a long time. In the context of volunteer computing and desktop grids, resources come from different sources and have diverse performance as well as unpredictable availability [7]. As a result, applications' tasks can get either failed, prolonged or finished with incorrect output. In an effort to maintain the correctness of the output and improve the performance of applications, speculative execution is used [9, 10, 66, 76, 100]. It launches multiple copies of a task across the system. First, these multiple copies reduce the possibility of task failure. Second, different copies can provide multiple outputs for correctness confrontation. Finally, it improves the performance of the task, thus it mitigates the impact caused by stragglers.

Shortest Remaining Task First. Anglano *et al.* [9] proposed a new mechanism to speculate tasks. This mechanism uses the progress rate to estimate the remaining time of all running tasks. The task with shortest remaining time is prioritized to be speculated first. As this mechanism attempts to finish shortest tasks faster, resources are freed faster for speculating longer tasks.

Resource Prioritization and Resource Exclusion. Kondo *et al.* [66] introduce new techniques for improving the effectiveness of speculative execution. First, resources are scored according to their performance (*e.g.*, by clock rate, by the number of cycles delivered in history). Task and its copies are prioritized to run of resources with the best scores. Second,

resource exclusion technique avoids launching task and its copies on resources with low scores. The low score threshold can be determined using a preset value. These techniques are effective when there are more free resources than the number of tasks.

In Big Data processing systems, speculative execution technique continues to be the *de-facto* technique to mitigate stragglers. Accordingly, much research effort [5, 6, 22, 73, 75, 117, 120, 121, 123, 131] has been dedicated to handle stragglers and improve the performance of Big Data processing systems, mostly relying on this technique. Hereafter, we discuss these works in order to provide a global picture of state-of-the-art straggler handling mechanisms.

Default Straggler Handling Mechanism. Dean *et al.* [31] propose a straggler handling mechanism for MapReduce in 2004. It adopts the *speculative execution* technique to handle stragglers. Upon the detection of new straggler, a speculative copy is launched. This speculative copy is expected to finish earlier and reduce the straggler execution time. It is important to mention that the detected stragglers are handled with no specific order. More importantly, regular tasks have higher priority compared to speculative copies. Speculative copies are launched only when i) there are available resources and ii) all regular tasks have been launched.

LATE Straggler Handling Mechanism. Zaharia *et al.* [131] proposed a new straggler handling mechanism, named *Last Approximate Task to Execute* (LATE), which also adopts the speculative execution technique to handle stragglers. LATE uses the progress rates of detected stragglers to estimate their expected remaining times (as shown in Equation 2.8).

Based on their expected remaining times, the detected stragglers are next sorted in descending order. In other words, the straggler with longest expected remaining time is placed first. When there are free resources to launch speculative copies, the stragglers in the sorted list are considered, sequentially by their order.

Mantri Straggler Handling Mechanism. Ananthanarayanan *et al.* [6] introduced Mantri, a resource-aware straggler handling mechanism. With this straggler detection mechanism, each detected straggler is tagged with a specific cause, using the information provided by the underlying straggler detection mechanism. Mantri adopts several techniques to handle stragglers: i) killing and later restarting the straggler or ii) launching a speculative copy of the detected straggler. For making straggler handling decisions, Mantri relies on two estimations, (1) estimation of straggler finish time and (2) estimation of the execution time of new copy or restarting task. Based on these estimations, Mantri launches speculative copies or kills and restarts stragglers only when there is a fair chance of reducing the stragglers' execution times with a low resource consumption.

GRASS Straggler Handling Mechanism. Ananthanarayanan *et al.* [5] proposed GRASS, a new straggler detection mechanism addressing the approximation applications, *i.e.*, the applications that accept inaccurate outputs within an error-bound (*e.g.*, machine learning applications [28, 71, 95, 109]). As a result, this type of application allows killing on-going tasks and early finishing, as long as the outputted results has satisfied the error-bound. GRASS is a combination of two copy allocating algorithms targeting small jobs and large jobs. For small jobs, the Greedy Speculation (GS) algorithm tries to launch as much speculative copies

as possible in order to maximize the output accuracy. Consequently, small jobs can achieve higher accuracy with a small extra resource consumption. For large jobs, the Resource Aware Speculative (RAS) algorithm only launches a small number of speculative copies, in order to save the resource consumption.

Dolly Straggler Handling Mechanism. Considering that the majority of jobs in production Big Data processing clusters are small jobs, Ananthanarayanan *et al.* [4] presented Dolly, a new approach for straggler handling. Dolly launches multiple copies (*i.e.*, *clones*) of all tasks belonging to small jobs, from the beginning of the execution. As soon as the completion of one clone, the other clones are killed to free resources. Statistically, it has been proven that this mechanism can significantly reduce the possibility of having long running stragglers. In practice, Dolly results in a significant performance improvement (up to 46% speed up) with the resource budget of only 5%.

Discussion. Existing straggler detection mechanisms mainly target a better performance improvement for Big Data processing systems. Indeed, this performance-driven approach has shown to significantly improve performance. However, it is not true to expect that an improvement in performance equals to a reduction in energy consumption. This is due to the fact that straggler handling mechanisms take extra actions (*e.g.*, launching speculative copies, executing multiple clones) which come at the cost of extra energy consumption. Nonetheless, existing studies do not pay enough attention to this. As a result, it is possible for the existing straggler detection mechanism to bring high performance improvement and result in high energy cost.

This thesis first of all provides an in-depth investigation to better understand the impact of straggler mitigation on both performance and energy consumption. The obtained results confirm that existing straggler handling mechanisms can result in a high energy cost. In other words, existing straggler handling mechanisms have low energy efficiency. Addressing this, we introduce energy-aware straggler handling mechanism that takes into consideration the energy cost when allocating speculative copies. Moreover, we propose a new straggler handling mechanism, which adopts the resource reservation approach to dynamically reserve the appropriate resource at the right moment for launching speculative copies.

2.6 Discussion: Paving the Way to Energy-efficient Straggler Mitigation

As the scale of clouds keeps expanding to cope with the increasing data explosion, energy consumption will play more important role in shaping and orienting the way Big Data processing systems are operated. As an example, the datacenters across the world consumed 416.2 TWh in 2015, which is higher than the total electricity consumption of United Kingdom in the same year [17, 40, 92]. At this rate, the increasing scale of Big Data processing systems is pushing the energy consumption to the upper bound allowance [82]. Consequently, energy consumption is expected to be not merely a constraint, but rather a limit in operating Big Data processing systems in the near future [82].

In parallel, performance variability is a major issue at such scale. It is responsible for a large number of stragglers. In the context of Big Data processing systems, stragglers

can strongly impede the performance. In response, much attention has been paid to mitigate stragglers, proposing different straggler detection mechanisms and diverse straggler handling mechanisms. These mechanism have been proven to bring high performance improvement. As a result, they are widely used in Big Data processing systems. Some of them have become the default features of *de-facto* Big Data processing frameworks, such as Hadoop [113], YARN [108] and Spark [130].

However, existing straggler mitigation techniques may come at a high cost of energy. Regarding the straggler detection, existing straggler detection mechanisms are equipped with simple detection algorithms, in order to quickly detect straggler at runtime. As a result, they may have high ratio of inaccurate detection. For instance, straggler detection mechanisms can overly detect normal tasks as stragglers. This results in a high number of unnecessary speculative copies, which are in turn responsible for high energy waste as they mostly get killed. At this point, it is important to accurately characterize the existing straggler detection mechanisms, to identify their strengths and weaknesses. Furthermore, increasing the detection accuracy will be the key to improve the energy-efficiency of existing straggler mitigation mechanisms.

Considering straggler handling phase, few studies pay attention to the impact of different speculative copy allocations on both performance and energy consumption. As a result, it may reduce the potential improvements straggler mitigation can bring. Taking this impact into consideration is the key to improve the performance and energy efficiency of straggler handling mechanisms. Moreover, the success of straggler handling can be significantly reduced when resources are unavailable. Consequently, novel solutions which can dynamically provide appropriate and timely resources for launching speculative copies are needed.

Through a number of contributions, the next chapters describe in detail how we address the aforementioned challenges, in an effort to improve the energy efficiency of straggler mitigation in Big Data processing systems.

Chapter 3

Understanding the Impact of Straggler Mitigation on Performance and Energy Consumption

Contents

3.1 Performance *vs.* Energy Trade-off of Speculative Execution 28

3.2 Understanding the Impact on Performance and Energy Consumption of Speculative Execution 30

3.3 Methodology Overview 31

3.3.1 Platform 31

3.3.2 Benchmarks 31

3.3.3 Hadoop deployment 33

3.4 Performance and Energy Footprints of Speculative Execution 34

3.5 Effectiveness of Speculative Execution 35

3.5.1 On the Performance Penalty of Speculative Execution 35

3.5.2 On the Power Cost of Speculative Execution 38

3.5.3 Zoom in on the Energy Impact of Speculative Execution 40

3.6 Impact of Speculative Copy Scheduling on Performance and Energy Consumption 41

3.6.1 Speculative Copies Are Delayed due to Resource Unavailability . . . 42

3.6.2 Impact of Speculative Copy Allocation on Performance and Energy Consumption 44

3.7 Conclusion 44

NOWADAYS, Big Data applications are executed in large-scale clouds, which consist of millions of machines [36]. This large number of machines collectively consume a huge amount of energy [40, 85]. Hamilton [46] has estimated that the electricity bill has exceeded 40% of the total of datacenters. As a result, energy consumption starts to severely constrain the design and the way Big Data processing systems are operated.

In parallel, performance variability is also considered as a major concern for large-scale Big Data processing systems. It results in a large number of stragglers which in turn negatively affects both performance and energy consumption. Recently, much attention has been paid to mitigate stragglers. Speculative execution is a key technique to handle stragglers in Big Data processing systems. It launches a speculative copy of each detected straggler, with the hope that it can finish earlier and shorten the execution time. However, speculative execution is not cost-free and may result in performance degradation and extra energy consumption. Unfortunately, very few work focuses on understanding the implications of speculative execution on the performance and the energy consumption of Big Data processing systems.

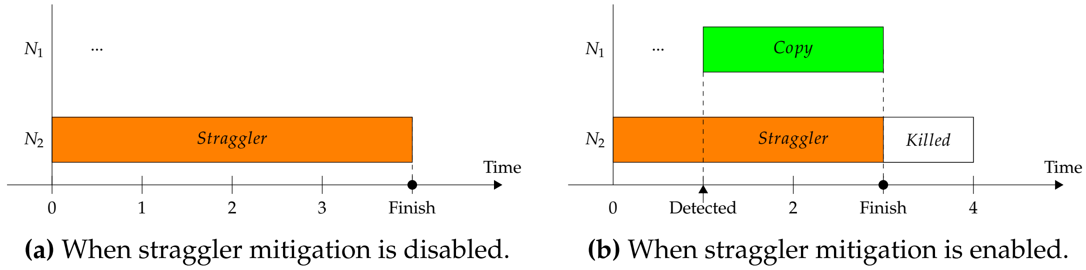
In this chapter, we provide an in-depth investigation of the impact of speculative execution on performance and energy consumption. To do so, we conduct a set of experiments on Grid'5000 with three representative Big Data applications. The acquired results are then analyzed in detail to reveal some insights into the impact on performance and energy consumption of speculative execution.

3.1 Performance *vs.* Energy Trade-off of Speculative Execution

Speculative execution is the major technique to handle stragglers in Big Data processing systems [108, 113, 130]. In practice, it can reduce the job execution time by up to 44% [31]. Although launching speculative copies may improve the performance, it comes with the cost of extra energy consumed by speculative copies. A simple example can illustrate the presence of this energy cost (see Figure 3.1).

Let us consider the scenario when running a Big Data job in a cluster of two nodes, respectively N_1 and N_2. A normal task takes 2 time units to finish. This task in turn consumes 2 energy units. During the job execution, there exists a straggler which runs on node N_2. This straggler takes 2x times longer than the normal tasks to finish, *i.e.*, 4 time units (this 2x ratio is the average ratio for stragglers recorded in many systems [6, 93]). Leaving this straggler to finish without having any mitigating action leads to a longer execution time and higher energy consumption. Briefly, without the presence of straggler mitigation, the straggler takes **4 time units** to finish and consumes **4 energy units** (see Figure 3.1a).

In contrast, when straggler mitigation is used, the straggler may be detected and a speculative copy may be launched. Existing straggler detection mechanisms leverage the progress or speed of tasks to detect stragglers [6, 31, 131]. Such information (*e.g.*, tasks' progress or speed) are used only if tasks have been running for a certain amount of time. In the example, we assume that the detection mechanisms takes one time unit to detect this straggler. Once the straggler is detected, a speculative copy of it is launched on node N_1. This node executes the copy at the normal speed. Thus, the copy finishes after two time units. Consequently, the straggler gets killed after running for 3 time units (see Figure 3.1b). In short, the speculative copy results in a shorter execution time of **3 time units**. However, considering the energy



(a) When straggler mitigation is disabled. **(b)** When straggler mitigation is enabled.

Figure 3.1 – An example to illustrate the potential energy cost of launching speculative copies.

consumption, the total energy consumption is **5 energy units**, which is the sum of the energy consumed by the killed straggler (*i.e.*, 3 energy units) and the copy (*i.e.*, 2 energy units). At this point, we can see that speculative execution can result in energy penalty even when it leads to performance improvement.

Even worse, speculative execution is not always successful. In this case, speculative copies cannot finish earlier than stragglers. Thereby, they get killed. The energy cost will be much higher in this case. In addition, these speculative copies can lead to a performance degradation, as they compete for resources with collocated regular tasks.

$$\text{Copy ratio} = \frac{\#\text{Speculative copies}}{\#\text{Total tasks}} \tag{3.1}$$

$$\text{Successful ratio} = \frac{\#\text{Successful speculative copies}}{\#\text{Total speculative copies}} \tag{3.2}$$

We analyze the traces from a production Hadoop cluster to provide more details about the presence of unsuccessful speculative copies in Big Data processing systems. These traces were collected during one month execution (*i.e.*, October 2011) of 775 jobs. We use two metrics in this analysis. One is the ratio of speculative copies to the total tasks (see Equation 3.1). The other metric is the ratio of successful speculative copies to the total speculative copies launched, as shown in Equation 3.2. The values of these two metrics are calculated per job. We then use the CDF to show the range and distribution of these values amongst the jobs (see Figure 3.2).

The results show that speculative copies contribute a considerable proportion to the total tasks and speculative copies of each job. This ratio is more than 20% for roughly 20% of the jobs. For some cases, this ratio exceeds 40%. However, only a small fraction of these speculative copies were successful. Successful speculative copies contribute less than 1% to the total speculative copies launched, in more than 60% of the jobs. This means that the majority of speculative copies were unsuccessful and got killed. This high ratio of unsuccessful speculative copies leads to a high cost of wasteful extra energy.

To this point, it is imperative to understand how much speculative execution can impact the overall performance and energy consumption of Big Data processing systems. In the rest of this chapter, we answer this question via an in-depth experimental study.

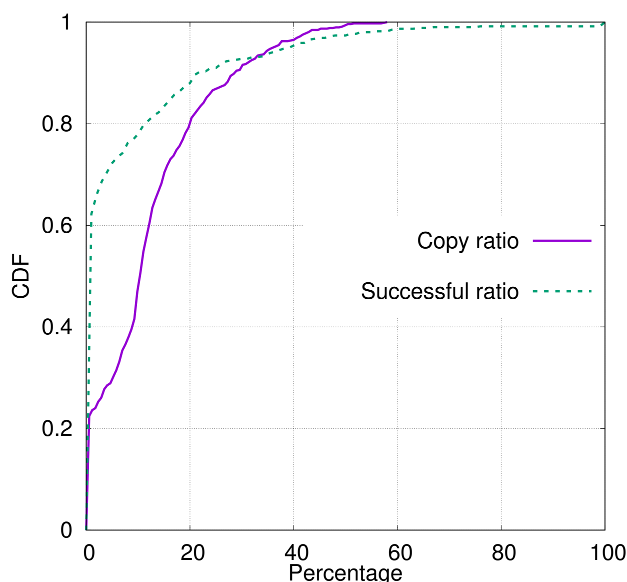


Figure 3.2 – Cumulative Distribution Function of the ratio of speculative copies and the successful ones in real production Hadoop cluster of CMU: There are 775 jobs submitted during 736 hours, running on a cluster of 145 nodes. The result shows that the speculative copies constitute a considerable part of the total tasks, but only a small fraction of them were successful.

3.2 Understanding the Impact on Performance and Energy Consumption of Speculative Execution

By the mean of experiments, our goals are i) to evaluate the impact of speculative execution in Big Data processing systems; and ii) to identify the important factors which can impact the performance and the energy efficiency when using speculative execution. In detail, we aim to answer the following questions:

- *What are the performances and energy footprints of speculative execution?* The wide adoption of Hadoop MapReduce results in a huge proliferation of Big Data processing applications [98, 114]. These applications exhibit different characteristics. Besides, Big Data processing systems are relentlessly scaling out the infrastructures to cope with the increasing data size. This results in various hardware heterogeneity degrees. In this study, we conduct experiments with three representative Big Data applications [1] on three clusters: (1) homogeneous cluster, (2) heterogeneous cluster which consists of machines with CPUs of various powers and (3) heterogeneous cluster with machines equipped with network of various bandwidths.

- *What are the factors contributing to the effectiveness of speculative execution?* As discussed, speculative execution can affect both performance and energy consumption. In this experimental study, we aim to explore the major factors which may affect the effectiveness of speculative execution. Furthermore, we quantitatively link these factors to their impact on performance and energy consumption. Finally, impact of these factors on the effectiveness of speculative execution is discussed with different applications and in clusters having different hardware heterogeneity degrees.

- *How does speculative copy scheduling affect performance and energy consumption?* Scheduling speculative copies is basically answering the questions of *when* and *where* to launch speculative copies. On the one hand, we setup experiments in which resources are not always available to early launch speculative copies. Thereby, these experiments provides useful insights of how late speculative copies affect performance and energy consumption. On the other hand, we conduct a set of experiments to imitate different speculative copy allocations. Then, we analyze the results to reveal the impact of different copy allocations on performance and energy consumption.

3.3 Methodology Overview

The experimental investigation conducted in this study focuses on exploring the implications of straggler mitigation on the energy consumption of Hadoop cluster under different workloads.

For this experimental study, having isolated and controllable results is essential. In the Clouds, the energy consumption is extremely difficult to measure as it depends on many factors (*e.g.*, how energy is distributed across different Virtual Machines within one physical machine), on which we have no control. As a result, we decided to conduct the experiments on a cluster without adopting the virtualization technique. This provides a fully controllable and highly stable environment for our experiments. However, it is important to take into consideration the resource contention and resource heterogeneity in the clouds, which are the main causes resulting in the occurrence of stragglers. With respect to this aspect, we propose a simple method of tuning the number of active cores on different machines to reproduce these two features of the cloud environment. With this method, we can produce repeatable and controllable heterogeneous environment for our experiments. Hereafter, we describe in detail the experimental environment: the platform, the used benchmarks, and the deployment setup.

3.3.1 Platform

The experiments are carried out on the Grid'5000 [59] testbed. The Grid'5000 project provides the research community with a highly-configurable infrastructure that enables users to perform experiments at large scales. The platform is spread over 10 geographical sites. For our experiments, we used nodes belonging to the Nancy site of Grid'5000. These nodes are outfitted with a 4-core Intel Xeon X3440 2.53 GHz CPU and 16 GB of RAM. Intra-cluster communication is done through a 1 Gb/s Ethernet network. As many as 40 nodes of the Nancy site are equipped with power monitoring hardware which consists of 2 Power Distribution Units (PDUs), each hosting 20 outlets. Since each node is mapped to a specific outlet, we are able to acquire coarse and fine-grained power monitoring information using the Simple Network Management Protocol (SNMP). It is important to state that Grid'5000 allows us to create an isolated environment in order to have full control over the experiments and the obtained results.

3.3.2 Benchmarks

MapReduce applications are typically categorized as CPU-intensive, I/O bound, or both. For our analysis, we selected two applications that are commonly used for benchmarking

Table 3.1 – Workload characteristics and configurations.

Application	**WordCount**	**Sort**	**CloudBurst**
Dominating phase	Map	Reduce	Reduce
Resources	CPU	Network	CPU
Input size	24.6 GB	24.5 GB	0.1 GB
Shuffle size	0.4 GB	24.5 GB	0.1 GB
Output size	0.2 GB	24.5 GB	9.7 GB

MapReduce frameworks: distributed *WordCount* and distributed *Sort* [1].

WordCount. This application counts the occurrences of each word in a large set of input files. Each Map task of *WordCount* application handles one fraction of the total input files. Each word in the input data of this Map task is used to create a key/value pair, where the key is the word and the value is 1. The Reduce tasks collect all the key/value pairs within their assigned key range. Then, they accumulate all pairs sharing the same key to return the final results. *WordCount* is a CPU-intensive application and it has a high input/output ratio. In other words, the output data size is typically small compared to the size of input data.

Sort. This application rearranges the order of data in a large collection of documents. Similar to *WordCount* application, each Map task is assigned with a fraction of the total input data. It simply reads the input data and creates the key/value pairs. The Reduce tasks collect their assigned pairs. Finally, they sort these key/value pairs and emit the output. This application consumes mainly I/O resources. It maintains the input/output data size ratio at 1 throughout its execution.

For our experiments, we used the HTML dataset collected from Wikipedia site as input data for both *WordCount* and *Sort*[1].

Real-life application. In addition to the two aforementioned benchmarks, we selected a real-life application, named *CloudBurst* [98]. *CloudBurst* is a MapReduce application designed to facilitate biological analysis. It leverages an optimized algorithm for mapping next-generation sequence data to the human genome and other reference genomes. *CloudBurst* spends most of the time in the Reduce phase to analyze the different mapping possibilities between the input and the reference genome data. During Reduce phase, this application mainly consumes CPU resources. In our experiments, we use the human genome sequence produced by the Genome Reference Consortium[2] as input data[3].

Table 3.1 summarizes the characteristics and the configurations of the three aforementioned applications. In this table, the shuffle size represents the total size of data transfer from Map tasks to Reduce tasks.

[1] http://dumps.wikimedia.org/enwiki/.
[2] http://www.ncbi.nlm.nih.gov/projects/genome/assembly/grc/.
[3] The human genome data is available at http://hgdownload.cse.ucsc.edu/goldenPath/hg38/chromosomes/.

3.3.3 Hadoop deployment

On the testbed described in Section 3.3.1, we configured and deployed a Hadoop cluster using the Hadoop 1.2.1 stable version [45]. The Hadoop cluster consists of 21 nodes. 20 nodes were used to serve as worker nodes, which store data and execute the tasks. The 21st node was configured to serve as the master node that is responsible for scheduling tasks and managing data read/write requests. In Hadoop, it leverages the notion of *slot* to specify the resource unit on which a map/ reduce task can run. We applied the default configuration of Hadoop to set the number of Map/Reduce slots that each node is capable of (2 Map slots per CPU core and 2 Reduce slots per node). Accordingly, each worker node was configured with 8 Map slots and 2 Reduce slots. The *CloudBurst* application is a reduce-intensive application and it requires mainly CPU resources. Consequently, each worker node was configured with 8 Reduce slots. Regarding the file system, we used the default file size of 64 MB and the default replication factor of three for input and output data, *i.e.*, the number of replicas of each file to be stored in the file system.

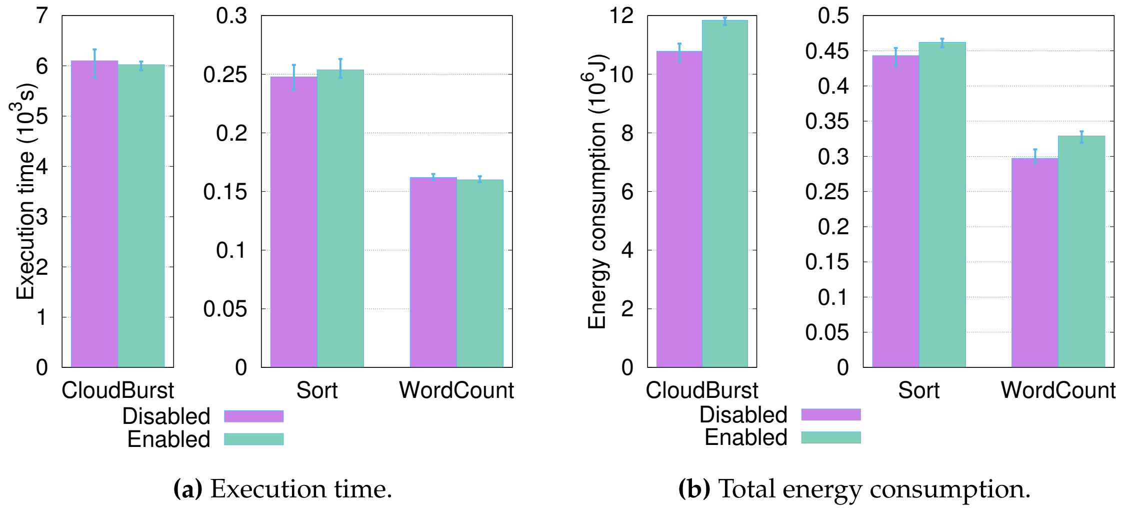
Heterogeneous Environment. Stragglers are mainly caused by resource contention and heterogeneity in Hadoop clusters. Therefore, we conduct the experiments on both homogeneous cluster and heterogeneous cluster. The homogeneous cluster, that we used, was the original cluster provided by Grid'5000. In order to produce repeatable heterogeneous environments, we created two heterogeneous Hadoop clusters.

In the first cluster, we vary the number of active cores per node from one to four. We divided the cluster into four groups. Each group consists of 25% of the total cluster nodes. All nodes in the first group were configured with 1 active core. The second group's nodes were set with 2 active cores, and so on. This setting brings a heterogeneity in CPU performance to our cluster.

In the second cluster, we vary the available network bandwidth to 25%, 50%, 75% and 100% of the maximum network bandwidth (1 Gb/s in our testbed). Similarly, we divide the cluster into four groups and the nodes in each group are configured with one of the four aforementioned network bandwidths.

While the tasks within the first cluster will exhibit variable performance due to different CPU capacities of the nodes, the tasks in the second cluster will exhibit different network access patterns according to the available network bandwidth. We run *CloudBurst* and *WordCount* on the first cluster and *Sort* on the second one.

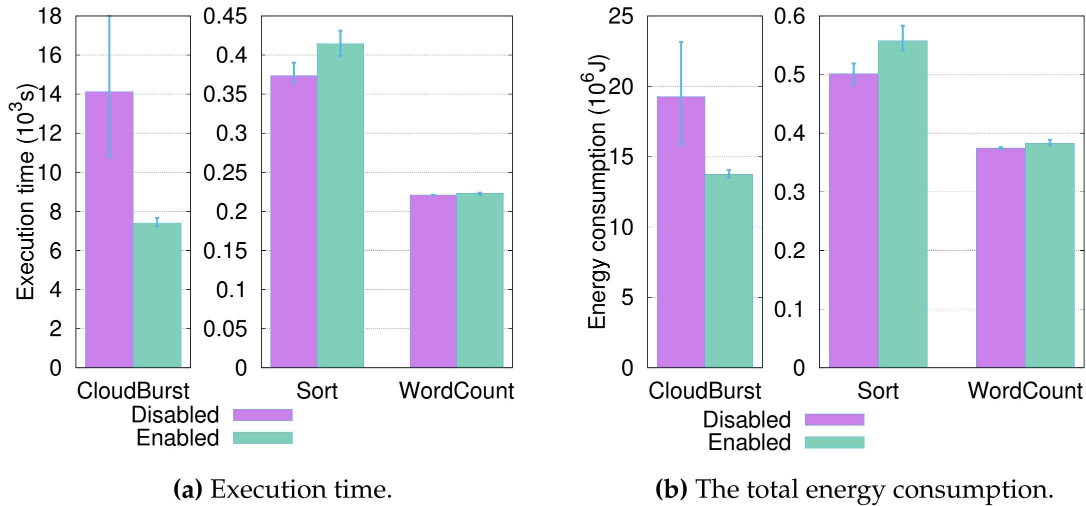
It is important to note that the total time used to conduct our experiments exceeded 40 hours on 21 nodes in Grid'5000. Each experiment was repeated three times and the average values are used in the subsequent analysis. The results presented in this chapter mainly focus on comparing between two cases, including when straggler mitigation is disabled and when straggler mitigation is enabled. For the rest of this chapter, the results when straggler mitigation is not used are denoted as *Disabled* and the results in the other case are denoted as *Enabled*. For clarification, the term straggler mitigation in this chapter is considered equal to the term speculative execution, as Hadoop adopts speculative execution technique to mitigate stragglers.



(a) Execution time.

(b) Total energy consumption.

Figure 3.3 – Application execution times and energy consumption when disabling and enabling straggler mitigation in homogeneous environment.



(a) Execution time.

(b) The total energy consumption.

Figure 3.4 – Application execution times and energy consumption when disabling and enabling speculative execution in heterogeneous environment.

3.4 Performance and Energy Footprints of Speculative Execution

In this section, we provide a high-level analysis of the experimental results we obtained. Our goal is to study the impact of speculative execution on the energy consumption of Hadoop cluster, when running the three aforementioned applications in both homogeneous and heterogeneous environments.

Homogeneous cluster. Figure 3.3b depicts the total energy consumption of the three applications when enabling and disabling straggler mitigation feature in homogeneous environments. The total energy consumption of our Hadoop cluster with straggler mitigation

enabled increases by 9.8%, 4.2% and 10.1% when running *CloudBurst*, *Sort* and *WordCount* applications, respectively. Note that the running time of both *CloudBurst* and *WordCount* applications is slightly shorter when straggler mitigation is used. Hence, these results confirm our intuition on the importance of understanding the extra energy cost of straggler mitigation in Hadoop.

Heterogeneous clusters. On the other hand, as shown in Figure 3.4b, using speculative execution as a straggler handling technique results in a significant reduction in the energy consumption of Hadoop cluster when running *CloudBurst* application. This is due to the improvement in the execution time. Surprisingly, speculative execution leads to longer execution time of both *Sort* and *WordCount* applications and therefore increases the energy consumption of Hadoop cluster.

In summary, we observe that:

- In a homogeneous environment, in contrast to expectations, speculative execution does not always reduce the execution time and results in an increase in the energy consumption of Hadoop cluster regardless of the running application.
- In a heterogeneous environment, speculative execution may substantially impact (positively or negatively) the energy consumption, depending on the application characteristics (map-intensive, shuffle-intensive, reduce-intensive) and on the type of resource heterogeneity (CPU, network bandwidth).

3.5 Effectiveness of Speculative Execution

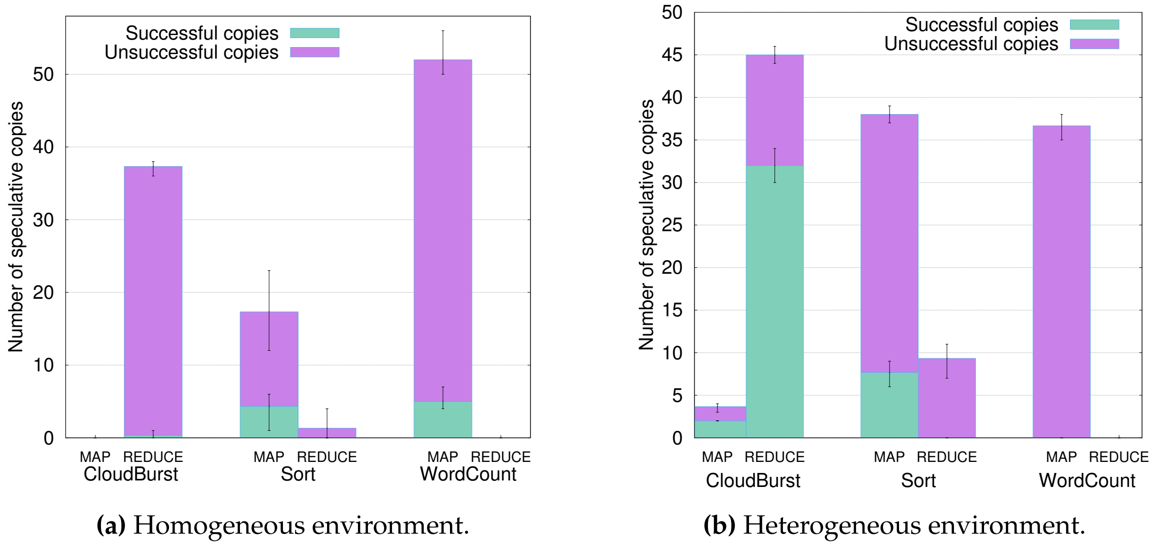
The previous section suggests that using speculative execution results in higher energy consumption of Hadoop cluster in homogeneous environment. We take a deeper look at these results to identify the main factors contributing to the energy cost of speculative execution. Then, we study the energy (reduction/increase) when using speculation by analyzing the results obtained in heterogeneous environments.

3.5.1 On the Performance Penalty of Speculative Execution

As shown in Figure 3.3a, *CloudBurst* and *WordCount* experience small improvements in terms of performance. In contrast, *Sort* suffers a slight degradation in performance when speculation is used.

Although the three applications run on homogeneous environment, Hadoop triggers a noticeable number of speculative copies, as shown in Figure 3.5a: 23% Reduce speculative copies for *CloudBurst*, 4.4% Map speculative copies for *Sort*, and 13.1% Map speculative copies for *WordCount*. However, the ratios of successful speculative copies are very low for the three applications. To understand these phenomena, we now discuss the results of each application separately.

***CloudBurst* application.** *CloudBurst* is a CPU-intensive application and the execution time is dominated by the Reduce phase. *CloudBurst* exhibits an explicit skew between different



(a) Homogeneous environment.

(b) Heterogeneous environment.

Figure 3.5 – Number of successful and unsuccessful speculative copies.

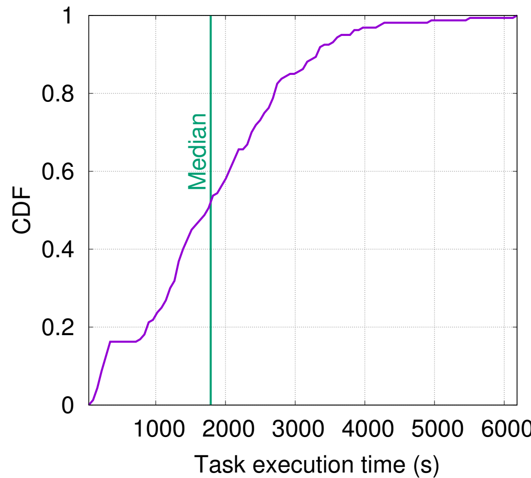
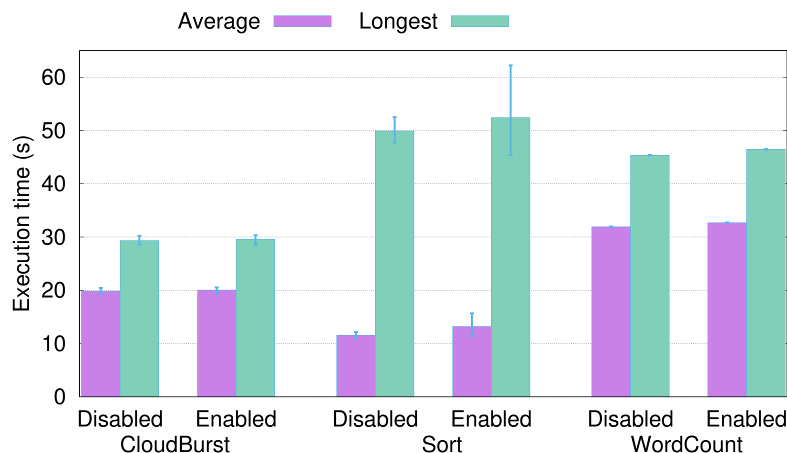


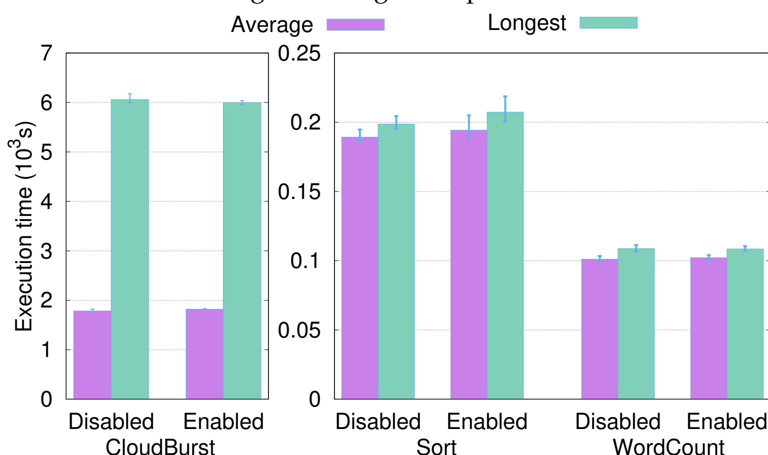
Figure 3.6 – Reduce tasks' skew in *CloudBurst* application: The longest Reduce task can take 100x times longer to finish compared to the shortest Reduce task, depending on their input data.

Reduce tasks (a similar observation is reported in [52]). Figure 3.6 clearly shows this phenomenon as the minimum and the maximum task execution times in homogeneous cluster can be up to 100x different. Disregarding this, Hadoop blindly considers long-running Reduce tasks as stragglers and therefore launches unnecessary speculative copies. As a result, Hadoop does not reduce the long execution time of stragglers in Reduce phase, as shown in Figure 3.7b.

***Sort* application.** *Sort* is a network-intensive application. The network load strongly affects the execution time. As shown in Figure 3.7b, the gap between the longest Reduce tasks and average task runtimes is relatively small, when running in homogeneous cluster. Consequently no Reduce speculative copy is launched. On the other hand, the gap between the



(a) The average and longest Map task execution times.

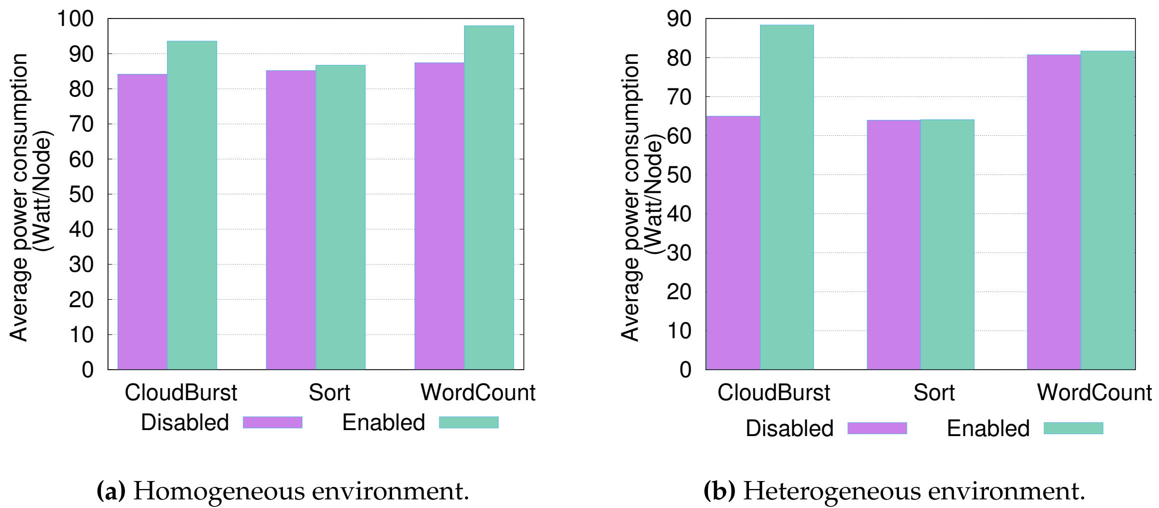


(b) The average and longest Reduce task runtimes.

Figure 3.7 – The average and longest task execution times with speculative execution disabled and enabled in homogeneous environment.

longest Map tasks and average task runtimes is big. This is mainly due to the non-local Map tasks (*i.e.*, non-local Map tasks take longer time to complete because they need to fetch their input data from remote nodes). Moreover, non-local Map tasks cause a big variation in Map task runtimes (similar observations are reported in [54]) according to the network load and the progress of the data transfer between Map tasks and Reduce tasks. Hadoop considers non-local Map tasks, which have long running times, as stragglers and launches speculative copies of them. However, 75% of the launched speculative copies are unsuccessful. Even worse, the resulted network contention leads to longer data shuffling phase of Reduce tasks (as shown in Figure 3.7b), and ends up in a longer execution time of the whole application.

***WordCount* application.** The execution time of *WordCount* is dominated by the Map phase. The gaps between the longest Map/Reduce tasks and average Map/Reduce task execution times are relatively small (as shown in Figure 3.7a and Figure 3.7b). However, the straggler detection mechanism used in Hadoop again considers non-local Map tasks as stragglers.



(a) Homogeneous environment.

(b) Heterogeneous environment.

Figure 3.8 – Average power consumption in different Hadoop clusters.

Consequently, a large number of speculative copies are launched, as shown in Figure 3.5a. The majority of these copies are unsuccessful. As a result, they do not reduce the gap of execution time between the longest task and the average task of *WordCount* application.

In summary, we observe that:

- Straggler detection mechanism in Hadoop relies on a too simplistic criterion, which does not consider the root cause of the variation in task execution times. We find that reduce-skew and non-local map tasks can lead to excessive unnecessary speculative copies.
- Unfortunately, these unnecessary speculative copies may slow down other running tasks as they compete for the resources and may result in a performance degradation.

3.5.2 On the Power Cost of Speculative Execution

A common trend can be observed: speculative execution leads to higher energy consumption in homogeneous environment. Hereafter, we present a detailed comparative discussion of the various running applications.

***CloudBurst* vs. *WordCount*.** These two applications are CPU-intensive applications and their execution times are negligibly impacted by speculative execution. As shown in Figure 3.8a, the average power consumption of a node increases by 11% (from 84.1 to 93.6 *W*) and 12% (from 87.4 to 97.9 *W*) for *CloudBurst* and *WordCount* applications when speculative execution is used, respectively. This is unexpected as the cluster resources are occupied by speculative copies contribute up to 18% of the total resource occupation time for *CloudBurst* application, and only 8% of the total resource occupation time for *WordCount* application, as shown in Figure 3.9. Intuitively, higher slot occupation will result in higher average power consumption. Furthermore, given that the idle time (*i.e.*, the time that nodes do not carry any tasks) when running *CloudBurst* decreases by 25% while it decreases by 12% when running

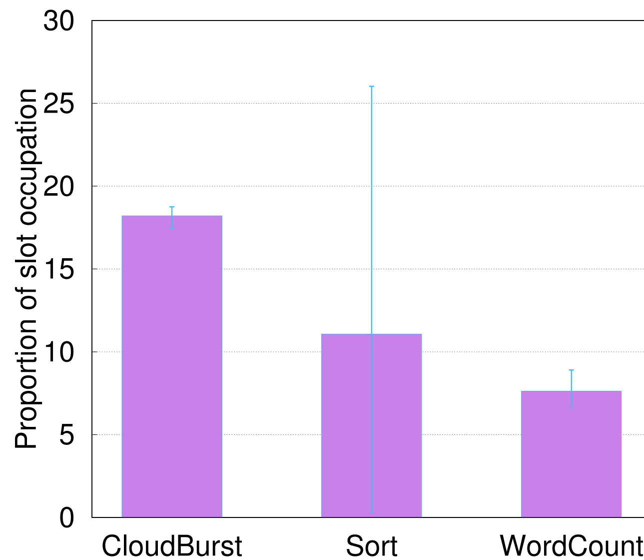
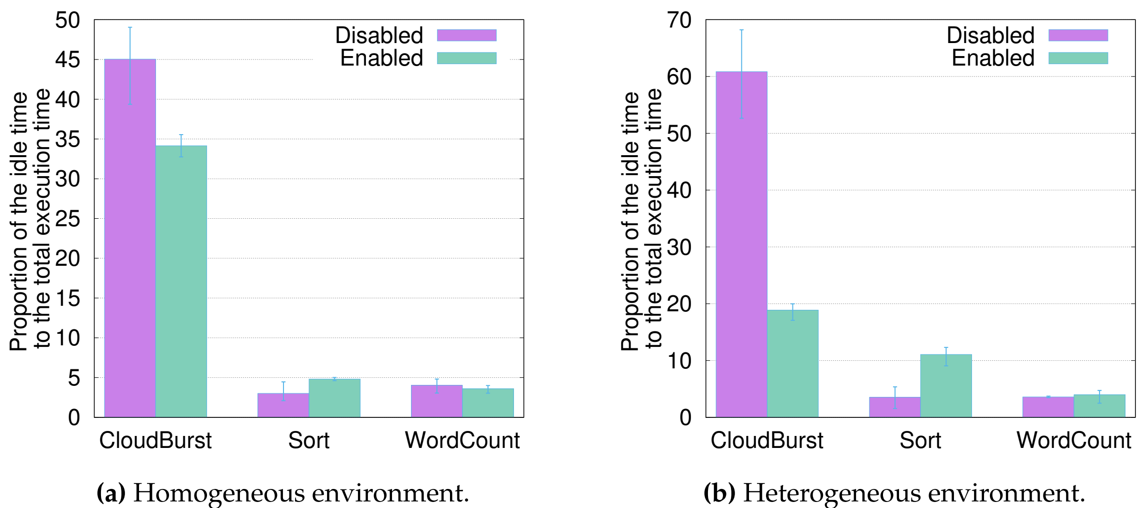


Figure 3.9 – Extra slot occupation due to speculative copies in homogeneous environment.



(a) Homogeneous environment.

(b) Heterogeneous environment.

Figure 3.10 – Total idle time when speculation is enabled.

WordCount (as shown in Figure 3.10a), it is expected to obtain higher increase when running *CloudBurst* application compared to *WordCount*. This leads us to the observation that the power cost of launching speculative copies varies according to the load of nodes. This in turn can strongly impact energy cost of speculative execution. This observation motivates us to further look at the power cost of launching speculative copies across nodes with different loads (see Section 3.6).

Sort. On the other hand, the slots which are occupied by speculative copies account for 11% of the execution times of the *Sort* application and results in only 1.8% increase in the average power consumption. This can be explained due to the increase in the idle time (as

shown in Figure 3.10a) and to low CPU usage exhibited in *Sort* (*i.e.*, the average CPU usage is almost 25% [56]). Thus, the increase in energy consumption when running *Sort* is strongly related to the increase in execution time.

In summary, we observe that:

- The energy consumption of a Hadoop cluster varies according to the running time of the applications and to the energy cost of speculation execution.
- The energy cost of speculative execution is proportional to the increase in the average power consumption in the cluster, which strongly depends on the duration of the unnecessary speculative copies (*i.e.*, extra slot occupation), on the resulted idle time, and on the allocation of speculative copies.
- The extra slot occupation and idle time are the major contributors to the extra power cost of speculative execution when running I/O bound application (*i.e.*, *Sort*), but they have less impact on the extra power cost of speculative execution when running CPU-intensive applications (*i.e.*, *CloudBurst*).

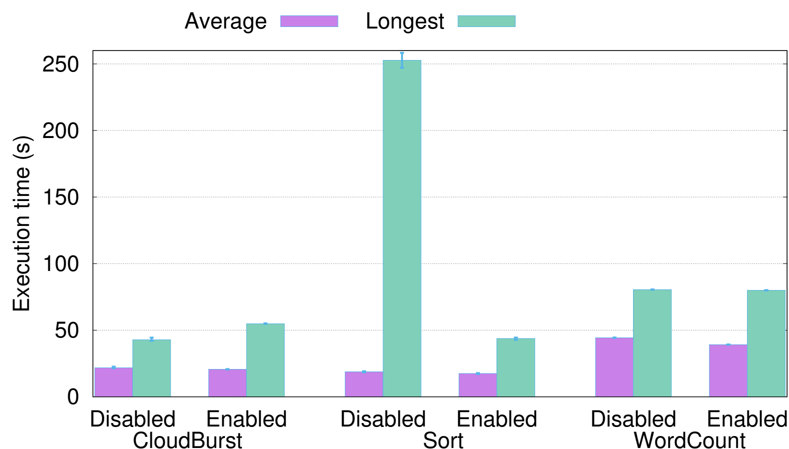
3.5.3 Zoom in on the Energy Impact of Speculative Execution

Speculative execution results in a significant reduction in the energy consumption of Hadoop cluster when running *CloudBurst* in heterogeneous clusters. We observe 28.7% energy reduction (as shown in Figure 3.4b). The significant reduction in execution time is the major contributor to this energy reduction (*i.e.*, the execution time is decreased by 47.6% as shown in Figure 3.4a). This is due to the high ratio of successful speculative copies (see Figure 3.5b): the ratio of successful speculative copies is 54.5% and 70.2% for Map tasks and Reduce tasks, respectively. More importantly, these successful speculative copies improve the average task execution times of Reduce tasks (the average task runtime is decreased by 32.7%) and reduce the execution time of the longest task by 48.7% (see Figure 3.11b). However, we can still see the natural skew-reduce issue: the gap between the longest Reduce task and the average Reduce task execution time is almost the same as in homogeneous cluster (see Figure 3.7b and 3.11b).

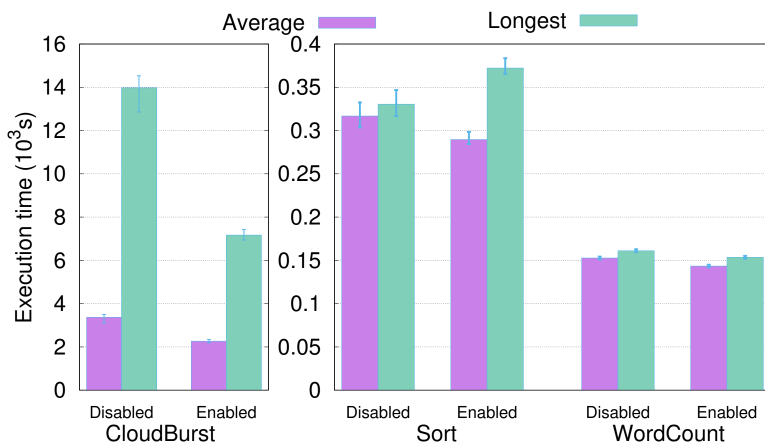
It is clear that the reduction in the energy consumption is not proportional to the execution time reduction. This is due to the 32% increase (*i.e.*, from 64.9 to 88.4 *W* as shown in Figure 3.8b) in average power consumption (*i.e.*, extra power consumption caused by speculative execution) and the significant decrease in idle time, see Figure 3.10b.

On the other hand, we can observe that in the case of *Sort* and *WordCount*, the Hadoop cluster consumes more energy when speculative execution is used. These results can be explained by the increase in the execution time of the two applications and by the increase in average power consumption per node due to the executions of speculative copies. The average power consumption increases from 63.9 to 64 *W* when running *Sort* and from 80.7 to 81.7 *W* when running *WordCount* application.

It is important to mention that speculative execution successfully reduces the longest map task by almost 80% in case of the *Sort* application, see Figure 3.11a. But, due to the high number of speculative Reduce tasks (all were not successful) and the resulted network contention, we observe an increase in the execution time of the longest Reduce task by almost 12%. As *Sort* is dominated by the completion of the last Reduce tasks, this results in longer execution time.



(a) The average and longest map task execution times.



(b) The average and longest reduce task execution times.

Figure 3.11 – The average and longest task execution times in heterogeneous environment.

In summary, we confirm that speculative execution — when necessary — can effectively mitigate stragglers, but may not necessarily reduce the overall execution times of the applications. Moreover, we observe that the reduction in energy consumption is only achieved when the execution time of the application is noticeably reduced. However, this reduction is not proportional to the performance improvement.

3.6 Impact of Speculative Copy Scheduling on Performance and Energy Consumption

When and *Where* to allocate speculative copies affect the performance and energy consumption of speculative execution. In this section, we first discuss how existing speculative execution mechanisms answer the question of *when*. Then, we discuss the resulting impact on performance and energy consumption. Second, we leverage the importance of how to answer the *where* question to the effectiveness of speculative execution. Accordingly, we in-

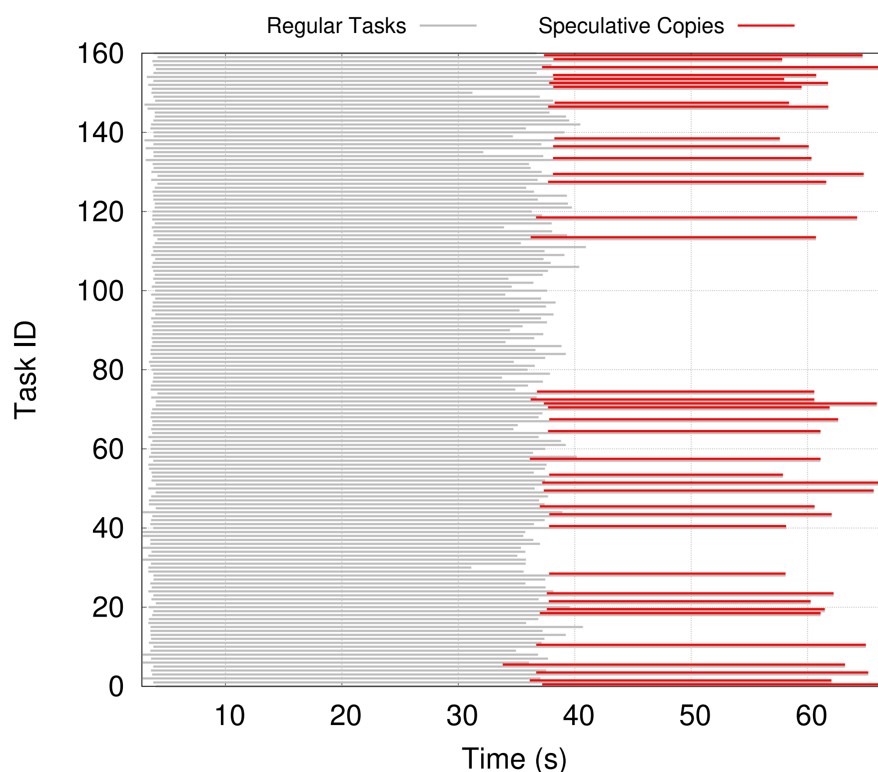


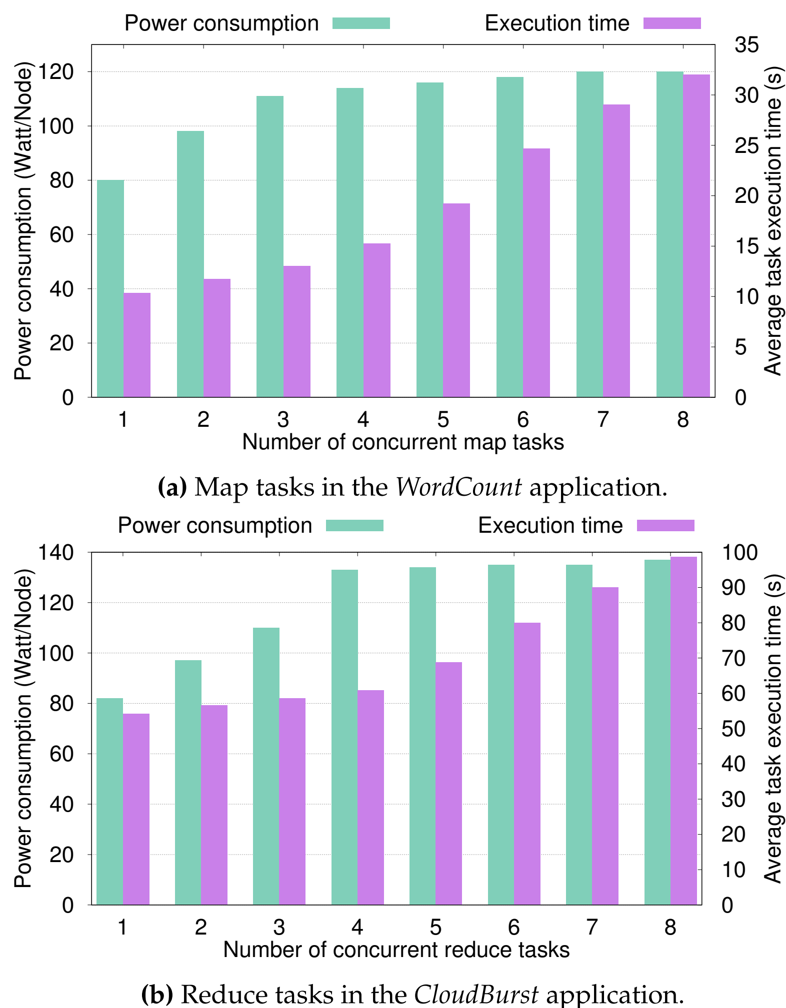
Figure 3.12 – *WordCount* application in heterogeneous environment. This figure uses lines to depict the executions of regular tasks and speculative copies. Due to the unavailability of slots, speculative copies has to wait until the completion of some regular tasks.

vestigate the impact of different copy allocations on performance and energy consumption.

3.6.1 Speculative Copies Are Delayed due to Resource Unavailability

In Hadoop, regular tasks are first considered to be launched when there are free slots. Speculative copies are only considered when all regular tasks have been launched. With this policy, speculative copies can be delayed due to resource unavailability as all slots are occupied by regular tasks. To illustrate this phenomenon, we conduct an experiment with *WordCount* application in our heterogeneous cluster. The input size of this *WordCount* job is 10 GB of data. Accordingly, this *WordCount* job comprises 160 Map tasks. These regular tasks occupy the whole cluster, which has 160 Map slots. As Figure 3.12 shows, the speculative copies have to wait until the completion of some regular tasks to be able to start.

This delay leaves the speculative copies less chance to successfully finish, as the stragglers have been running for a long time. Moreover, these stragglers also consume energy during that time. As a result, this delay also affects the effectiveness of speculative execution, with respect to energy consumption. As shown in Figure 3.12, speculative execution has to wait for 34 seconds until the first speculative copies can be launched. These speculative copies take in average 30 seconds to finish. As a result, stragglers are killed after running for 64 seconds in average. Assume that resources are earlier available. For instance, resources are available after 15 seconds of running. In this case, stragglers are killed after 45



(a) Map tasks in the *WordCount* application.

(b) Reduce tasks in the *CloudBurst* application.

Figure 3.13 – The average task execution time and power consumption when varying the number of concurrent running tasks.

seconds of running. This equals to an improvement of 30%. Besides, the energy consumed by stragglers are also reduced. Unfortunately, the unavailability of resources prevents speculative execution from reaching this improvement.

Even worse, Big Data jobs can consist of thousands of tasks [93]. The regular tasks of these large jobs can occupy the cluster for a very long time. As a result, the delay for speculative copies will be more significant in this case. This raises yet another important problem of resource unavailability for speculative execution.

We notice that the execution of speculative copies may be delayed when regular tasks occupy the whole cluster. This delay can lead to unsuccessful speculative copies which have negative impact on both performance and energy consumption.

3.6.2 Impact of Speculative Copy Allocation on Performance and Energy Consumption

In this section, we study the impact of speculative copy allocation on power consumption *at node level*. To do so, we first study the average power consumption of a node in a homogeneous Hadoop cluster when varying the number of concurrent running tasks for the same application. Here we show the results when varying the number of concurrent reduce tasks for *CloudBurst*. As shown in Figure 3.13b, our results indicate that the average power usage of a node gradually increases when increasing the number of concurrent tasks from 1 to 4 and it remains the same when the number of concurrent tasks is > 4. On the contrary, the average task execution time slightly increases when the number of concurrent tasks is ≤ 4. The difference is higher when the number of concurrent tasks is > 4. The same behavior is observed while varying the number of concurrent map tasks of *WordCount* application (as shown in Figure 3.13a).

In summary, we find a clear trade-off between performance and power consumption when scheduling a speculative copy. Launching speculative copies on idle nodes or nodes with a small number of running tasks result in lower average task execution time but leads to higher additional power consumption per node. In contrast, launching speculative copies on nodes with larger number of running tasks results in higher average task execution time but lower additional power consumption per node.

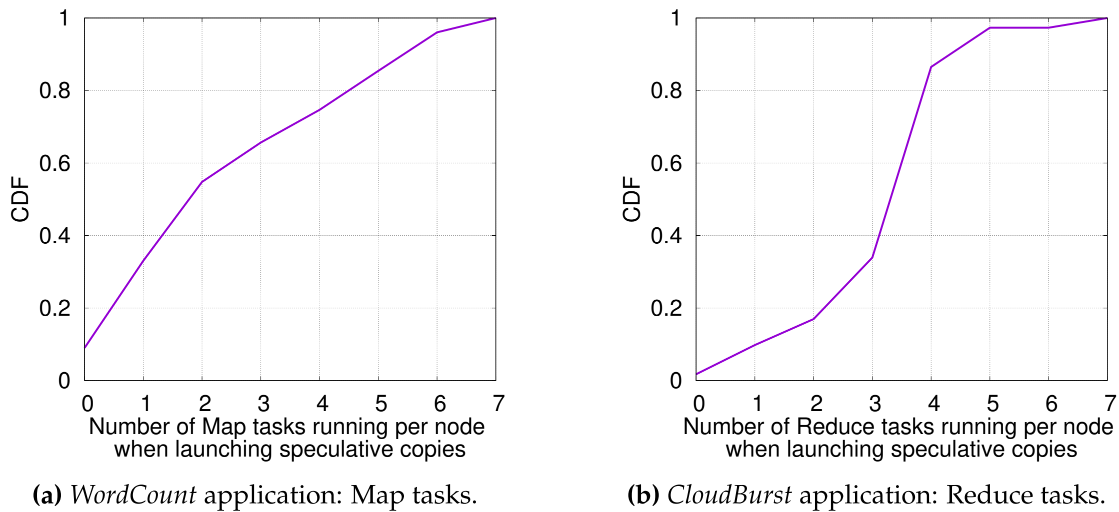
Thus, as Section 3.5.1 suggested, speculative copy allocations can significantly impact the energy impact of speculative execution. We now focus on *CloudBurst* and *WordCount* applications and study their sensitivity to speculative copy allocation.

We plot the CDF of the number of current running tasks on a node when launching speculative copies. Figure 3.14b shows that 62% of speculative copies are launched on nodes each of which hosts at least four running Reduce tasks. As shown in Figure 3.13b, these speculative copies have minor impact on the average power consumption of the nodes on which they run. However, the same observation does not apply to the *WordCount* application, as we can see in Figure 3.14a. Only 24% of speculative copies are launched on nodes each of which hosts at least four running tasks. 76% of speculative copies are launched on nodes with low number of running tasks. These copies result in high additional power consumption to the nodes which host them. This explains the results in Section 3.5.1.

In summary, we conclude that an energy-aware speculative execution mechanism, or straggler handling mechanism in general, is necessary to reduce the energy consumption of Hadoop cluster. The energy-aware approach must consider the impact of launching speculative copies on the overall energy consumption of Hadoop clusters. That is, to find where to schedule speculative copies in order to achieve the best trade-off between performance (*i.e.*, reducing the long-running stragglers) and energy consumption (*i.e.*, minimizing the extra energy consumption).

3.7 Conclusion

With the increasingly large scale of Big Data processing systems, energy consumption has become a major concern in recent years. Similarly, straggler mitigation has become the key feature of Big Data processing systems.



(a) *WordCount* application: Map tasks. **(b)** *CloudBurst* application: Reduce tasks.

Figure 3.14 – The distribution of the current running tasks per node when launching speculative copies.

In this chapter, by means of experimental evaluation, we have shown the impact of straggler mitigation on the energy consumption of Hadoop clusters. We have observed that the default straggler detection mechanism in Hadoop is not accurate and may lead to excessive unnecessary speculative copies. Therefore, it increases the energy consumption of Hadoop clusters. We have quantified the energy cost of speculative execution. We find that the average power consumption in the cluster, when enabling speculative execution, strongly depends on the duration of the speculative tasks (*i.e.*, extra slot occupation), on the idle time, and on the allocation strategy for speculative copies. We conclude that speculative execution may result in a reduction in the energy consumption if and only if the running time of the application is noticeably reduced to compensate the energy cost of speculative execution. Furthermore, we demonstrate that existing speculative execution mechanisms launch speculative copies very late due to resource unavailability. This can strongly degrade the effectiveness of speculative execution. Finally, we discussed the trade-off between performance and energy consumption when scheduling a speculative copy.

It is important to note that the findings that we present in this chapter are not limited to Hadoop and can be applied to different Big Data processing frameworks that are featured with speculative execution to handle stragglers (*e.g.*, Spark [130]).

In the next chapters, we discuss in detail our contributions addressing the aforementioned issues, towards energy-efficient straggler mitigation for Big Data processing systems.

Chapter 4

Measuring and Enabling the Energy Efficiency of Straggler Detection

Contents

4.1 Energy Inefficiency of Existing Straggler Detection Mechanisms 48
4.2 A Framework to Evaluate Straggler Detection Mechanisms 49
4.2.1 Metrics for Characterizing Straggler Detection Mechanisms 50
4.2.2 Linking Straggler Detection Metrics to Performance 53
4.2.3 Characterizing Straggler Detection Mechanisms via the Proposed Metrics 56
4.3 Hierarchical Straggler Detection: A Green Straggler Detection Mechanism 62
4.3.1 Design Principles 63
4.3.2 Architecture 63
4.3.3 Characterizing the Hierarchical Straggler Detection Mechanism . . . 64
4.3.4 Evaluating the Effectiveness of Straggler Detection Mechanisms . . . 65
4.3.5 Evaluating *Hierarchical* with Different Applications and Slow-node Thresholds 70
4.4 Conclusion 73

EXISTING straggler mitigation techniques do not always improve performance and may result in high energy consumption [93]. Therefore, it is imperative to quantitatively characterize existing straggler mitigation techniques in order to estimate their impact on performance and energy consumption. This information is the stepping stone to improve existing straggler mitigation techniques, with respect to performance and energy efficiency.

In this chapter, we introduce a comprehensive framework to characterize and evaluate existing straggler detection mechanisms. We start with a set of metrics that can be used to

characterize straggler detection mechanisms, including *Precision*, *Recall*, *Detection Latency*, *Undetected Time* and *Fake Positive*. Then, we develop an architectural model by which these metrics can be linked to execution time and energy consumption. In addition, we conduct a set of experiments on Grid'5000 and use these metrics to characterize the state-of-the-art straggler detection mechanisms.

Based on the proposed metrics, we introduce an energy-driven straggler detection mechanism. This mechanism, called *Hierarchical*, targets a high *Precision* in order to reduce the wasteful energy consumption on unnecessary speculative copies. As its name suggests, it works as a secondary straggler detection layer on the top of regular straggler detection mechanisms. It considers tasks at the node-level. Only tasks on nodes with relatively low performance, compared to the average cluster performance, are taken into consideration while detecting stragglers. We evaluate our *Hierarchical* straggler detection mechanism through a set of experiments on Grid'5000 with representative Big Data applications.

The rest of this chapter is organized as follows. Firstly, we analyze the traces of Hadoop production cluster to illustrate the inefficiency of existing straggler mitigation techniques. Then, the metrics for characterizing straggler detection mechanisms are presented in detail. Subsequently, we introduce our *Hierarchical* straggler detection mechanism.

4.1 Energy Inefficiency of Existing Straggler Detection Mechanisms

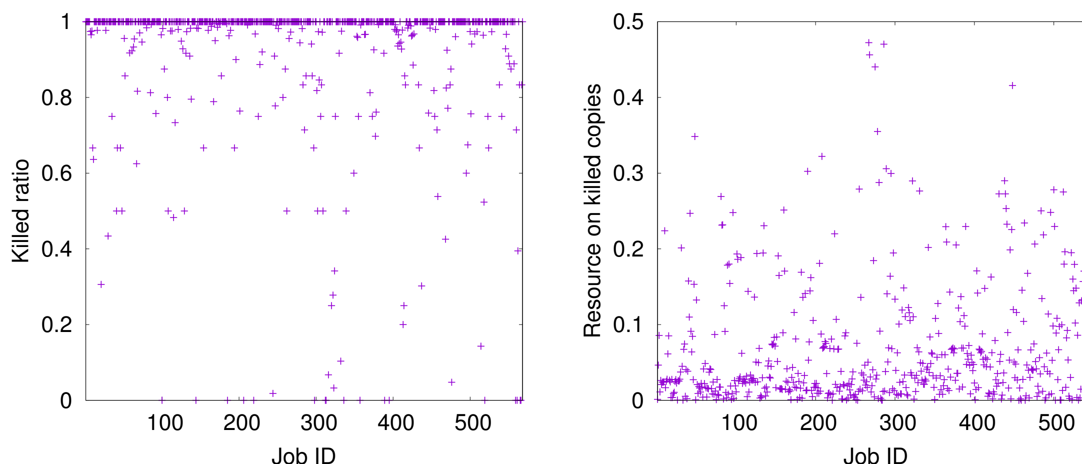
Currently, the common wisdom applied in existing straggler detection mechanisms is to detect as many stragglers as possible in order to cut the long running tail in job execution. For example, *Default* [31] decides a task with progress less than 80% of the average progress as straggler. *LATE* [131] marks the tasks with speed less than the mean speed minus the standard deviation as stragglers. *Mantri* [6] considers tasks with 1.5x times longer execution time than average execution time as stragglers.

$$\text{Killed ratio} = \frac{\text{\#Killed speculative copies}}{\text{\#Total speculative copies}} \tag{4.1}$$

$$\text{Resource on killed copies} = \frac{\text{\#Resource consumption of killed copies}}{\text{\#Total job's resource consumption}} \tag{4.2}$$

We have analyzed the traces from a Hadoop production cluster at CMU collected in Octobar 2012 [93]. Figure 4.1 shows the ratio of killed speculative copies, *i.e.*, unsuccessful copies, over all copies for each Hadoop job, as well as the ratio of resources consumed by the killed copies over the total resource consumption of a job. Equations 4.1 and 4.2 demonstrate how to calculate these metrics. We observe that many speculative copies are unsuccessful. These unsuccessful speculative copies in turn waste a lot of resources, which in other words can be converted into energy waste. For example, among the total 568 jobs, there are 370 jobs which have speculative execution with zero successful copies. For some jobs, the killed speculative copies consume more than 40% of the job's total resource consumption.

There exists several reasons which can cause this high ratio of killed copies. Amongst them, late detection and wrongly detection are two major sources causing this. Identifying the root causes is essential to any further improvement. Unfortunately, there exists no dedicated measuring metric for straggler detection mechanisms which can specifically indicate



(a) Ratio of killed speculative copies. **(b)** Resource consumption ratio on killed copies.

Figure 4.1 – Hadoop production trace analysis: The ratio of killed speculative copies to the total speculative copies launched is high. For more than 65% of the jobs, all speculative copies, which were launched, are killed. These unsuccessful speculative copies consume wastefully resources while running. The ratio of resource consumption on these killed speculative copies can be more than 40% of the total job resource consumption.

the characteristics of these mechanisms. In the next section, we introduce a set of dedicated metrics designed for characterizing straggler detection mechanisms.

4.2 A Framework to Evaluate Straggler Detection Mechanisms

Most of the existing work on straggler detection focuses on the impact of their detection mechanisms by evaluating the reduction in the job's execution time [4, 6, 22, 31, 58, 131] or the reduction in execution time of long-running stragglers [4, 6]. Other studies assess the impact of the straggler detection mechanisms through evaluation metrics such as the total number of successful speculative copies [6, 122, 123], extra resource usage [6, 122]. Those metrics are insufficient, imprecise, and sometimes even result in incorrect interpretations.

To illustrate this observation, assume a scenario where a Big Data job consists of 100 tasks. Amongst these tasks, there exists 10 stragglers. In this scenario, the straggler detection mechanism A detects 50 tasks as stragglers. Amongst these detected stragglers, there are 5 actual stragglers. The other 45 detected stragglers are actually normal tasks, which were overly detected as stragglers. The straggler detection mechanism B detects only 5 stragglers. These 5 detected stragglers are actual stragglers. At the end, both A and B result in the same execution time. If execution time is used as a metric to evaluate these two straggler detection mechanisms, the conclusion is they have similar characteristics. However, this conclusion is clearly inadequate and misleading. Moreover, the conclusion obtained by using this metric may be dramatically changed when these two detection mechanisms are used in different systems.

In this section, our study tackles this issue by introducing a list of evaluation metrics to comprehensively characterize straggler detection mechanisms. The following sections

Table 4.1 – Technical terms and definitions related to speculative execution.

Terms	Definitions
Execution time	The time that a task/job takes from the moment it starts until it finishes.
Slot/container	The resource unit which executes the Map/Reduce tasks.
Wave	This term refers the ratio of job size, represented by the tasks number, to the cluster capacity. If the tasks number of a job equals x times the cluster capacity, we can say that the job's execution consists of x waves.
Detected straggler	The task that is detected by the straggler detection as straggler.
Speculative copy	The replica instance of the task, which is detected as straggler.
Successful speculative copy	The speculative copy that can finish before its original task. This task is marked as successful. The original task is killed upon the finish notification of the copy.
Unsuccessful speculative copy	In contrast, the unsuccessful copy is the copy that cannot finish before its original task. It gets killed upon the completion of its original task.
Speculative lag	This parameter is used to trigger the straggler detection and handling. The straggler detection mechanism starts to look up for stragglers if: (i) there are no waiting unscheduled tasks and (ii) the job has been running for longer than the Speculative Lag. In Hadoop, the default value for this parameter is 60 seconds.

describe these metrics and provide hints to use them in characterizing straggler detection mechanisms.

Technical Terms and Definitions. For the sake of references, we gather all related technical terms as well as their definitions in Table 4.1.

4.2.1 Metrics for Characterizing Straggler Detection Mechanisms

In this section, we discuss in detail the shortcomings of existing metrics in evaluating straggler detection mechanisms. Then we present a list of new metrics to characterize stragglers and evaluate the effectiveness of detection mechanisms.

Table 4.2 – Existing metrics for evaluating straggler detection mechanisms.

Goal	Metric	Description	Related work
Efficiency	Execution time	Measurement of the impact of speculative execution on reducing the execution time	[4, 6, 22, 58, 131]
	Resource consumption	Measurement of the reduction in total resource consumption	[4, 6, 122]
	Heavy-tail reduction	Reduction of the ratio between the longest and average tasks' execution time	[4, 6]
	Wasteful resource occupation	The amount of resource consumed by unsuccessful copies	[131]
Characterizing	Number of speculative copies	The number of speculative copies launched throughout the execution	[6, 117, 131]
	Number of successful copies	Total number of speculative copies successfully finished	[6, 117]

4.2.1.1 Lack of evaluation metrics for straggler detection

Many studies have concerned improving the speculative execution in Big Data processing systems. In order to measure the impact of the proposed solutions, as well as characterize the straggler detection mechanisms, some individual metrics have been proposed. Table 4.2 lists some existing metrics which are used in studies related to straggler mitigation.

While efficiency metrics can measure the impact of a straggler detection mechanism on a given system, the characterizing metrics are insufficient to explain this efficiency and may result in incorrect interpretations. Furthermore, they cannot be used in a mathematical model to predict performance.

As an example, we analyzed a one-month trace of three production Hadoop clusters [93] and studied the correlation between the number of successful speculative copies and the heave-tail execution reduction. Although this *number of successful copies* metric is used to explain the impact in reducing the execution time [117] of speculative execution, Table 4.3 shows that the absolute value of the correlation coefficient only ranges between 0.04-0.14 for the three clusters (while the maximum absolute value of the correlation coefficient is 1.0). This means that there is no strong correlation between this metric and the effectiveness of speculative execution. This demonstrates that it is not sufficient to use these metrics to evaluate the efficiency of straggler detection. Thus, we need to consider a complete framework for straggler detection and mitigation. Such framework can allow system administrators to

Table 4.3 – Traces of three Hadoop production clusters. The table presents the correlation coefficient between the successful speculative copy ratio and the ratio of the longest and the average task's execution time for each job. The results show that the correlation is not significant.

Cluster	Date	#Jobs	#Tasks	#Successful copies	Correlation
M45	04-2010	1735	1759434	6085	0.055
OPENCLOUD	01-2011	989	1310160	26746	0.042
WEB MINING	10-2012	1074	1000427	5136	-0.14

predict the performance of straggler detection mechanisms in different scenarios and thus help them to select the detection mechanism which best fits their needs.

4.2.1.2 Precision, Recall, Detection Latency and Undetected Time

We now discuss new metrics to evaluate the effectiveness of straggler detection mechanisms. First, it is important to understand the status of each task *post-execution*. A task T can be either a straggler ($T \in \texttt{stg}$) or not ($T \in \overline{\texttt{stg}}$), and detected as a straggler ($T \in \texttt{det}$) or not ($T \in \overline{\texttt{det}}$). There exists several definitions of stragglers. For instance, Dean *et al.* [31] defines a straggler as a task which takes at least 20% longer time to complete compared to the average task execution time. When detecting stragglers, the natural idea that comes to mind is what was indeed detected. In particular it is very natural to consider the following parameters:

False Negative : A straggler that occurred but was not detected by the mechanism ($\texttt{stg} \wedge \overline{\texttt{det}}$).

True Positive : A detected straggler ($\texttt{stg} \wedge \texttt{det}$).

False Positive : A task detected that in the end was not a straggler ($\overline{\texttt{stg}} \wedge \texttt{det}$).

Note that *False Negative*, *True Positive* and *False Positive* are parameters that are well-known in the detection community (see for example in failure detection [13, 37]). Based on these, we can define the *Precision* p and *Recall* r of a straggler detection mechanism, that are:

$$p = \frac{|\texttt{stg} \wedge \texttt{det}|}{|\texttt{det}|} = \frac{|\text{True positive}|}{|\text{True positive} + \text{False positive}|} \tag{4.3}$$

$$r = \frac{|\texttt{stg} \wedge \texttt{det}|}{|\texttt{stg}|} = \frac{|\text{True positive}|}{|\text{True positive} + \text{False negative}|}. \tag{4.4}$$

Simply put, the *Precision* is the number of correct straggler detected amongst all detected tasks, and the *Recall* is the ratio of detected stragglers amongst all stragglers that are present in the system.

However, while these parameters are sufficient in the failure detection community, they do not suffice for straggler detection. While a failure is a very punctual event that you either detect or not, a straggler is a long-lasting event. It seems important to reward detectors that could detect stragglers very early. Intuitively, there is major a difference between a

straggler detection mechanism that detects a straggler after 60 seconds of execution and one that detects it after 10 minutes.

To measure this, we introduce a new parameter, namely the *Detection Latency*. For a detected straggler, *Detection Latency* expresses how fast the straggler is detected compared to its normal execution time, that is:

$$\text{Detection Latency} = \frac{\sum_{\texttt{stg} \wedge \texttt{det}} \frac{\text{Time to detection}}{\text{Normal execution time}}}{|\texttt{stg} \wedge \texttt{det}|} \quad (4.5)$$

In existing studies on straggler detection, the average task execution time is widely considered as normal execution time [6, 31]. However, recent studies have demonstrated that this approach results in misleading information [52, 54]. For instance, the execution times of different tasks might not be identical [52]. In this case, considering average task execution time as normal execution time of an arbitrary task may be inaccurate. In Section 4.2.3.2, we discuss in detail this issue and introduce a more accurate method to estimate the normal execution time of each task.

In addition, similarly there is a difference for a non-detected straggler between one that finishes in twice its normal execution time, and one that finishes after ten times its normal execution time. *Intuitively, this gives an idea of the cost of a non-detected straggler.* To measure this, we introduce a new parameter, namely the *Undetected Time*. For a non-detected straggler, *Undetected Time* expresses how long does it take to execute compare to its normal execution time.

$$\text{Undetected Time} = \frac{\sum_{\texttt{stg} \wedge \overline{\texttt{det}}} \frac{\text{Execution Time}}{\text{Normal execution time}}}{|\texttt{stg} \wedge \overline{\texttt{det}}|} \quad (4.6)$$

For *Detection Latency* and *Undetected Time*, the smaller the better, while for *Precision* and *Recall*, the higher the better.

4.2.2 Linking Straggler Detection Metrics to Performance

In this section, we propose a model to predict the energy consumption and slowdown of a node. Then, we use this model to illustrate the relationship of our metrics with execution time and energy consumption.

4.2.2.1 Architectural Models for Performance and Energy Consumption

We start by providing a very simple model for a multi-threaded, multi-core machine. We argue that this model, although simple, is realistic enough to allow for improvements in the manipulation of stragglers.

Assume that we have a multi-core node with c cores that support t tasks each. The maximum number of tasks for the node is then $c \times t$. In the following, we assume scattered-thread strategies (hence the number of tasks between any two cores on each node differ at most by one).

Power Consumption. A multi-core node has different states in which the power-consumption may differ: it can be turned off hence not consuming any power, or turned on and having a static power consumption ($\mathcal{P}_{\text{static}}$). Then depending on the number of cores that are active, a dynamic power ($\mathcal{P}_{\text{dyn}}$) is added that we assume proportional to the number of active cores. With n being the number of tasks running concurrently, the power $\mathcal{P}$ is:

$$\mathcal{P} = \begin{cases} 0 & \text{for } n = 0 \text{ (turned off)} \\ \mathcal{P}_{\text{static}} + n \times \mathcal{P}_{\text{dyn}} & \text{for } 1 \leq n \leq c \\ \mathcal{P}_{\text{static}} + c \times \mathcal{P}_{\text{dyn}} & \text{for } c < n \leq ct \end{cases}$$

Based on this, we can divide the nodes into three categories depending on the number of tasks running concurrently:

- $\mathcal{A}$: The nodes that are turned off, adding a task on them would increase the current power consumption by $\mathcal{P}_{\text{static}} + \mathcal{P}_{\text{dyn}}$.
- $\mathcal{B}$: The nodes that have between 1 and $c - 1$ tasks running on them, adding a task on them would increase the current power consumption by $\mathcal{P}_{\text{dyn}}$.
- $\mathcal{C}$: The nodes that have between c and $c \times t - 1$ tasks running on them, adding a task on them would not increase the power consumption as all the cores are running already.

Average Slowdown Factor and Interference Model. Hereafter, we model the slowdown to a task caused by resource contention between concurrent tasks running on the same node. We denote the average slowdown factor as α. We observe that α equals to one when the number of concurrent tasks is less than the number of cores c. This is because each task can be executed on a dedicated core and there is hardly any contention between the tasks. When the number of tasks increases beyond c, the contention also increases. Let n be the number of running tasks. We define the average slowdown factor α as below.

$$\alpha = \begin{cases} 1 & \text{for } 1 \leq n \leq c \\ \frac{n}{c} & \text{for } c < n \leq ct \end{cases} \tag{4.7}$$

4.2.2.2 On the Impact of Precision and Recall on Energy Consumption and Execution Time

In this section we give a mathematical intuition to help understanding the impact of different characteristics of a detection mechanism on the performance of the system.

Let us consider an application of n tasks, on an architecture with $2n$ nodes each with a single core. We assume that amongst those, we have a ratio ϕ of stragglers. We use a very simple algorithm to deal with stragglers:

(i) It schedules speculative copies of all detected stragglers on new nodes as soon as they are detected.

(ii) For a handled stragglers, when either the straggler or the speculative copy finishes, the other one get killed.

We now do the following assumption as first-order approximations (FOA):

- For a detected straggler, we assume that the speculative copy finishes before the straggler (otherwise the detection is useless and one could count it as non-detected, note that we discuss this hypothesis in Section 4.2.3.2 with the introduction of *Fake Positive* metric).
- We assume that the detection of a non-straggler occurs on average at 50% its execution. In Section 4.2.3, we observe that the average *Detection Latency* of existing straggler detection mechanisms is 45% of normal execution time.

We can evaluate the performance and energy of a task depending on different parameters, namely whether it is a straggler ($\mathtt{stg}$) or not ($\overline{\mathtt{stg}}$), and whether it was detected as a straggler ($\mathtt{det}$) or not ($\overline{\mathtt{det}}$). Then we can determine FOA for both the time overhead and energy consumption of tasks based on these, namely:

Detected Stragglers. They occur with probability:

$$\mathbb{P}(\mathtt{stg} \wedge \mathtt{det}) = \mathbb{P}(\mathtt{stg})\mathbb{P}(\mathtt{det}|\mathtt{stg}) = \mathtt{Recall} \times \phi \tag{4.8}$$

$$\begin{aligned} \text{Time Overhead} &\approx 1 + \text{Detection Latency} \\ \text{Energy cost} &\approx (2 + \text{Detection Latency}) \times (\mathcal{P}_{\text{static}} + \mathcal{P}_{\text{dyn}}) \end{aligned}$$

The time overhead is the time it takes to detect the straggler (*Detection Latency*) and the time for a normal task execution (normalized by 1). A high *Detection Latency* results in late speculative execution. Consequently, it leads to longer execution time. With respect to energy consumption, as both the straggler and the speculative copies are running, it comes with the energy cost. Moreover, high *Detection Latency* results in extra energy cost, as stragglers consume more energy during the time detection mechanism takes to detect them.

Non-detected Stragglers. They occur with probability:

$$\mathbb{P}(\mathtt{stg} \wedge \overline{\mathtt{det}}) = \mathbb{P}(\mathtt{stg})\mathbb{P}(\overline{\mathtt{det}}|\mathtt{stg}) = (1 - \mathtt{Recall}) \times \phi \tag{4.9}$$

$$\begin{aligned} \text{Time Overhead} &\approx \text{Undetected Time} \\ \text{Energy cost} &\approx \text{Undetected Time} \times (\mathcal{P}_{\text{static}} + \mathcal{P}_{\text{dyn}}) \end{aligned}$$

The time overhead is the time it takes for the non-detected straggler to execute (*Undetected Time*). A long-running non-detected straggler can severely prolong the whole job's execution. A better straggler detection mechanism should have smaller *Undetected Time*. The same holds for the energy. High value of *Undetected Time* results in long execution time during which non-detected stragglers consume energy.

Detected Non-stragglers. They occur with probability:

$$\mathbb{P}(\overline{\texttt{stg}} \wedge \texttt{det}) = \frac{False_P}{n} = \texttt{Recall} \times \phi \times \frac{1 - \texttt{Precision}}{\texttt{Precision}} \tag{4.10}$$

$$\text{Time Overhead} \approx 1$$
$$\text{Energy cost} \approx 1.5 \times (\mathcal{P}_{\text{static}} + \mathcal{P}_{\text{dyn}})$$

Our first order approximation states that we detect (and handle) the *False Positive* at half its execution on average, hence during half it's execution the energy cost is doubled. A higher *Precision* reduces this energy cost.

Non-detected Non-stragglers. They occur with probability:

$$\begin{aligned} \mathbb{P}(\overline{\texttt{stg}} \wedge \overline{\texttt{det}}) &= 1 - \mathbb{P}(\texttt{stg} \wedge \overline{\texttt{det}}) - \mathbb{P}(\texttt{stg} \wedge \texttt{det}) - \mathbb{P}(\overline{\texttt{stg}} \wedge \texttt{det}) \\ &= 1 - \phi \times (1 + \frac{\texttt{Recall} \times (1 - \texttt{Precision})}{\texttt{Precision}}) \end{aligned} \tag{4.11}$$

$$\text{Time Overhead} \approx 1$$
$$\text{Energy cost} \approx (\mathcal{P}_{\text{static}} + \mathcal{P}_{\text{dyn}})$$

Simply put, one can see it as:

- *Recall* impacts performance. A higher *Recall* results in higher performance improvement as more stragglers are detected.
- *False Positive* (and with this, *Precision*) impacts energy efficiency. A higher *Precision* results in lower number of detected non-stragglers (*i.e.*, wrongly detected stragglers). This in turn reduces the wasteful energy consumed by unnecessary speculative copies of these detected non-stragglers.

Again, we want to stress out that this is a very naïve model. For instance *False Positive* can also impact performance, in the case when there are not enough nodes to duplicate all detected stragglers. In this case, *False Positive* will delay the speculative copies of real stragglers. Nonetheless, this intuition still provides useful information for linking the proposed metrics to performance and energy consumption. In Section 4.2.3, we use this intuition to interpret the results of existing straggler detection mechanisms when using our evaluation metrics.

4.2.3 Characterizing Straggler Detection Mechanisms via the Proposed Metrics

In this section, a set of experiments will be conducted in order to illustrate the usage of our proposed metrics, *e.g.*, *Precision*, *Recall*, on characterizing the straggler detection mechanisms. We start with the detailed description of the experimental setups used throughout our experiments. Then, the evaluation metrics are used to characterize existing straggler detection mechanisms. Finally, we analyze the obtained characteristics of each straggler detection mechanism in order to provide in-depth straggler detection mechanism evaluation and characterization.

4.2.3.1 Experiment Setup

Testbed. All the experiments conducted throughout the evaluation were executed on Grid'5000 testbed [59], *i.e.*, a scientific testbed that enables the run of experiments at large scales with highly-configurable infrastructure located across 10 sites in France. For our experiments, we used a 21-node cluster on Nancy site. Each node is equipped with 4-core CPU Intel Xeon X3440, 16 GB of memory, 300 GB of storage and 1 Gb/s Ethernet network for intra-cluster communication. Moreover, each node on this cluster is attached with power monitoring hardware, *i.e.*, Power Distribution Units (PDUs)). They allow us to acquire fine-grained power consumption information on each node individually during the experiments.

Platform. We deployed Linux Ubuntu 16.04 LTS on our cluster. Moreover, Hadoop 2.7.3 [108], *i.e.*, the stable version released in August 2016, was used for running our experiments. We configured Hadoop with one dedicated node as the master node. The rest 20 nodes were worker nodes. Each worker node was set to have maximum 8 *Map* slots and 8 *Reduce* slots. Regarding the file system configuration, we kept the default setting, where the replication factor was 3 and the chunk size was 64 MB.

Application. As precision is part of the evaluation, we wanted to fully control the behavior of the application and to avoid the unexpected deviation which can impact the results. Therefore, we chose *WordCount*, a simple yet representative MapReduce application. Moreover, the experiments with only this application took more than 100 hours to finish.

The basic characteristics of the *WordCount* job are presented in Table 4.4. The cluster capacity is 160 Map slots. This job, which consists of 320 Map tasks, runs two waves of Map tasks. We decide this setting based on the real-life Hadoop production cluster traces [93]. As we observed, there were roughly 25% of the jobs which are 2-wave+. More importantly, these 25% 2-wave+ jobs contribute roughly 90% to the total launched Map tasks. Figure 4.2 illustrates the high ratio of the multiple-wave jobs in a real-life production Hadoop cluster. Therefore, our setting can accurately reflect the jobs' execution in real-life MapReduce systems.

Straggler Injection. We propose a scheme to proactively control the straggler ratio injected throughout the experiments. We divide the 20 workers into four groups $G_i, i = 1..4$. Each group G_i consists of p_i percent of the total 20 nodes. A node belonging to group G_i has i active cores out of four. Each value of this four-dimension vector makes a specific scenario $\mathcal{C}_j = \langle p_1, p_2, p_3, p_4 \rangle$. Throughout our experiments, we vary this straggler injection ratio to present different scenarios covering a broad range of straggler occurrence, which are: $\mathcal{C}_1 = \langle 35, 35, 5, 25 \rangle$, $\mathcal{C}_2 = \langle 25, 25, 25, 25 \rangle$, $\mathcal{C}_3 = \langle 10, 10, 5, 75 \rangle$ and $\mathcal{C}_4 = \langle 5, 5, 0, 90 \rangle$. Simply put, in configuration $\mathcal{C}_4$, 5% of the nodes have only one active core, 5% have two active cores, and 90% have four active cores. Hence for an identical load on all nodes, 10% of the tasks should be stragglers.

Straggler Detection Mechanisms. Throughout our experiments, we examined two straggler detection mechanisms, two from the literature: *Default* [31] and *LATE* [131] mechanisms. These two mechanisms are available in existing Hadoop [113] and YARN [108] implementations.

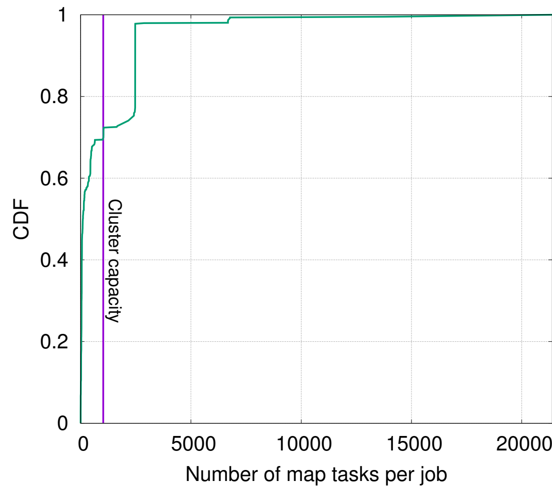


Figure 4.2 – The distribution of Map task number in the production Hadoop cluster for the jobs submitted in October 2012: This cluster is equipped with 64 machines, 8-core CPU and 16GB of memory each. By default, the cluster capacity is 1024 Map slots. We noticed that there were roughly 30% of the jobs having more than 1024 Map tasks. In other words, 30% of the jobs run multiple Map waves. Moreover, 25% of the jobs run more than 2 waves. It is important to emphasize that these 25% 2-wave+ jobs contribute almost 90% to the total launched Map tasks.

Table 4.4 – Application characteristics and configurations.

Feature	Value
Dominant phase	Map
Dominant resources	CPU
Input size	20 GB
Shuffle size	400 MB
Output size	100 MB
No. Map tasks	320
No. Reduce tasks	160

4.2.3.2 Evaluation of Straggler Detection Mechanisms

In this section, a set of experiments is conducted in order to illustrate the usage of our proposed metrics on characterizing straggler detection mechanisms. First of all, we present the method for classifying the stragglers. Subsequently, we characterize the straggler detection mechanisms *Default* [31] and *LATE* [131] using our metrics.

Detecting a Straggler. The key problem here is to detect after each run which tasks were stragglers and which were not in order to determine the characteristics of the straggler detection mechanisms. According to the straggler definition [31], a straggler is a task that takes more than 1.2 times *the time it normally takes to be executed*.

We propose a method to determine the *normal execution time* of a task. Intuitively, homogeneous environment is an environment (i) with very few stragglers, and (ii) where the stragglers can be manually detected. By running enough executions of an application in

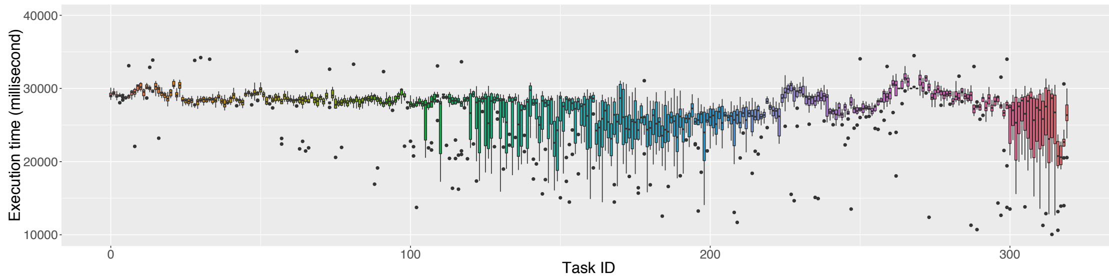


Figure 4.3 – Distribution of execution times (in milliseconds) per task ID for the *WordCount* application on a homogeneous platform.

a homogeneous environment, one can compute the normal execution time of each task by taking the X quantile (*i.e.*, a proportion X of the execution times were below this value and $1 - X$ above), where X can be defined according to those results. By comparing the task's execution time to the *normal execution time*, this method targets a high straggler detection accuracy and eliminates the wrong stragglers detected caused by the non-local task execution or the task execution skewness [6, 22].

Based on this, we ran 10 times the *WordCount* application with identical input setups and measured the execution time of each task individually. We plot these results in Figure 4.3.

In this case, we could then evaluate that the *normal execution time* is the 0.5 quantile time (meaning that 50% of the execution times are below this value and 50% above this value) amongst the different execution times. We determined it based on the hypothesis that in the homogeneous case, the stragglers correspond to the outliers observed. For instance, in Figure 4.3, there are 21 outliers. We give in Table 4.5 the number of supposed stragglers depending on the quantile chosen.

Limitations. We want to point out that we do not believe that there is a perfect way to determine the *normal execution time*. In particular, we expect that the threshold for stragglers will differ according to the application studied. However one does not need the exact value of execution time to detect a straggler. Let us consider the following values:

- T_{task} the *normal execution time* (unknown).
- $s_{\min}$ the minimum slowdown of stragglers.
- $\varepsilon_{\text{setting}}$ the possible variation in time due to settings (depends on the application and the machine).

Therefore, one can compute a sufficient condition for $\varepsilon_{\text{guess}}$ the error authorized in the normal execution time guessed:

$$1.2T_{\text{task}}(1 + \varepsilon_{\text{guess}}) \leq s_{\min} T_{\text{task}}(1 - \varepsilon_{\text{setting}}) \tag{4.12}$$

$$T_{\text{task}}(1 + \varepsilon_{\text{setting}}) \leq 1.2T_{\text{task}}(1 - \varepsilon_{\text{guess}}) \tag{4.13}$$

Table 4.5 – Straggler ratio on an homogeneous platform for the *WordCount* application, based on the definition of the normal execution time: the normal execution time is the time for the X *quantile* of execution time.

Quantiles of execution time	0.25	0.5	0.6	0.7	0.9
Ratio of stragglers	8.16%	0.66%	0.31%	0.16%	0.06%

Equation 4.12 ensures that a straggler is detected as a straggler and Equation 4.13 ensures that a non-straggler will not be detected as a straggler. Such a value only exists if:

$$\varepsilon_{\text{setting}} \leq \frac{s_{\min} - 1}{s_{\min} + 1} \tag{4.14}$$

which means that the variation due to settings is not too big. In fine, this says that $\varepsilon_{\text{guess}}$ should satisfy the following condition

$$\varepsilon_{\text{guess}} < \min\left(\frac{s_{\min}(1 - \varepsilon_{\text{guess}})}{1.2} - 1, \frac{0.2 - \varepsilon_{\text{guess}}}{1.2}\right) \tag{4.15}$$

This provides a sufficient condition for the imperfection in determining the execution time. It should be noted that in our case, the threshold that we chose (0.5 quantile) allowed to closely determine the expected number of stragglers in homogeneous environment (see Table 4.5) as well as in all heterogeneous configuration scenario $\mathcal{C}_{i=1..4}$.

Impact of the Speculative Lag on the Straggler Detection Effectiveness. For detecting the stragglers, it is important to decide at which moment of the execution to trigger the detection process. For instance, making the decision at the very early stage of the execution is not efficient as there is not much information about the execution at that time. On the contrary, triggering the detection at the very end of the task executions can strongly reduce the chance of the speculative copy to successfully finish.

In Hadoop, there is a parameter called *Speculative lag* which specifies the waiting time after the job triggers the straggler detection. This value is set to 60 seconds by default [113]. However, this static parameter does not work well with different applications having different execution characteristics. Therefore, it is possible for a real straggler to be detected too late, so that its speculative copy has no chance to finish before the straggler. We call this *Fake Positive* detection. In the scope of this work, we define a straggler is *Fake Positive* detected, if it is detected at the moment such that its remaining time is smaller than its *normal execution time*.

$$\text{Fake positive} = \frac{|\text{Fake positive}|}{|\text{True positive} + \text{False positive}|} \tag{4.16}$$

Thus, the presence of this metric reduces the value of the *Precision p*.

$$\text{Precision} = \frac{|\text{True positive} - \text{Fake positive}|}{|\text{True positive} + \text{False positive}|} \tag{4.17}$$

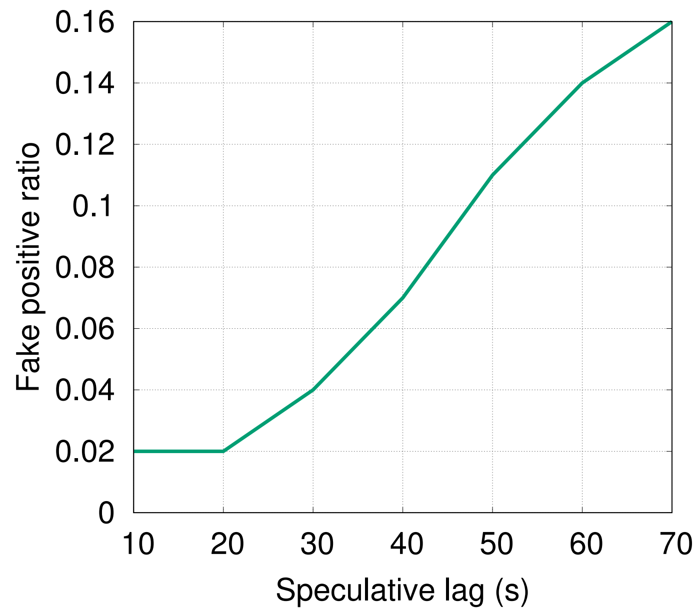


Figure 4.4 – Impact of *Speculative lag* on the *Fake Positive* ratio with *Default* straggler detection mechanism while running *WordCount* application.

In order to understand the impact of the *Speculative lag* parameter on causing the *Fake Positive* detection, we conduct a set of experiments with *WordCount* application while monitoring the *Fake Positive* ratio (calculated by the Equation 4.16). We run the *Default* straggler detection mechanism with various values of *Speculative lag*. Figure 4.4 illustrates that the default value of 60 seconds of the *Speculative lag* parameter can result in a high *Fake Positive* ratio (0.14 in this case). As Map tasks normally take 30 seconds to finish. Triggering the detection only after 60 seconds of running is too late.

The presence of this parameter with a high value reduces the *Precision* of the detection mechanism. For instance, if the *Precision* is 0.5 (without considering the *Fake Positive*), and the *Fake positive* is 0.14. The real *Precision* accordingly is reduced down to 0.36 (re-calculated using Equation 4.17). This equals to a reduction of 28%.

In contrast, a 20- value of *Speculative lag* keeps the *Fake positive* ratio stable at a small value. *For the rest of our experiments, we set the Speculative lag to 20 seconds*. This *Speculative lag* configuration, which is calibrated according to the *WordCount* application, can assure a negligible impact of the Fake Positive ratio on our experimental results. For other applications with different execution characteristics, the relevant *Speculative lag* values should be adjusted accordingly.

Characterization of the Straggler Detection Mechanisms. On Table 4.6, we present the characteristics of the two straggler detection mechanisms — *Default - D* and *LATE - L*. The metrics that we use to characterize the mechanisms are: *Precision*, *Recall*, *Detection Latency - DL*, *Undetected Time - UT* and *Fake Positive - FP*. We study each of the two mechanisms in four different scenarios.

The *Default* straggler detection mechanism has quite low *Precision*. In the $\mathcal{C}_4$ scenario, *Precision* is only 0.12, which means 88% of detected tasks were not actual stragglers. The

Table 4.6 – The characteristics of two state-of-the-art straggler detection mechanisms when running in four different scenarios. The evaluation metrics are: *Precision*, *Recall*, *Detection Latency - DL*, *Fake Positive - FP* and *Undetected Time - UT*.

$\langle p_1, p_2, p_3, p_4 \rangle$	Mechanism	Precision	Recall	DL	FP	UT
$\langle 35, 35, 5, 25 \rangle$	D	0.64	0.48	0.55	0.05	2.10
	L	0.83	0.61	0.44	0.04	1.80
$\langle 25, 25, 25, 25 \rangle$	D	0.64	0.46	0.41	0.02	1.80
	L	0.82	0.60	0.38	0.04	1.60
$\langle 10, 10, 5, 75 \rangle$	D	0.22	0.41	0.49	0.07	1.90
	L	0.24	0.55	0.48	0.06	1.80
$\langle 5, 5, 0, 90 \rangle$	D	0.12	0.50	0.51	0.01	2.10
	L	0.31	0.62	0.48	0.03	2.00

Recall metric has moderate values in most of the cases. However, there exists fairly low values in some cases (0.41 in the $\mathcal{C}_3$ scenario). This observation suggests that there exists some potential to significantly improve the *Default* mechanism, in both *Precision* and *Recall*.

The *LATE* straggler detection mechanism is based on progress rate to detect stragglers. This detection mechanism results in a better *Precision* as well as *Recall* compared to *Default*. However, in the $\mathcal{C}_4$ scenario, we notice a *Precision* of 0.31. Besides, the *Recall* values of *LATE* are only a little higher compared to the *Recall* values of *Default*. This again implies that *LATE* mechanism also can still be improved targeting higher *Precision* or *Recall*.

In addition, we also discuss the *Detection Latency* and the *Undetected Time* metrics. Regarding the *Detection Latency*, we notice that it usually takes from 30% to 60% of the normal execution time for the two mechanisms to detect stragglers. These values of *Detection Latency* imply that speculative copies only have chance to successfully finish if the straggler takes at least 130%–160% of the normal execution time to finish. This provides useful information for improving straggler handling mechanisms.

The *Undetected Time* metric specifies the impact of non-detected stragglers in causing the long-running tasks to the job. As we can observe, the difference in the *Undetected time* values of the two straggler detectors is negligible. Which means that the non-detected stragglers have similar execution time. As a result, the overhead in execution time and energy consumption of non-detected stragglers is similar for both *Default* and *LATE*.

In brief, our metrics can easily indicate the characteristics of stragglers detection mechanisms. More importantly, the results show that there is still room for further improving existing straggler detection mechanisms.

4.3 Hierarchical Straggler Detection: A Green Straggler Detection Mechanism

The experimental results in previous section illustrate that the existing straggler detection mechanisms have low *Precision*. In this section, we introduce a novel straggler detection mechanism which adopts the hierarchical approach to detect stragglers. This mechanism aims at a higher *Precision* in order to reduce the wasteful energy consumed by unnecessary

speculative copies. We then conduct a set of experiments to illustrate and evaluate the impact of this new straggler detection mechanism on both performance and energy consumption.

4.3.1 Design Principles

In response to the drawbacks of existing straggler detection mechanisms, we introduce in this section a novel straggler detection mechanism adopting the hierarchical approach, called *Hierarchical*. The main goal of this *Hierarchical* straggler detection mechanism is to reduce the energy consumption while guaranteeing comparable performance. The design objectives are as follows.

(i) The *Hierarchical* straggler detection mechanism is designed to improve the *Precision* (reducing the number of wrongly detected stragglers). The increment in *Precision* is expected to improve the energy efficiency, as the number of wasteful speculative copies for wrongly detected stragglers is reduced.

(ii) The *Hierarchical* aims to keep a low *Undetected Time* by focusing on the stragglers which are expected to have longest execution times. This improves performance as the longest stragglers are the most harmful stragglers to the job performance.

In parallel, it is expected that this comes at the following costs:

(i) The *Hierarchical* straggler detection mechanism is designed to reduce the number of wrong detection. However, it may also reduce the number of *True Positives*. As a result, the *Hierarchical* is expected to have lower *Recall*.

(ii) The *Hierarchical* straggler detection mechanism computes extra information in order to detect a stragglers. This may increase the time to detect a straggler. This can result in a higher *Detection Latency*. However, this additional latency is expected to be in the order of milliseconds.

4.3.2 Architecture

In this section, we present the architecture of the *Hierarchical* straggler detection mechanism. The *Hierarchical* mechanism works as a secondary straggler detection layer on top of an existing straggler detection mechanism. The goal of this detection layer is to select the *critical stragglers*, *i.e.*, the long-running stragglers which strongly affect the job execution time, from the list of stragglers detected by an existing detection mechanism.

The secondary detection layer considers the performance of nodes, on which tasks are running, to detect stragglers. That means, it detects only the stragglers on very slow nodes. The reason for this strategy is that most stragglers are caused by node-level problems, such as a node with worn-out hardware and node-level resource contentions which lead to slow tasks [6]. We identify all the nodes with performance less than β times of the average node performance as *slow nodes*. This parameter is called *slow-node threshold*. In the evaluation, we discuss the impact of this parameter on the speculative execution results.

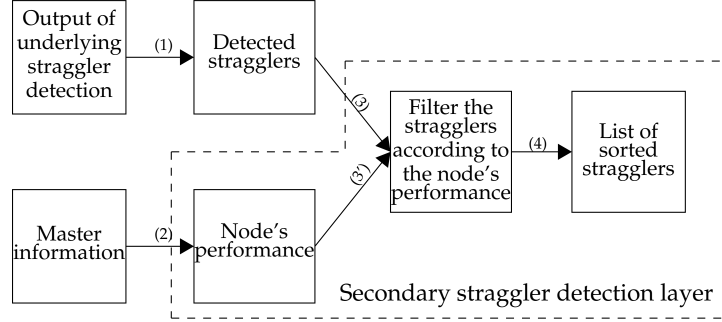


Figure 4.5 – *Hierarchical* straggler detection architecture.

Figure 4.5 shows the design of the secondary straggler detection layer. Specifically, it takes the stragglers detected by the underlying straggler detection layer as input. Then, it calculates the performance of each node and filters out the stragglers that are not hosted on slow nodes. We calculate the performance of a node using the following equation.

$$\text{Perf}_{\text{node}} = \frac{1}{n} \times \alpha \times \sum_{i=1}^{n} \text{Perf}^{i}_{\text{task}} \tag{4.18}$$

where α is the slowdown factor and

$$\text{Perf}^{i}_{\text{task}} = \frac{\text{progress} \times \text{input}}{\text{duration}} \tag{4.19}$$

Equation 4.19 evaluates the performance for a specific task, where *progress* represents the ratio of finished work over the task's total work, *input* is the size of the task's input data in bytes and *duration* is the time from the starting moment of the task. This information of each task is extracted from the Master node's database. Equation 4.18 means that the performance of a host is defined as the sum of the performances of all tasks running on the host.

After filtering, the final list of stragglers are sorted according to their respective performance and the most critical straggler (with the worst performance) is placed in the beginning of the list. We filter and sort the stragglers according to Equations 4.18 and 4.19 to optimize the energy efficiency of speculative execution. Actually, a long-running straggler executing on a slow node is expected to be more critical. Such stragglers are the main reason of causing heavy tails in job executions and as a result wasting energy. Thus, handling those critical stragglers first can potentially lead to better energy efficiency.

It is important to note that our secondary straggler detection layer is independent from the underlying detection layer, and therefore it can be easily integrated with any existing straggler detection mechanisms.

4.3.3 Characterizing the Hierarchical Straggler Detection Mechanism

We implemented *Hierarchical* in two stable versions of Hadoop (1.2.1 and 2.7.3 versions). These implementations consist of roughly 2000 lines of Java codes. Our straggler detection mechanism is implemented as extra module to allow users to easily enable or disable *Hierarchical* using the Hadoop configuration file. It can be used on the top of existing straggler detection mechanisms in Hadoop, *e.g.*, *Default* or *LATE*. The results that we present in the

Table 4.7 – The characteristics of *Hierarchical* when running in four different scenarios. In comparison with *Default* and *LATE*, *Hierarchical* has higher *Precision* and lower *Recall* (see Table 4.6).

$\langle p_1, p_2, p_3, p_4 \rangle$	Precision	Recall	DL	FP	UT
$\langle 35, 35, 5, 25 \rangle$	1.00	0.47	0.65	0.00	2.10
$\langle 25, 25, 25, 25 \rangle$	1.00	0.33	0.57	0.01	1.80
$\langle 10, 10, 5, 75 \rangle$	0.88	0.37	0.55	0.04	1.90
$\langle 5, 5, 0, 90 \rangle$	0.98	0.38	0.53	0.02	2.10

rest of this chapter are obtained when running with *Hierarchical* on the top of the *Default* straggler detection mechanism in the 2.7.3 version.

In this section, we use our proposed metrics to characterize the *Hierarchical* straggler detection mechanism. To do so, we conduct a set of experiments on Grid'5000 with *Hierarchical*. Then, we compare the obtained results to the characteristics of *Default* and *LATE*, as shown in Table 4.6. For this set of experiments, we use the same infrastructure, platform and application setting as mentioned in Section 4.3.3.

We firstly notice that *Hierarchical* has very high *Precision*, up to 1.0 in most of the cases. These values are very high compared to *Default* and *LATE*. On the other hand, it has fairly low *Recall* values. This indeed reflects the design goal of *Hierarchical* mechanism, which mainly focuses on improving the accuracy and efficiency while accepting a lower *Recall* as the cost.

Regarding the *Detection Latency*, we notice that the values of *Detection Latency* of *Hierarchical* are similar to *LATE* and *Default*. This shows that the overhead caused by extra calculation in *Hierarchical* is negligible. Regarding the *Undetected Time* metric, the *Hierarchical* mechanism has similar *Undetected Time* compared to *Default*. This means the non-detected stragglers when using *Hierarchical* have similar execution time in average, compared to when *Default* is used. Disregarding the low *Recall* of *Hierarchical*, this result suggests that *Hierarchical* can have comparable performance with *Default*. We will examine this in the next evaluation.

4.3.4 Evaluating the Effectiveness of Straggler Detection Mechanisms

In this section, we conduct a set of experiments to evaluate the effectiveness of two state-of-the-art straggler detection mechanisms *Default* and *LATE*, as well as our *Hierarchical*. By analyzing the results, we demonstrate that existing evaluation metrics can result in misleading information. Finally, we illustrate how the characteristics, which are obtained using our metrics, can be used to accurately indicate the performance and energy consumption of straggler detection mechanisms.

4.3.4.1 Methodology

In this evaluation, speculative copies can be early launched. Hereafter, we discuss the importance of early speculative copy launching for our evaluation. Then, we present the methodology of providing early available resource.

Early Launching Speculative Copies. By default, Hadoop triggers the speculative execution at the end of the execution when there are available resource. However, as the stragglers can occur at any moment of the execution, this policy can lead to the case when some early stragglers cannot be detected or handled [6, 122, 123]. For instance, a detected straggler might not be handled because all resources are occupied. The results obtained in this case cannot precisely reflect the effectiveness of straggler detection mechanisms. By triggering the straggler detection at early stage of the execution, we expect to provide the comprehensive information about the effectiveness of straggler detection mechanisms in various scenarios.

Cluster Configuration for Resource Availability. In order to provide early available resource, we allocate a fraction x of the total resources for launching regular tasks. The speculative copies run on $100 - x$ of reserved resource. This $100 - x$ percent of cluster capacity is evenly reserved across the nodes. Which implies that each node will reserved $100 - x$ percent of its total capacity.

Setup. We vary the reservation resource percentage x from 50 to 100 ($x = \{100; 95; 90; 75; 50\}$). Our goal is to cover as much as possible the diversity of the scenarios when having different resource quotas for early launching speculative copies. At $x = 100$, there are no reserved resource for early launching speculative copy. Thus, the copies can only start at the end of the execution. For the other values, speculative copies are launched on the $100 - x$ percent reserved resource as soon as there are stragglers detected. Moreover, a sharing resource reservation policy, called *Shared*, is also presented in the evaluation. With this policy, regular tasks and speculative copies are sharing the free resource from the very beginning of the execution. Last but not least, when there are no waiting regular tasks, all free slots are used for launching speculative copies. This is applied for all of the aforementioned policies.

Hereafter, we present the results when running with 6 different speculative reservation policies, including *Shared* and x, where $x = \{100; 95; 90; 75; 50\}$. We compare the behaviors of our cluster when running three different straggler detection mechanisms, *i.e.*, *Default*, *Hierarchical* and *LATE*. In this evaluation, we present the results when running in $\mathcal{C}_2 = (25, 25, 25, 25)$ scenario.

4.3.4.2 Impact of Straggler Detection Mechanisms with Different Resource Reservation Policies

First of all, we present the results in Figures 4.6, 4.7 and 4.8. The values in Figures 4.6 and 4.7 are normalized to the result when using *Default* mechanism in the *100* reservation scenario.

Regarding execution time, having available resource for launching early speculative copies can result in a considerable reduction in execution time. This reduction can be up to 10% in the case of the *Shared* resource reservation policy with the *Hierarchical* straggler detection mechanism (see Figure 4.6). Regarding energy consumption, *Shared* reservation policy similarly results in a 6% of reduction. This illustrates the importance of providing early and enough resource for improving the effectiveness of straggler detection mechanisms.

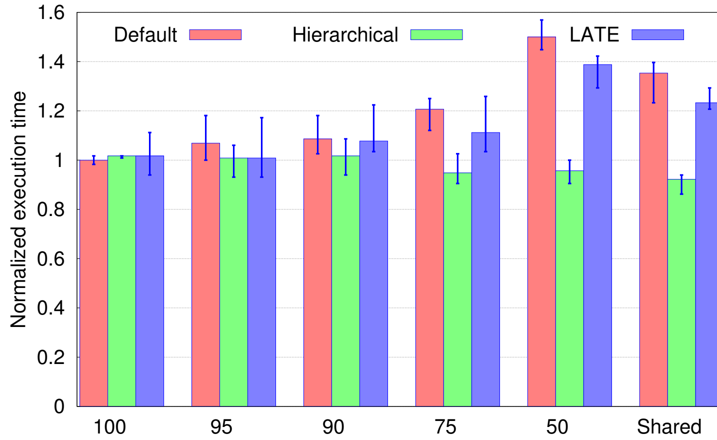


Figure 4.6 – Execution time comparison.

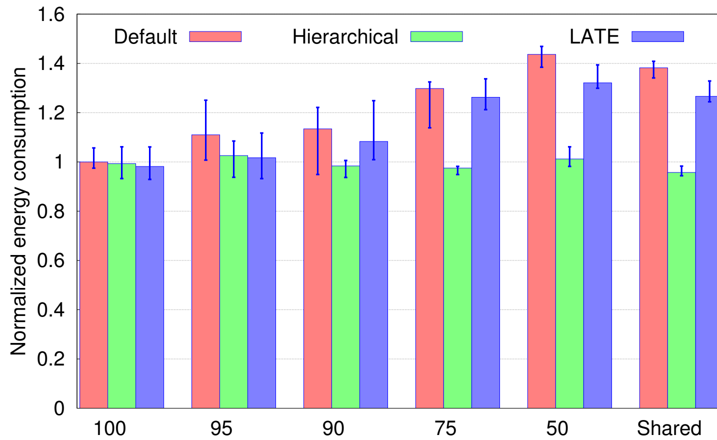


Figure 4.7 – Energy consumption comparison.

Amongst the reservation policies, the *Shared* policy appears to have the best effectiveness, especially when the *Hierarchical* is used. This is due to the nature of this sharing reservation policy where (i) the speculative copies can have early resource to run and (ii) the regular tasks take the remaining resources and maximize the resource utilization. However, both *Default* and *LATE* result in longer execution time (up to 35% and 23% respectively) and higher energy consumption (up to 38% and 27% respectively) when *Shared* is used. In the next section, we use the characteristics, obtained when using our metrics, to explain these results.

In brief, we can conclude that the resource reservation policy can strongly impact the performance and energy efficiency of straggler detection mechanism.

4.3.4.3 Evaluation of Straggler Detection Mechanism Using Proposed Metrics

In this section, we discuss why existing metrics cannot provide the relevant explanation for various execution behaviors of different straggler detection mechanisms. Subsequently, we demonstrate that our metrics can easily interpret these behaviors.

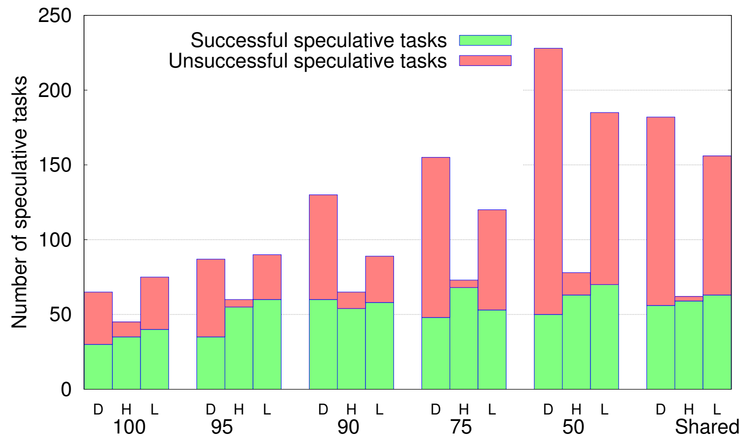


Figure 4.8 – Number of speculative copies.

Misleading Existing Metrics. We have discussed existing evaluation metrics in Table 4.2. First, we consider the *number of speculative copies*. Comparing the results shown in Figures 4.6, 4.7 and 4.8, we observe that there is no clear correlation between the number of speculative copies and the reduction in execution time or energy consumption. As an example, the higher number of speculative copies launched does not result in shorter execution time. Especially in the case of *Default* mechanism with *50* scenario, it launches 2.9x more copies than the *Hierarchical*, but the job takes roughly 36% longer time to finish.

At this point, we might think that the *number of successful copies* could be a better metric for evaluating the effectiveness of straggler detection mechanisms. However, results show that this metric again does not accordingly reflect the execution time reduction nor the energy consumption contraction. For instance, although *LATE* mechanism has a high number of successful speculative copies in many scenarios, it does not come with a high execution time reduction, (see Figure 4.6 and 4.8 at *90*, *50* and *Shared* scenarios).

To this end, it is insufficient and sometimes even imprecise to use the existing metrics on interpreting the effectiveness of straggler detection mechanisms.

Interpreting Results With Our Metrics. Hereafter, we illustrate how to use our metrics to interpret the impact of different straggler detection mechanism on performance and energy consumption.

First, we provide the characteristics of the three straggler detection mechanisms in Table 4.8. We observe that the characteristics of each straggler detection mechanism mostly stay the same through different resource reservation scenarios. These values will be used as primary guidelines to explain and understand the impact of each straggler detection mechanism on the cluster's behavior.

As shown in Figure 4.6, the *Default* results in a longer execution time compared to *LATE* and, especially, to *Hierarchical* in most of the cases. This is due to its low *Precision* which results in a high number of unsuccessful speculative copies (Figure 4.8). These unsuccessful copies compete with regular tasks and lead to a significant degradation in performance (up to 36% degradation in the case of *50*). These copies also result in a high amount of wasteful energy consumption (31% extra energy consumption with the *50* policy).

Table 4.8 – The characteristics of the three straggler detection mechanisms with different resource reservation ratios. These data are computed using the monitoring information of all running tasks during the execution.

Ratio	Mechanism	Precision	Recall	Detection Latency	Undetected Time
100	D	0.62	0.36	0.41	2.50
	H	1.00	0.22	0.50	2.52
	L	0.69	0.45	0.36	2.26
95	D	0.55	0.41	0.17	2.00
	H	0.99	0.29	0.23	2.08
	L	0.68	0.60	0.18	2.00
90	D	0.53	0.42	0.15	2.03
	H	1.00	0.30	0.03	2.10
	L	0.63	0.63	0.16	2.01
75	D	0.58	0.57	0.08	2.08
	H	0.99	0.29	0.05	2.12
	L	0.67	0.65	0.18	2.02
50	D	0.57	0.68	0.01	2.08
	H	0.96	0.32	0.03	1.99
	L	0.60	0.70	0.09	2.04
Shared	D	0.55	0.56	0.32	2.01
	H	0.99	0.29	0.65	2.03
	L	0.65	0.60	0.17	2.28

Considering *LATE*, it results in a slightly lower number of unsuccessful copies (see Figure 4.8), as it has a higher *Precision*. However, as we have mentioned, its *Precision* and *Recall* are still relatively low. As a result, *LATE* can sometimes result in an increment of 31% in execution time and 23% in energy consumption, with *50* policy.

On the other hand, *Hierarchical* mechanism, which targets a high *Precision*, results in a more efficient speculation with a low unsuccessful number (see Figure 4.8). More importantly, as *Hierarchical* is designed to target the potentially longest stragglers, it has closely similar *Undetected Time* compared to *Default*. Thus, the non-detected stragglers of *Hierarchical* have small impact on execution time. Consequently, *Hierarchical* leads to an efficient execution in terms of both performance and energy consumption. Figure 4.6 and 4.7 show that it can result in a reduction of 10% and 6% in execution time and energy consumption, respectively, when *Shared* policy is used, compared to *Default* in *100* scenario. Compare between three straggler detection mechanisms in *Shared* scenario, *Hierarchical* can result in a reduction of 32% in execution time, and 31% energy consumption reduction, compared to *Default* mechanism. This result proves that the *Hierarchical*, which was designed targeting high *Precision* and energy efficient, has successfully accomplished its goals.

In summary, the diverse behaviors of the cluster, caused by the impact of different detection mechanisms, are clearly explainable with the help from our evaluation metrics.

4.3.5 Evaluating *Hierarchical* with Different Applications and Slow-node Thresholds

In this section, we evaluate *Hierarchical* in Hadoop cluster. *Hierarchical* is configured to run on the top of two state-of-the-art straggler detection mechanisms: *Default* [31] and *LATE* [131]. We choose to run two representative Big Data applications from the Puma benchmark suite [1]. We also tune the value of the slow-node threshold *beta* and evaluate the impact of different values on performance and energy consumption. Hereafter, we mention in detail the experiment methodology of our evaluation.

4.3.5.1 Experimental Setup

Testbed. All of our experiments were conducted on a cluster of 21 nodes from the Nancy site of Grid5000 testbed [59]. We configured the cluster with one master and 20 workers. Each node in the cluster is equipped with 4-core Intel 2.53 GHz CPU, 16 GB of RAM and 1 Gbps Ethernet network. The power consumption of the nodes is monitored by Power Distribution Units. Thereby, we can acquire fine-grained and accurate power consumption values during the experiments. All experiments are run for 10 times and the average values are reported.

Applications. We adopt two widely-used MapReduce applications chosen from the well-known Puma MapReduce benchmark suite [1]. The two applications have different characteristics, where *Sort* is an I/O-intensive application, *WordCount* mainly consumes the computation resources. The input data sizes of the two applications are set to 10 GB. The number of Map and Reduce tasks are both set to 160. It is important to note that *CloudBurst* is not available in Hadoop 2.7.3. Therefore, we do not use it in our experiments.

Straggler Injection. In order to inject stragglers, we use the Dynamic Voltage-Frequency Scaling technique (DVFS) [29] to tune the CPU frequencies (hence the computation capabilities) of nodes. According to the Hadoop production cluster traces at CMU [93], the number of stragglers varies from 0 to 40% of the total number of tasks. We choose a straggler ratio of 20% in our experiments. Thus, we set four nodes out of the 20 workers in our cluster to lower CPU frequencies, which are 1.20 GHz, 1.33 GHz, 1.46 GHz and 1.60 GHz.

Comparisons. We compare the *Hierarchical* straggler detection mechanism with the *Default* straggler detection [31] and *LATE* [131] straggler detection mechanisms. We also compare the results while tuning the value of the *slow-node threshold* β (from 0.2 to 1.0).

4.3.5.2 Experimental Results

Figure 4.9 shows the performance and energy results of a single WordCount job running with different straggler detection mechanisms. We use the default copy allocation method in this experiment. In the x-axis, "D" stands for the *Default* straggler detection mechanism, "L" stands for the *LATE* detection mechanism, "D+Hx" and "L+Hx" stand for using the *Hierarchical* straggler detection mechanism on top of *Default* and *LATE*, respectively, where "x" stands for the value of the slow-node threshold β.

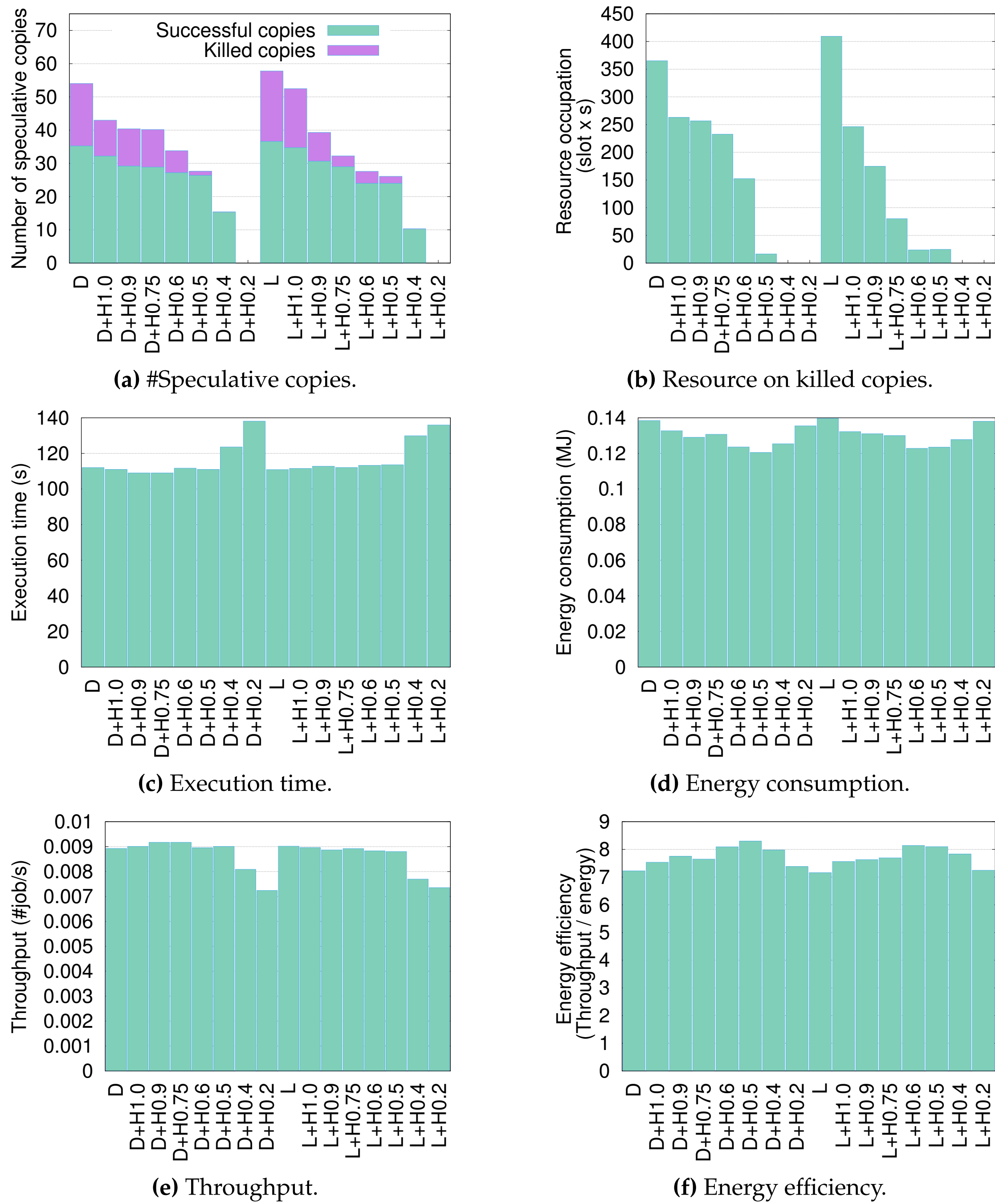
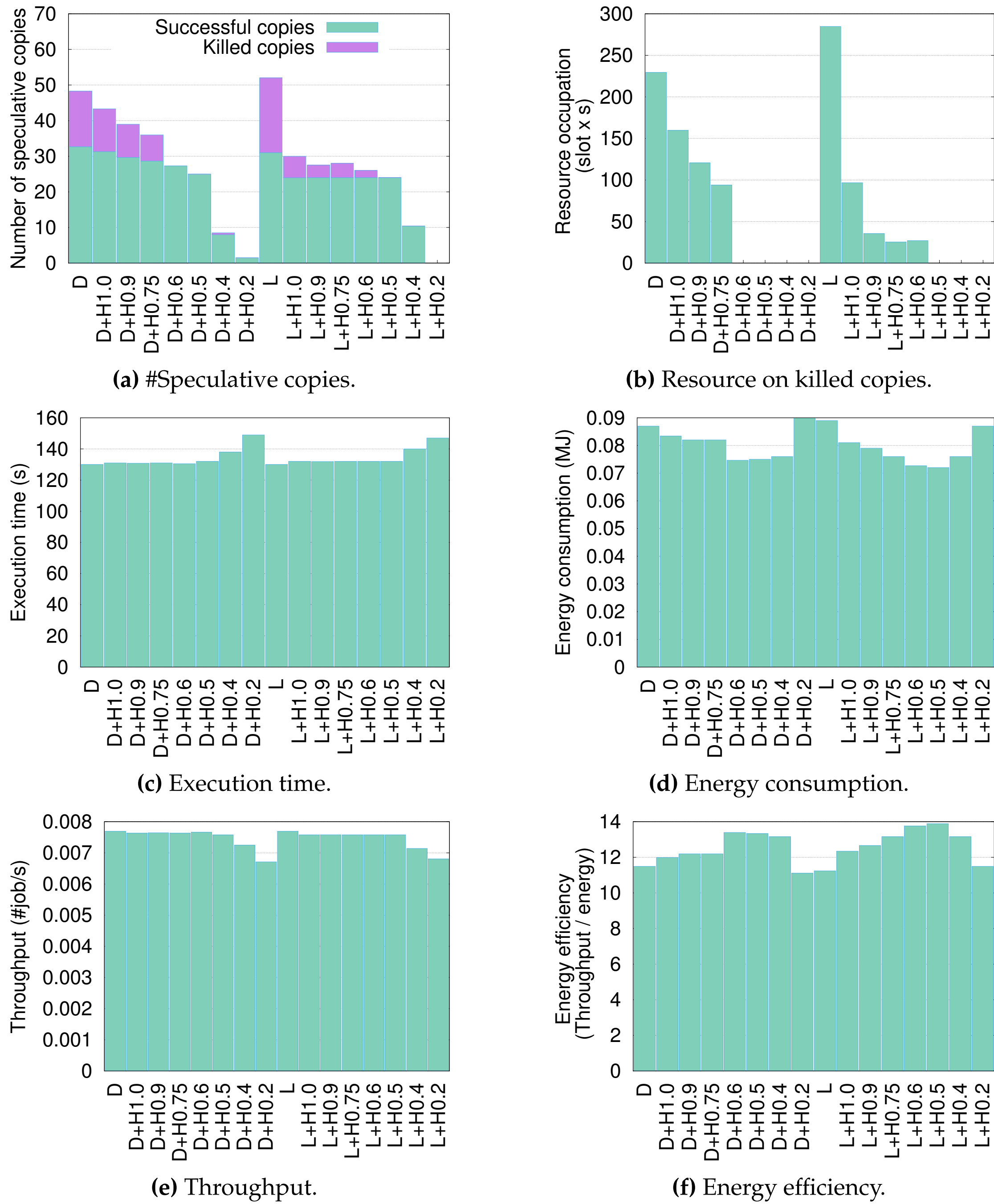


Figure 4.9 – The *WordCount* application with different straggler detection mechanisms.

We have the following observations. First, Figure 4.9a shows that the *Hierarchical* straggler detection mechanism can greatly reduce the number of unsuccessful speculative copies, and the reduction increases with the decrease of β. As a result, the amount of resources wasted on the killed copies is reduced (see Figure 4.9b) by up to 100% compared to *Default* and *LATE* (when $\beta = 0.4 and 0.2$). The total energy consumption is also reduced (see Figure 4.9d) by up to 12% compared to *Default* at $\beta = 0.5$ and up to 11% compared to *LATE* at



(a) #Speculative copies.

(b) Resource on killed copies.

(c) Execution time.

(d) Energy consumption.

(e) Throughput.

(f) Energy efficiency.

Figure 4.10 – The *Sort* application with different straggler detection mechanisms.

$\beta = 0.6$.

Second, adding the *Hierarchical* mechanism does not sacrifice the performance too much (except when $\beta = 0.2$ and 0.4) as shown in Figure 4.9c and 4.9e. In such cases, *Hierarchical* results in a performance degradation. This is mainly because that when β is too small, some of the real stragglers are not detected. For example, when $\beta = 0.5$, almost all speculative copies of detected stragglers are successful. With smaller values of β, the number of detected

stragglers reduces and the number of successful copies reduces.

Third, the *Hierarchical* straggler detection mechanism can obtain better energy efficiency compared to *Default* and *LATE* (except again when $\beta = 0.2$ and 0.4), as shown in Figure 4.9f. When $\beta = 0.5$, we obtain the best energy efficiency, which is 14% higher than *Default*. With *LATE*, the best energy efficiency is achieved when $\beta = 0.6$ as the energy efficiency is increase by 12.5%.

Similar observations are obtained with the *Sort* application. Figure 4.10 shows the results obtained when running this application. When $\beta = 0.6$, *Hierarchical* improves the energy efficiency by 14.5% compared to *Default*. When *Hierarchical* is used with $\beta = 0.5$, the energy efficiency is increased by 16.5% compared to *LATE*.

In summary, these results demonstrate that the Hierarchical *straggler detection mechanism can greatly improve the energy efficiency while guaranteeing a comparable performance.*

4.4 Conclusion

In this chapter, we demonstrate that existing evaluation metrics can result in misleading information while evaluating straggler detection mechanisms. Targeting this issue, we introduce a set of dedicated metrics to evaluate straggler detection mechanisms. Besides, we present a mathematical intuition for linking the proposed metrics to the performance and energy consumption overheads.

The proposed metrics indicate that state-of-the-art straggler detection mechanisms are not accurate in detecting the real stragglers. They may overly detect normal tasks as stragglers, which results in a high amount of extra energy consumption caused by unnecessary speculative copies. Tackling this aspect, we introduce a new energy-driven straggler detection mechanism. This mechanism focuses on detecting the stragglers which run on slow nodes. This can reduce the chance of mis-detecting stragglers. As a result, it improves the detection accuracy. The evaluation results indicate that this mechanism indeed can improve the accuracy and reduce by up to 100% of wasteful energy consumption by killed copies.

Chapter 5

Energy-aware Straggler Handling for Big Data Processing Systems

Contents

5.1 Energy-aware Speculative Execution Controller Architecture 76
5.1.1 Allocation Problem Description 76
5.1.2 Copy Allocation Heuristic 77
5.2 Evaluation 78
5.2.1 Experimental Methodology 78
5.2.2 Results with the *WordCount* Application 81
5.2.3 Results with the *Kmeans* Application 86
5.2.4 Results with the *Sort* Application 87
5.3 Conclusion 89

ENERGY consumption is an important concern for Big Data processing systems, which results in huge monetary cost for Big Data processing system operators [46]. Due to the hardware heterogeneity and contentions between concurrent workloads, straggler mitigation is important to many Big Data applications running in Big Data processing systems. The speculative execution technique is widely used to handle stragglers.

Allocating speculative copies to different nodes can result in different performance and energy consumption results [89]. For instance, launching speculative copies on nodes with a small number of running tasks can result in short execution time but can lead to high power consumption (as shown in Chapter 3). Unfortunately, existing copy allocation methods do not consider this aspect.

In this chapter, we take a step forward to improve the energy efficiency of straggler handling in Big Data processing systems. Accordingly, we introduce an energy-aware copy

allocation method. This method takes into consideration the difference in performance as well as energy consumption of different copy allocations. Consequently, it allocates speculative copies to the resources which offer low energy consumption with high performance. Our straggler handling mechanism is implemented in Hadoop. Experimental results show that our energy-aware copy allocation method can reduce energy consumption while guaranteeing performance comparable with state-of-the-art copy allocation methods.

Speculative Copy Allocation Matters

We have observed that there is a trade-off between performance and energy consumption for speculative copies executing on different nodes, according to the current status of the nodes. This trade-off was discussed in detail in Chapter 3. Unfortunately, existing copy allocation methods do not pay much attention on this aspect. For instance, *Default* [31] follows the simple First Come First Serve (FCFS) policy to allocate copies to the first available resource, without considering any of the performance and energy objectives. In *Mantri* [6], the task placement is mainly based on the performance objective. A speculative copy is launched on resource only if there is a fair chance that it can finish earlier than the original task. As a result, these copy allocation methods may result in high energy consumption for Big Data processing systems.

In the following sections, we present a novel straggler handling mechanism. This mechanism is equipped with an energy-aware method to allocate speculative copies.

5.1 Energy-aware Speculative Execution Controller Architecture

In this section, we firstly describe the notion of speculative copy allocation. Then, the detailed design of our speculative copy allocation method is discussed. Finally, the implementation of the this allocation method is introduced.

5.1.1 Allocation Problem Description

Let us consider a list of detected stragglers. A speculative copy allocation method is needed to handle these stragglers. Its mission is to map each straggler to a node with available slots and start a copy of the straggler on that node, in order to optimize the overall energy efficiency of speculative execution. Let us assume that there are S copies (s_i, $i \in [1, S]$) to be launched and N nodes (n_j, $j \in [1, N]$) with available slots to execute these copies. We can easily formulate the copy allocation problem as a variant of the classic Bin-Packing problem, where the size of each bin (*i.e.*, a node) equals to the number of available slots in the bin. Thus, the copy allocation problem is a NP-hard problem. In the next subsection, we propose a heuristic to obtain a good solution to this problem.

Resource Availability. When the value of N is small, there are not many choices to allocate speculative copies. As a result, it limits the improvement in energy efficiency which is brought by the copy allocation method. Thus, we adopt the same methodology as Delay scheduling [128]. That is, we first check the value of N when allocating speculative copies. If N is small, then we wait for a few seconds to have more idle nodes. By default, $N = 1$ is considered as small. In our experiments, we wait three seconds when $N = 1$.

5.1.2 Copy Allocation Heuristic

There are many existing heuristics to solve Bin-Packing problems (*e.g.*, First-Fit and Best-Fit heuristics). In this work, we propose a heuristic similar to Best-Fit heuristic to address our copy allocation problem. At first, there is a set of detected stragglers, provided by the underlying straggler detection mechanism. Then, the speculative copy allocation method searches the node that can best fit each straggler sequentially.

Let us define the fitness of mapping a straggler to a node according to the energy efficiency of the mapping. The energy efficiency is affected by both the energy consumption and the performance of job. Given any map from Straggler i to Node j, we first provide two models to estimate the job execution time variation and the energy consumption variation caused by launching a speculative copy of Straggler i on Node j.

Execution Time Variation Estimation. At first, we sort the list of detected straggler according to their criticalness. A straggler, which is expected to finish further in the future, is considered more critical. The most critical straggler by nature dominates the job execution time. Handling critical stragglers can directly contribute to the reduction of job execution time. The most critical straggler is placed at the top of the list. This sorting is similar to the sorting algorithm we used for the *Hierarchical* straggler detection mechanism as described in Chapter 4.

Thus, we can estimate the job execution time variation ΔT_{ij} caused by launching a copy of Straggler i on Node j using the difference between the task execution time of Straggler i before and after launching the copy. Assume that Straggler i is running on Node k.

$$\Delta T_{ij} = \alpha_k \times \frac{(1 - \text{progress}_i) \times \text{input}_i}{\text{Perf}^k_{\text{host}}} - \alpha_j \times \frac{\text{input}_i}{\text{Perf}^j_{\text{host}}} \tag{5.1}$$

where the first term stands for the left over time for the straggler to finish if no copy is launched and the second term stands for the execution time of the copy on Node j. In this Equation, α is the average slowdown ratio, which was introduced in Chapter 4.

Energy Consumption Variation Estimation. Executing a new copy consumes more energy. However, it at the same time saves energy as it shortens the execution time of the straggler. We can formulate the energy consumption variation caused by launching a copy of Straggler i on Node j as follows.

$$\begin{aligned} \Delta E_{ij} &= (\mathcal{P}_k + \mathcal{P}_j) \times T_s - (\mathcal{P}_k + \mathcal{P}'_j) \times T_c \\ &= \mathcal{P}_k \times \Delta T_{ij} + \mathcal{P}_j \times T_s - \mathcal{P}'_j \times T_c \end{aligned} \tag{5.2}$$

where T_s equals to the first term of Equation 5.1 and T_c equals to the second term of Equation 5.1. $\mathcal{P}_j$ and $\mathcal{P}'_j$ are the power consumption of Node j before and after adding a copy of Straggler i, which are computed using Equation 4.2.2.1 in Chapter 4.

Best-fit Copy Allocation Heuristic. Given the above two models and the definition of energy efficiency, we can select the map which yields the best ΔE_{ij} result as the Best-Fit solution, *i.e.*, the highest improvement to energy efficiency. **Algorithm 1** presents the general flow of our copy allocation heuristic.

Initiation: Straggler i at the head of the straggler list is selected (line **2**). The initial value of energy consumption variation ΔE is set to zero (line **3**).

Searching for the best copy-node mapping: Straggler i is mapped to each idle Node j (line **4**). The energy consumption variation ΔE_{ij} is calculated for this mapping (line **5**). If the value of ΔE_{ij} for this mapping is higher compared to the best energy consumption variation ΔE, Node j is selected as new best mapping and the best energy consumption variation is set with ΔE_{ij} (line **6–9**).

Allocating speculative copies: Straggler i is removed from straggler list (line **11**). A speculative copy of Straggler i is launched on the node that is labeled as best mapping (line **12**).

```
1  while straggler_list is not empty do
2      Straggler i is the head of straggler_list;
3      best_fitness = 0;
4      for Node j in idle_nodes do
5          calculate ΔE_ij using Equation 5.2;
6          if ΔE_ij > best_fitness then
7              best_map = j;
8              best_fitness = ΔE_ij;
9          end
10     end
11     remove Straggler i from straggler_list;
12     launch a copy of Straggler i to Node best_map;
13 end
```

Algorithm 1: Speculative copy allocation heuristic.

5.2 Evaluation

In this section, we evaluate our copy allocation method in real Hadoop cluster and compare it with existing copy allocation methods. We implement our allocation method in the Hadoop 1.2.1 stable version, with roughly 1500 lines of Java code. Users can easily enable or disable our method using the Hadoop general configuration file.

5.2.1 Experimental Methodology

In this section, we discuss in detail our experimental methodology.

Testbed

All of our experiments were conducted on a cluster of 21 nodes from the Nancy site of Grid5000 testbed [59]. We configured the cluster with one master and 20 workers. Each node in the cluster is equipped with 4-core Intel 2.53 GHz CPU, 16 GB of RAM and 1 Gb/s Ethernet network. The power consumption of the nodes is monitored by Power Distribution Units. Thereby, we can acquire fine-grained and accurate power consumption values during the experiments. All experiments are run 10 times and the average values are reported.

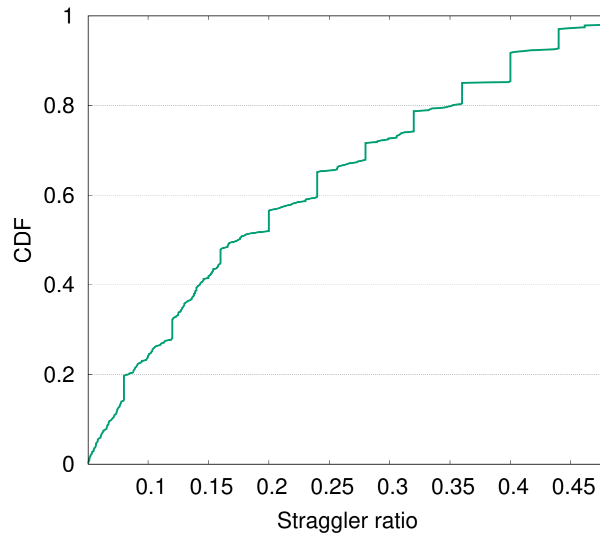


Figure 5.1 – The CDF of straggler ratio per job extracted from the traces of Hadoop production clusters at CMU

Straggler Detection Mechanisms

In our experiments, we use the *Hierarchical* straggler detection mechanism (introduced in Chapter 4). *Hierarchical* is configured to run on top of state-of-the-art straggler detection mechanisms *Default* and *LATE*. The slow-node threshold β of *Hierarchical* is set to 0.5. With this setting, *Hierarchical* is expected to significantly reduce the number of unnecessary speculative copies (as shown in Chapter 4). All copy allocation methods run with this straggler detection setup throughout our experiments.

Applications

We adopt three widely-used MapReduce applications chosen from the well-known Puma MapReduce benchmark suite [1]. Two of them, which are *WordCount* and *Sort*, are used in the evaluations of Chapters 3 and 4. The third application is *Kmeans*. This is a compute-intensive application which classifies input data into multiple clusters. In our experiments, we use *Kmeans* to classify movies based on the ratings from anonymous users [1].

The input data sizes of the three applications are all set to 10 GB [1]. The numbers of Map and Reduce tasks are both set to 160.

Straggler Injection

For injecting stragglers in our experiments, we study the ratio of straggler occurrence in Hadoop production clusters. Specifically, we analyze the traces from Hadoop production clusters at CMU [93]. These traces were collected in October 2012. Figure 5.1 depicts the CDF of straggler ratio (*i.e.*, the ratio of stragglers over total number of tasks per job) for more than 1000 jobs. We observe that the straggler ratio varies within a large range, from 0.05 up to 0.45.

In our experiments, we set the straggler ratio at 0.2, which is close to the average straggler ratio as shown in Figure 5.1. In order to inject stragglers, we use the Dynamic Voltage-Frequency Scaling technique (DVFS) to tune the CPU frequencies (hence the capabilities) of nodes. Given the straggler ratio of 0.2, we set four nodes out of 20 workers in our cluster to lower CPU frequencies, which are 1.20 Ghz, 1.33 GHz, 1.46 GHz and 1.60 GHz (the default frequency is 2.53 GHz).

Comparisons

We compare our proposed copy allocation heuristic (denoted as *Smart*) with the default speculative copy allocation method of Hadoop, denoted as *Default*. In addition, we implemented in Hadoop the following two copy allocation methods:

Performance-driven Copy Allocation. It differs from *Smart* in that it launches speculative copies on nodes which yield the best performance. Specifically, the general flow of this method is similar to *Smart* (as shown in Algorithm 1). The major difference is that the *Performance-driven* copy allocation method considers only the performance of speculative copies (as computed by Equation 5.1).

Power-driven Copy Allocation. With this copy allocation method, speculative copies are launched on nodes such that the speculative copies yield the lowest additional power consumption. The additional power consumption ΔP_{ij} for launching copy i on node j is computed as follows:

$$\Delta P_{ij} = \mathcal{P}'_j - \mathcal{P}_j \tag{5.3}$$

where $\mathcal{P}'_j$ is the power consumption of Node j after launching Copy i and $\mathcal{P}_j$ is the power consumption of Node j before launching Copy i (as computed in Equation 5.2).

Metrics

The metrics that we use in our evaluation are listed as follows:

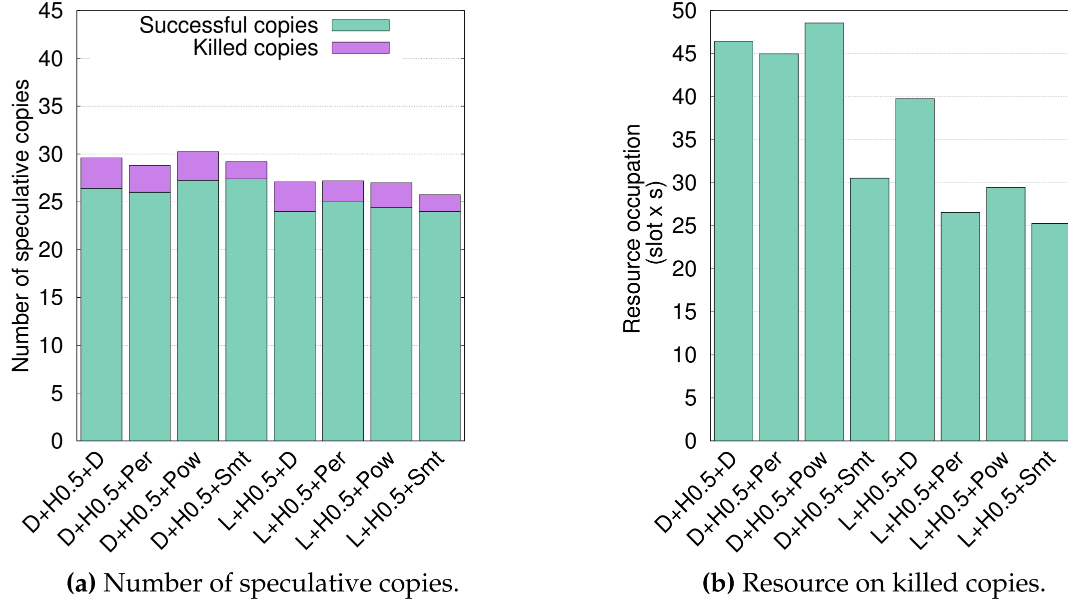
Number of successful speculative copies. This is the total number of speculative copies which successfully finish during the job execution.

Number of killed speculative copies. In contrast with the aforementioned metric, this metric represents the total number of speculative copies which could not finish before their original tasks and get killed.

Resource consumption on killed speculative copies. This is calculated by summing up the amount of time that killed speculative copies occupy resources (*i.e.*, slots).

Execution time. The amount of time from the start of the first task until the completion of the last task.

Energy consumption. The total energy consumed by the cluster during the execution of a job.



(a) Number of speculative copies.

(b) Resource on killed copies.

Figure 5.2 – The *WordCount* application with different copy allocation methods: Comparison on number of successful speculative copies, number of killed copies and resource consumption on killed copies.

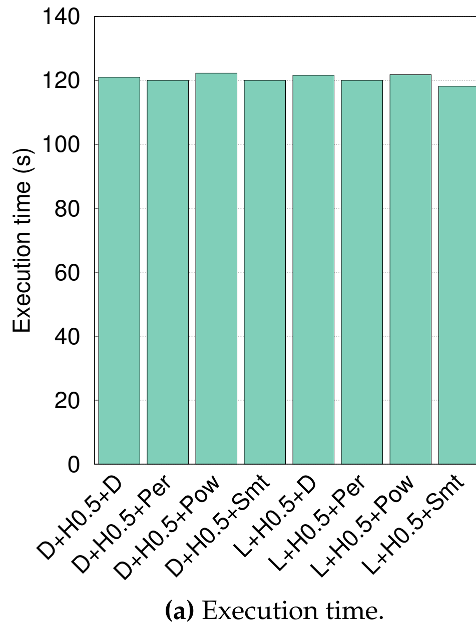
Energy efficiency. This metric is calculated as the ratio of performance over energy consumption. The performance of a job is computed as the inverse of the job's execution time.

Hereafter, we compare four speculative copy allocation methods when running *WordCount*, *Kmeans* and *Sort* applications. We start the evaluation with results when running the *WordCount* application.

5.2.2 Results with the *WordCount* Application

We evaluate in total eight combinations of the straggler detection mechanisms and copy allocation methods. Specifically, we denote the *Default* straggler detection mechanism by *D* and *LATE* by *L*. The *Hierarchical* straggler detection, which is denoted by *H*, is applied on top of these detection mechanisms. For copy allocation methods, we again denote *Default* copy allocation method by *D*. Our energy-aware copy allocation method is denoted by *Smt*. The *Power-driven* copy allocation is denoted by *Pow* and the *Performance-driven* copy allocation method is denoted by *Per*.

Number of Speculative Copies. Figure 5.2 displays the results of running a single *WordCount* job with different speculative copy allocation methods. We observe that the number of successful speculative copies are very close between the four copy allocation methods (as shown in Figure 5.2a). Similarly, the difference in number of killed speculative copies between four methods is small. This is because *Hierarchical* with β of 0.5 has eliminated a large number of inaccurate straggler detection (as shown in Chapter 4). As a result, the detected



(a) Execution time.

Figure 5.3 – The *WordCount* application with different copy allocation methods: Comparison on execution time.

stragglers are mostly actual stragglers with significantly longer execution time, compared to average task execution time. This explains the low number of killed speculative copies.

We further dig into the small difference between the four copy allocation methods, with respect to the number of speculative copies and the resource consumption of killed copies. We notice that this small difference between allocation methods has its own meaning. For instance, *Power-driven* and *Default* have higher number of killed copies as well as higher resource consumption on killed copies, compared to *Performance-driven* and *Smart*. The *Power-driven* copy allocation method launches speculative copies on the nodes with the lowest power cost. Usually, this will lead the speculative copies to be launched on nodes that are already hosting many on-going tasks. As a result, the performances of these copies are low. At the end, these copies can get killed. This explains the high resource consumed by killed copies of *Power-driven*. For *Default*, it does not take into account either performance or energy consumption while allocating copies. The speculative copies may be allocated to nodes which have poor performance. Consequently, this result in higher resource wasted on killed speculative copies.

Performance Evaluation. We continue the evaluation with results related to execution time. Figure 5.3 presents these results. We observe that the *Power-driven* copy allocation method has a relatively long execution time (as shown in Figures 5.3a). This is expected since the *Power-driven* copy allocation method does not take into account the performance when launching speculative copies. As a result, it can launch speculative copies to resources with low performance. These copies take long time to finish. Thereby, they increase the job's execution time. Similarly, the *Default* copy allocation method has also relatively long execution time, since it does not take into consideration the performances of different speculative copy allocations.

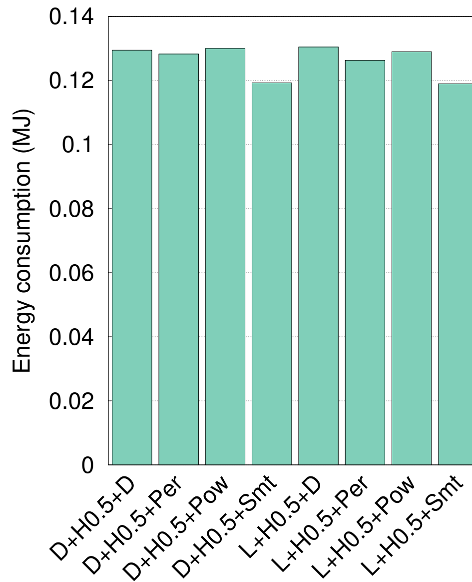


Figure 5.4 – The *WordCount* application with different copy allocation methods: Comparing overall energy consumption.

In contrast, *Smart* and *Performance-driven* have slightly shorter execution time. The *Performance-driven* method aims to allocate speculative copies to resources on which the copies can have the highest performance. As a result, they finish earlier and shorten the job's execution time. The *Smart* copy allocation method reduces the resource occupation of killed speculative copies by up to 36% compared to *Default* (as shown in Figure 5.2b). Consequently, it reduces the resource contention caused by these copies. As a result, successful speculative copies as well as regular tasks can finish earlier. This in turn reduces the execution time of the *WordCount* application.

Energy Consumption Evaluation. Let us take a look at the overall energy consumption of *WordCount* job with different copy allocation methods (as shown in Figure 5.4). We observe that the *Smart* copy allocation method results in the lowest energy consumption amongst four copy allocation methods. Compared to *Default*, the energy reduction is up to 9%. Besides, the *Power-driven* copy allocation method has fairly high energy consumption. It can be up to 8.5% higher energy consumption compared to *Smart*.

This is because *Power-driven* only considers the power cost and omits the performance of speculative copies. Since energy consumption is the product of power and execution time, the overall energy consumption can still be higher (the reduction in power consumption cannot compensate the increase in execution time when allocating speculative copies). Besides, the *Performance-driven* copy allocation method can result in high energy consumption. This is because the speculative copy allocations with the best performance may have high energy consumption (as described in detail in Chapter 3). This result emphasizes the important impact of different speculative copy allocation on both performance and energy consumption.

Energy Efficiency Comparison. Figure 5.5 presents the comparison on energy efficiency of four copy allocation methods. As we can observe, *Default* and *Power-driven* both have rela-

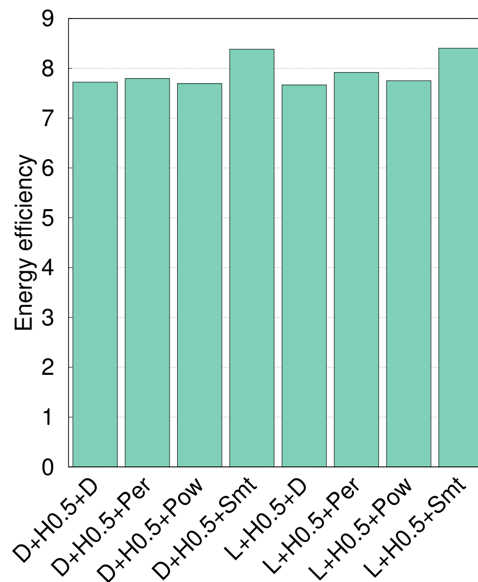


Figure 5.5 – The *WordCount* application with different copy allocation methods: Comparing energy efficiency.

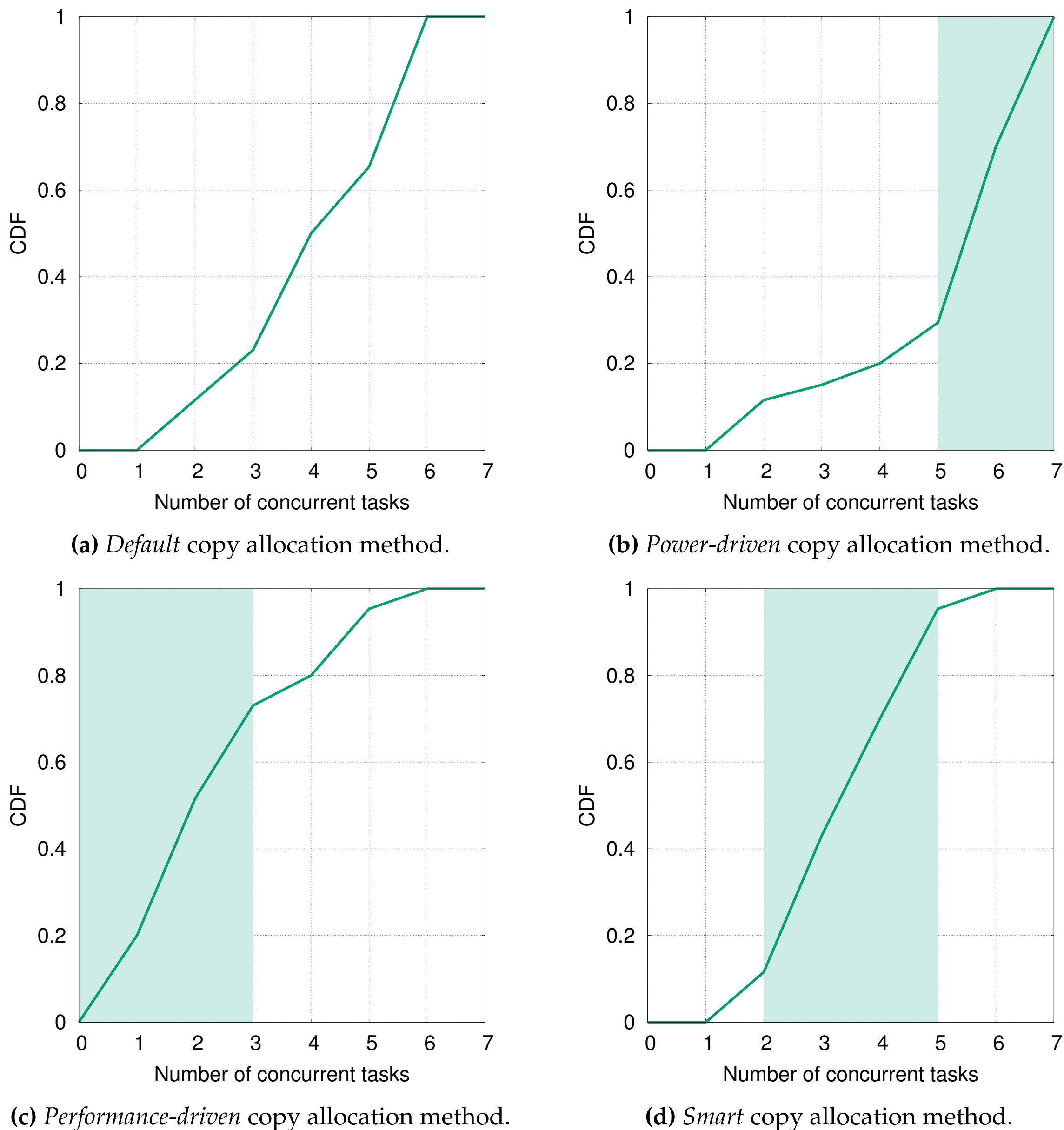
tively low energy efficiency. The energy efficiency of *Performance-driven* is only 2% higher. On the other hand, *Smart* can increase the energy efficiency by up to 12%, compared to the *Default* copy allocation method.

Getting Deeper: Speculative Copy Allocation. In order to better understand how different methods allocate speculative copies, we dig into the status of the nodes on which speculative copies are launched. Figure 5.6 depicts the CDF of the number of concurrent tasks per node when launching speculative copies, with different copy allocation methods. First, we notice that the number of concurrent tasks per node distributes evenly from 1 to 6, when speculative copies are launched by *Default* (as shown in Figure 5.6a). This reflects exactly how *Default* was implemented, which does not imply any specific constraint on which nodes speculative copies should be launched.

Second, *Power-driven* copy allocation method tends to launch speculative copies on nodes, which already have a high number of running tasks. For instance, more than 70% of speculative copies are launched on nodes with at least 6 on-going tasks (as shown in Figure 5.6b). *Power-driven* launches copies mainly on this type of nodes because it will cause small extra power consumption to the node.

Third, *Performance-driven* copy allocation method prioritizes launching speculative copies to the nodes that host a small number of on-going tasks. More than 70% of speculative copies are launched on nodes, which is hosting less than 3 on-going tasks. On such nodes, speculative copies can run faster since there is small contention between tasks. As a result, *Performance-driven* reduces the execution times of speculative copies. This results in a shorter job execution time (as shown in Figure 5.3a).

Fourth, *Smart* launches speculative copies mostly on nodes with moderate number of on-going tasks. 80% of speculative copies are launched on nodes that host from 3 to 5 on-going tasks. On these nodes, speculative copies can run with a high performance, whereas



(a) *Default* copy allocation method.

(b) *Power-driven* copy allocation method.

(c) *Performance-driven* copy allocation method.

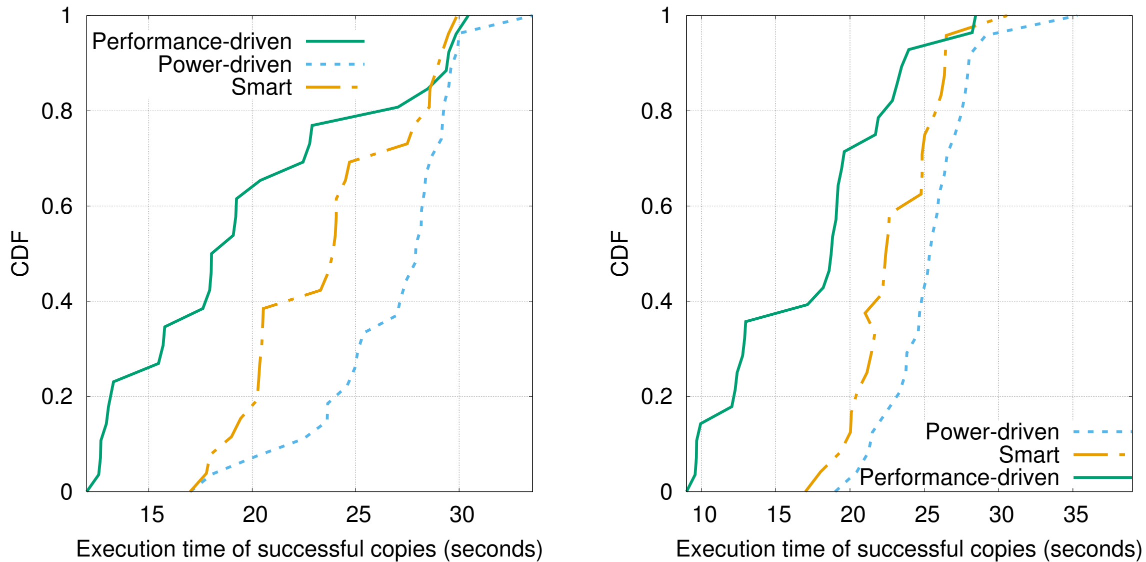
(d) *Smart* copy allocation method.

Figure 5.6 – The *WordCount* application: Comparing the speculative copy allocations of diffent allocation methods.

the energy consumption of these speculative copies are expected to be minimal.

Zoom in on the Execution Time of Successful Speculative Copies. Finally, we focus on the execution time of successful speculative copies between three copy allocation methods, including *Performance-driven*, *Power-driven* and *Smart*. Figure 5.7 presents the CDF of the execution time of successful speculative copies with these methods when using either *Default+Hierarchical* or *LATE+Hierarchical*.

As expected, we find that the *Performance-driven* copy allocation method has the shortest



(a) When the *Default* and *Hierarchical* straggler detection mechanisms are used.

(b) When the *LATE* and *Hierarchical* straggler detection mechanisms are used.

Figure 5.7 – The *WordCount* application: The number of concurrent tasks per node when allocating speculative copies with different copy allocation methods.

median execution time for successful speculative copies. On average, the execution time of successful speculative copies is 25% and 33% shorter compared to *Smart* and *Power-driven*, respectively. However, we can notice that *Performance-driven* also has some long running speculative copies. This is due to the fact that there may be no nodes with high performance when speculative copies are allocated. In this case, *Performance-driven* may have to allocate speculative copies to nodes with a high number of on-going tasks. On such nodes, speculative copies have longer execution time. This phenomenon is also observed in Figure 5.6c.

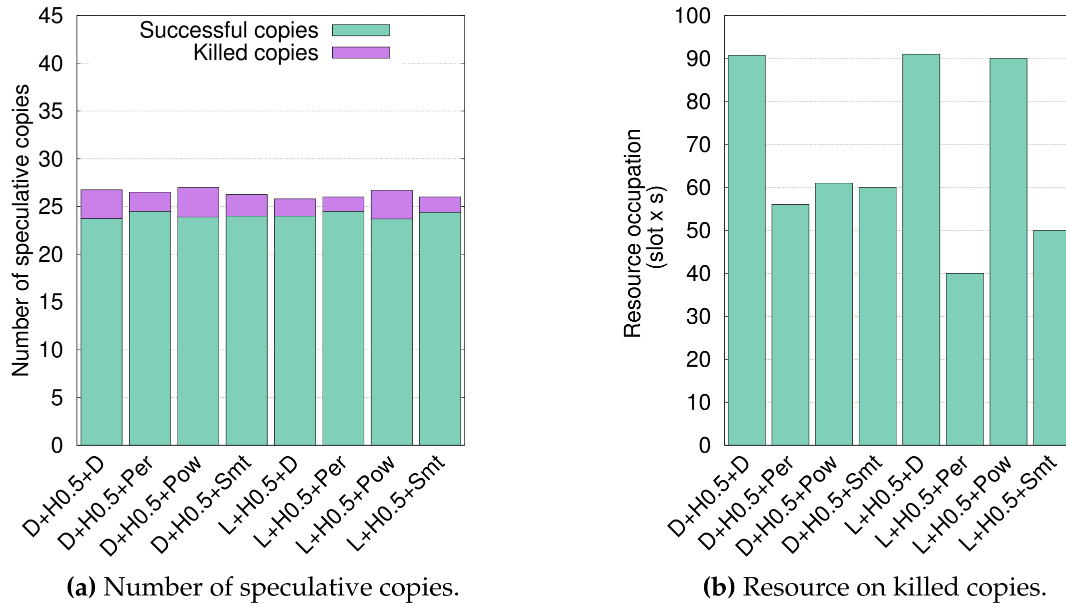
The *Smart* copy allocation method results in medium execution time of successful speculative copies. As shown in Figures 5.7a and 5.7b, the execution time of successful copies varies in smaller range, compared to the cases when *Performance-driven* or *Power-driven* is used. This is because *Smart* launches speculative copies mainly on nodes with moderate number of on-going tasks. Speculative copies running on these nodes have good performance while resulting in a low energy consumption. This is the major factor that leads to the high energy efficiency of *Smart*.

5.2.3 Results with the *Kmeans* Application

Figures 5.8 and 5.9 present the results when running with *Kmeans* application. We record very similar results compared to what we observe when running with *WordCount*.

In general, *Performance-driven* and *Smart* have relatively higher number of successful speculative copies and lower number of killed copies, compared to *Default* and *Power-driven* (see Figure 5.8a). As a result, *Performance-driven* and *Smart* reduce the resource consumption on killed copies.

In particular, the resource consumption on killed copies of *Performance-driven* is reduced by 56% compared to *Default* and 55% compared to *Power-driven*, when *Hierarchical* is used on



(a) Number of speculative copies. **(b)** Resource on killed copies.

Figure 5.8 – The *Kmeans* application with different copy allocation methods: Comparison on number of successful speculative copies, number of killed copies and resource consumption on killed copies.

the top of *LATE* (as shown in Figure 5.8b). In this case, *Performance-driven* also has 25% lower resource on killed speculative copies, compared to *Smart*. As a result, *Performance-driven* results in a relatively lower execution time, compared to the other three copy allocation methods, as shown in Figure 5.9a.

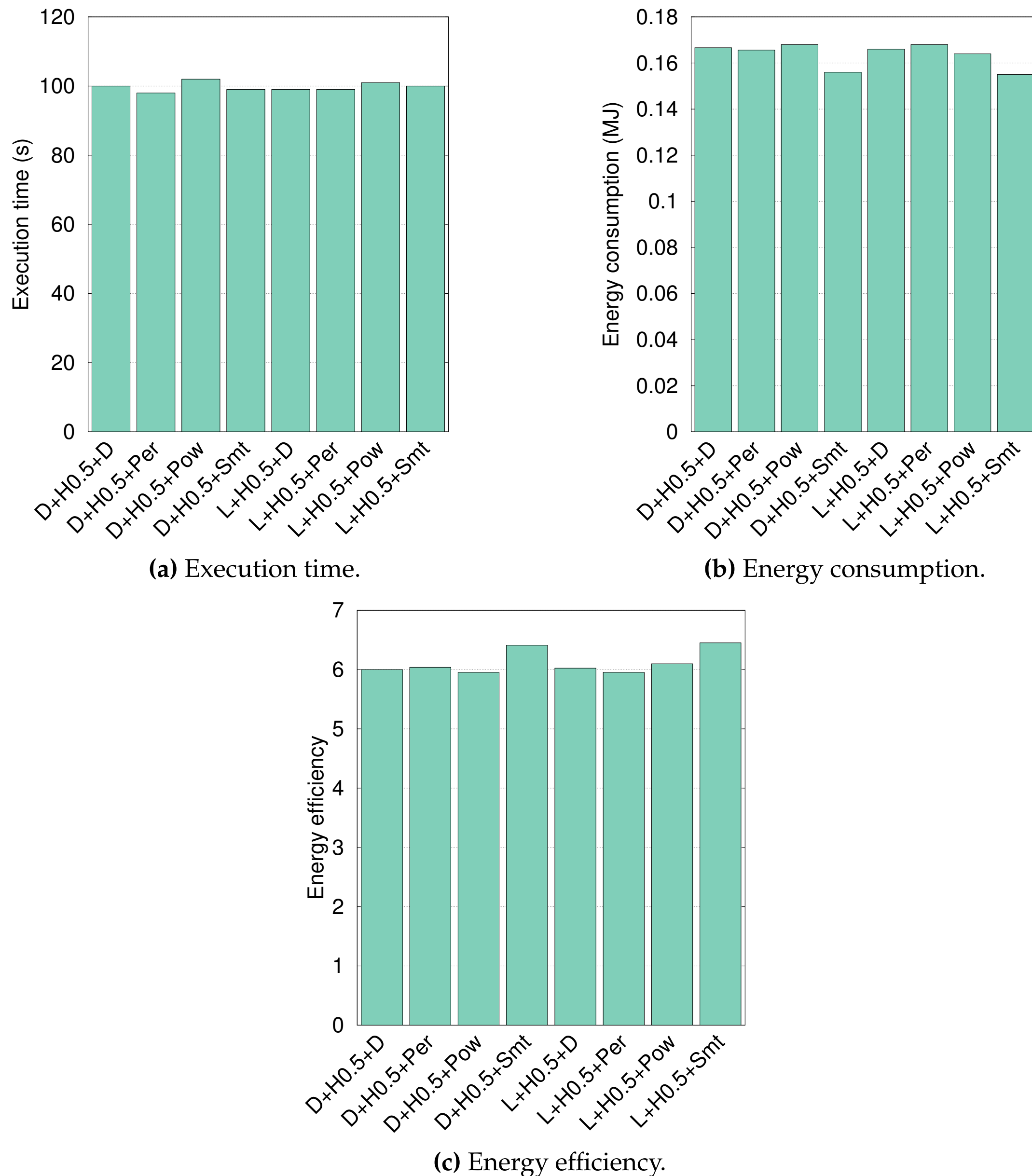
With regard to energy consumption, *Performance-driven* has a high energy consumption. Even worse, it has the highest energy consumption when *Hierarchical* is used on the top of *LATE* (see Figure 5.9b). In the same case, *Smart* has the lowest energy consumption, with 8.5% lower energy consumption compared to *Performance-driven*. This result emphasizes on the important impact of different copy allocations, not only on performance but also on energy consumption.

Finally, *Smart* has the best energy efficiency amongst four copy allocation methods, with up to 10% higher energy efficiency compared to *Power-driven* (as shown in Figure 5.9c). In contrast, *Performance-driven* has relatively low energy efficiency. When *Hierarchical* is used on the top of *LATE*, it even has the lowest energy efficiency, with 9% lower energy efficiency compared to *Smart*.

5.2.4 Results with the *Sort* Application

Figures 5.10 and 5.11 display the results when running with *Sort*. In general, we again record trends similar to what we observed for *WordCount* and *Kmeans*.

As shown in Figure 5.10a, the *Default*, *Performance-driven* and *Smart* allocation methods have no killed speculative copies. The only exception is *Power-driven*. It has a small number of killed copies. This is because *Power-driven* allocates copies to nodes with the lowest additional power cost. Speculative copies on these nodes are most likely to have low performance. As a result, these speculative copies may get killed. Accordingly, these copies with



(a) Execution time.

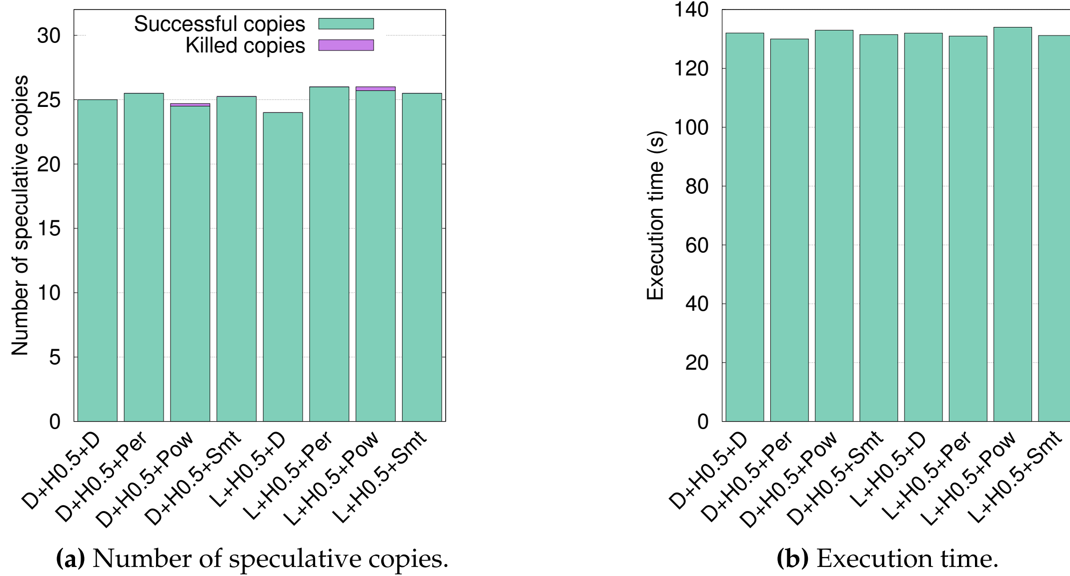
(b) Energy consumption.

(c) Energy efficiency.

Figure 5.9 – The *Kmeans* application with different copy allocation methods: Comparison on execution time, energy consumption and energy efficiency.

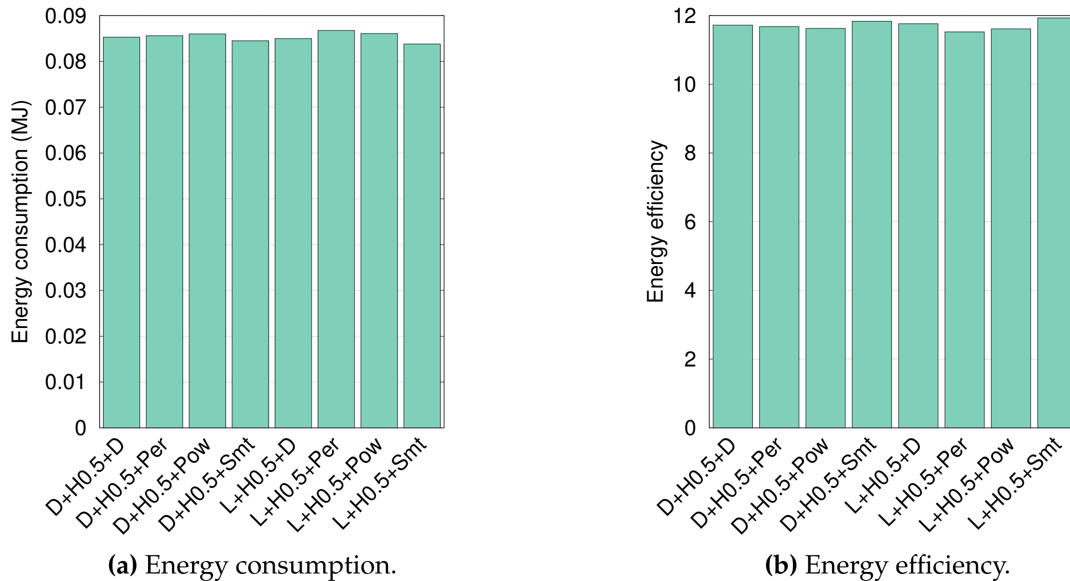
low performance also result in relatively long execution time when *Power-driven* is used, as shown in Figure 5.10b. In contrast, *Performance-driven* and *Smart* have slightly shorter execution time.

Regarding energy consumption, *Smart* again has a relatively low energy consumption, compared to the other three allocation methods. In contrast, the energy consumption is slightly higher when *Performance-driven* is used. This high energy consumption reduces the energy efficiency of *Performance-driven*, disregarding its short execution time. Compared to *Performance-driven*, *Smart* increases the overall energy efficiency by 3.5% (see Figure 5.11b).



(a) Number of speculative copies. **(b)** Execution time.

Figure 5.10 – The *Sort* application with different copy allocation methods: Comparison on number of successful speculative copies, number of killed copies and execution time.



(a) Energy consumption. **(b)** Energy efficiency.

Figure 5.11 – The *Sort* application with different copy allocation methods: Comparison on energy consumption and energy efficiency.

5.3 Conclusion

The increasing trend towards Big Data processing in the clouds [34, 77], the inevitable resource heterogeneity in the clouds [34, 97], and the proliferation of MapReduce applications [98, 101, 114] elevate straggler mitigation to a key issue. Straggler mitigation techniques have been shown to have a potentially high impact on both performance and energy consumption. Thereby, more attention must be paid on improving existing straggler miti-

gation techniques, not only from the performance point of view, but also from an equally important aspect, namely, energy efficiency.

In this chapter, an energy-aware copy allocation method is introduced, taking into consideration this aspect. The evaluation results illustrate that this method can significantly improve the energy efficiency of Big Data processing clusters. The presence of this copy allocation method provides users with more options to select an appropriate speculative copy allocation method, depending on their specific goals and constraints.

What remains to study? The speculative copy allocation method that we present in this chapter can work effectively if there are many resources available. However, having a large amount of free resources to select at the exact moment when stragglers are detected is not trivial. In many cases, the regular tasks of very large jobs fully occupy the resources until the very last wave [93]. In this scenario, passively waiting for free resources will not help. It requires a proactive mechanism to provide appropriate and timely resources for launching speculative copies. In the next chapter, we introduce a new reservation-based straggler handling mechanism to address this issue.

Chapter 6

Improving the Energy Efficiency of Straggler Handling: A Reservation-based Approach

Contents

6.1 WHEN and WHERE Questions: Impacts of the Answers 92
6.1.1 When to Launch: A Fixed Solution is Not Always Good 92
6.1.2 Where to Launch: Heterogeneity Has to be Considered 93
6.1.3 A Motivating Example 95
6.2 Design Overview 96
6.3 Proposed Techniques 97
6.3.1 Window-based Resource Reservation 97
6.3.2 Heterogeneity-Aware Copy Allocation 99
6.4 Methodology 101
6.5 Experimental Evaluation 104
6.5.1 Comparison of Different Speculative Execution Mechanisms 104
6.5.2 Sensitivity Study 108
6.6 Conclusion 114

WHEN and WHERE to launch the task across a large cluster are the classic questions for task scheduling problem in general [80]. How to answer these questions determines the overall performance and energy consumption. The problem of speculative copy scheduling shares the same concerns. How to answer these two questions can strongly affect the impact of the speculative execution on both performance and energy efficiency.

In Chapter 5, we have proposed a solution that answers the question of *where* by taking into consideration the impact of resource heterogeneity. However, this solution only works when there are enough free resources to consider. Unfortunately, this scenario is not always true in Big Data processing systems. In large Big Data jobs, regular tasks dominate the resources, leaving the speculative copies to be starved until the final wave [93]. This phenomenon has been discussed in Chapter 3. The long delay of speculative execution limits the potential improvement that speculative copies can bring. Even worse, these late speculative copies may get killed. This results in wasteful extra energy consumption. Therefore, only addressing the question of *where* is not sufficient, while ignoring the question of *when*.

In this chapter, we firstly demonstrate how insufficient answers to the questions of *when* and *where* can negatively impact the performance and energy consumption. Subsequently, we discuss the deficiencies of existing studies in answering these questions. Our goal is to optimize both the execution time and energy consumption for Big Data processing applications by smartly handling stragglers at runtime. Accordingly, we introduce a new straggler handling mechanism, which adopts the resource reservation approach to proactively provide appropriate and timely resources to speculative copies. First, we introduce a novel resource reservation mechanism for launching speculative copies. The resources with low energy consumption and high performance are proactively reserved for speculative copies. This aims to answer the question of *where* to launch speculative copies. Second, we propose a window-based straggler handling mechanism to dynamically decide the best timing for launching speculative copies. This in turn answers the *when* question.

6.1 WHEN and WHERE Questions: Impacts of the Answers

For speculative executions, there are two main design factors, namely *when* to launch a speculative copy and *where* (*i.e.*, on which node) to launch it. We study existing speculative mechanisms and analyze their effectiveness in addressing the two questions.

6.1.1 When to Launch: A Fixed Solution is Not Always Good

Concerning the question of *when*, we analyzed the well-known CMU traces [93]. We define the Earliness of a speculative copy as:

$$\text{Earliness} = 1 - \frac{\text{StartTime}_c - \text{StartTime}_s}{\text{Median Execution Time}} \tag{6.1}$$

where StartTime_c denotes the starting moment of speculative copy, StartTime_s denotes the starting moment of straggler and Median Execution Time is the median task execution time. The value of this parameter indicates how early the speculative copy is launched, with respect to the starting moment of straggler. Besides, we consider the execution time of speculative copies. Only successful speculative copies are considered, as they successfully finish and we can compute their execution time. Figure 6.1 depicts the correlation between the earliness of speculative copies and their execution time. In this figure, the execution time of speculative copies is normalized by the median task execution time. We present the results on roughly 400 out of the total 1000 jobs.

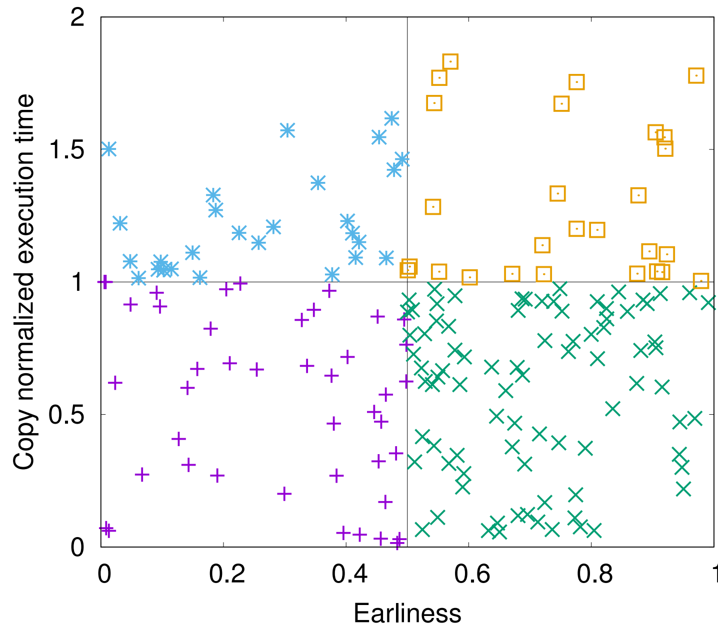


Figure 6.1 – Early speculative copies do not always result in higher performance for speculative copies.

On one hand, the *late* speculation handling leads to long execution time as shown by the blue stars in the top left corner of the figure. This can result in low performance improvement. Ren et al. [93] reported that many reduce tasks are speculated too late, resulting in wasteful speculative copies running for less than 10% of the tasks' normal execution time before being killed.

On the other hand, one may expect that *early* speculation handling should result in better straggler mitigation, as it can reduce the execution time of stragglers. Indeed, many studies aim to introduce early speculation handling solutions [6, 22, 117, 120, 121, 123]. However, this is not always true referring to the data plotted in orange squares in the top right corner of the figure. Some of these early copies, although launched right after the stragglers, have very bad performance. Actually, their execution time is at least 1.5x longer than the median task execution time. Thus, we conclude that existing speculative execution techniques with a **fixed solution** (either early or late) to the *when* question cannot always lead to good straggler mitigation results.

6.1.2 Where to Launch: Heterogeneity Has to be Considered

Concerning the question of *where* to launch the speculative copies, we study how different copy allocations differ regarding performance and energy efficiency.

Since the clusters are usually shared by multiple jobs concurrently, resource contention can lead to performance degradation when collocating tasks with similar critical resource requirement on the same node. For example, Yildiz et al. [125] have shown that resource contention can lead to up to 2.5x slowdown. We also observe from the CMU traces that the performance variation between different speculative copies within one job can be large.

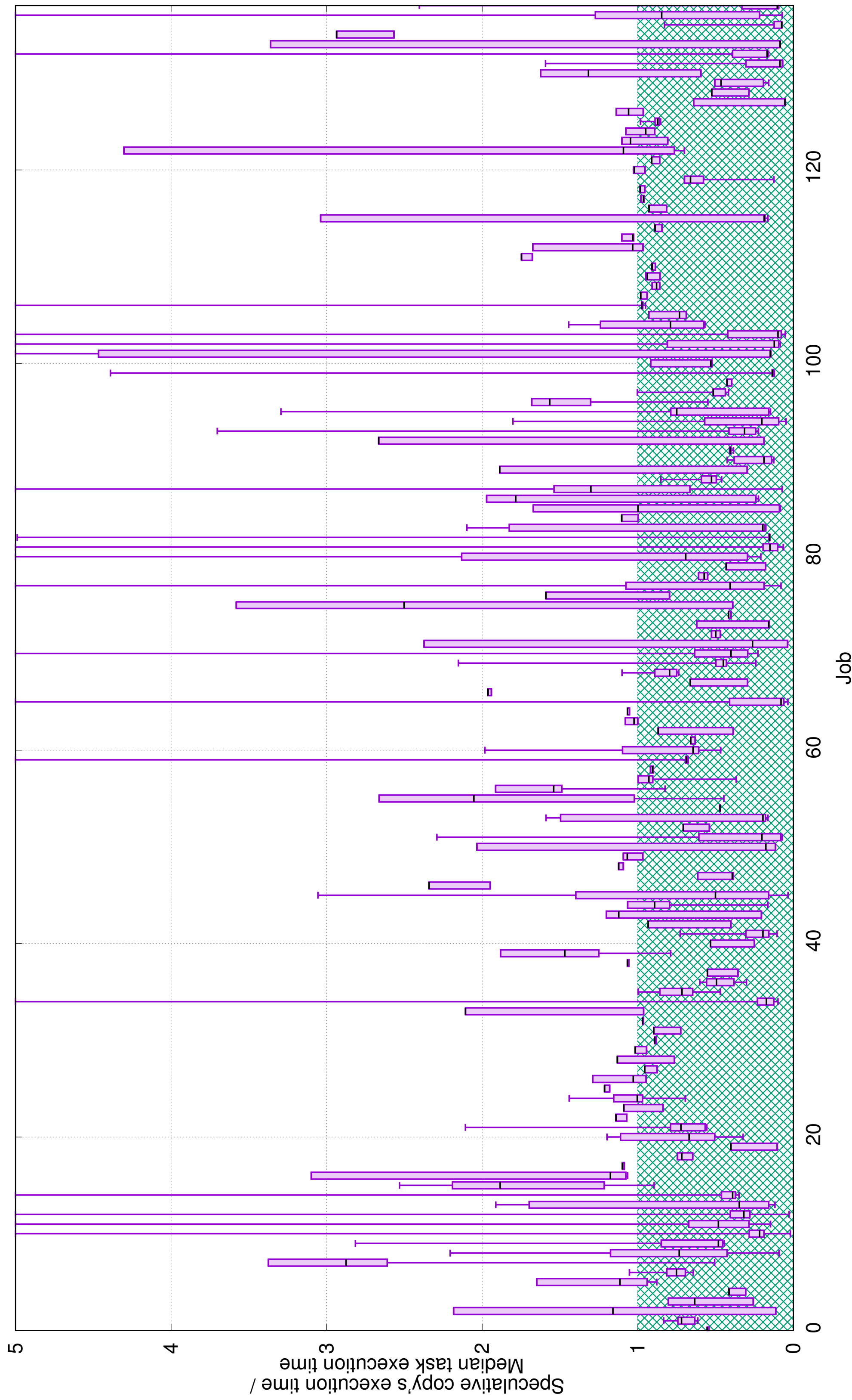


Figure 6.2 – Performance variability of speculative copies in the Hadoop production cluster at CMU.

Figure 6.2 displays a *box-and-whisker* diagram to show the performance variation of speculative copies in the CMU traces. We only consider Map tasks in this figure. For each job, we compute the ratios between copy execution time and the median task execution time. The *box-and-whisker* diagrams show the distribution (the minimum, 25th quantile, 50th quantile, 75th quantile and the maximum values) of these ratios per job. The execution time ratio between speculative copies and median task execution time is preferred to be less than 1 (the diagonally crossed zone on Figure 6.2). However, results demonstrate that some speculative copies take up to 5x longer compared to the median task execution time. Moreover, we find that the difference between the execution time of two copies within the same job can reach up to 50x.

Regarding energy consumption, we have observed that there is a trade-off between the performance and energy consumption for tasks executing on different nodes, according to the current status of the nodes. This has been discussed in Chapters 3 and 5. Allocating speculative copies to different locations, which may have different numbers of running tasks, can result in different performance and energy consumption results. Thus, it is important to take into consideration the impact of heterogeneity on performance and energy consumption when making speculative copy allocation decisions.

6.1.3 A Motivating Example

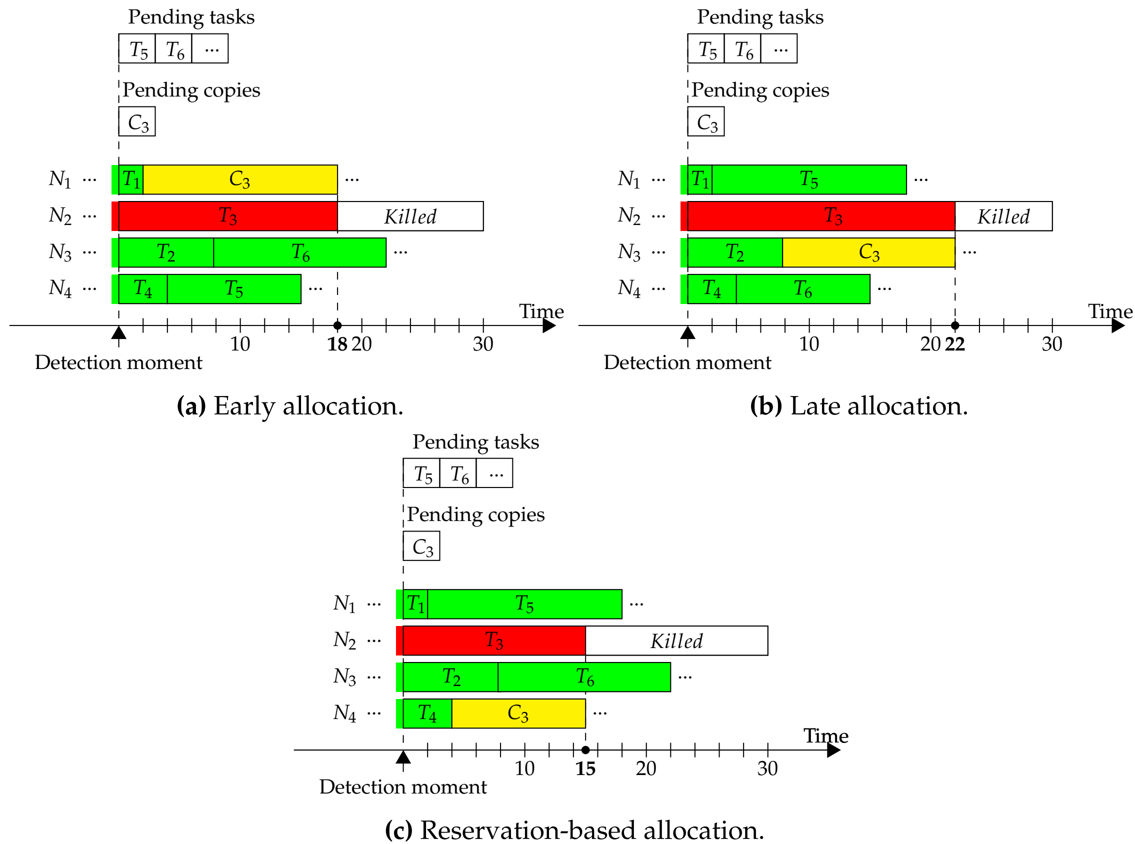
To better explain the two deficiencies mentioned above, we present an illustrative example on Figure 6.3. Let us consider a cluster with four nodes and one running job. The job consists of multiple tasks and the tasks perform differently on different nodes due to hardware heterogeneity. During runtime, we detect that T_3 is a straggler. Thus, we need to decide the right time and location to launch Copy C_3 to handle the straggler.

The Early copy allocation mechanism launches C_3 at the first freed slot in node N_1 (see Figure 6.3a). The execution time of C_3 on N_1 is 16. As a result, T_3 finishes at time 18. With Late copy allocation, C_3 is launched after all regular tasks in the queue (*i.e.*, T_5 and T_6) have been scheduled (see Figure 6.3b). Although the execution time of C_3 on N_3 is shorter than on N_1 (14 unit of times), the finish time of T_3 is later (*i.e.*, 22) due to the late start of the copy.

In comparison, we propose a reservation-based straggler handling mechanism, which adaptively decides both the start time and location of C_3 as shown in Figure 6.3c. The detailed design of this mechanism is introduced in below. The execution time of C_3 in this case is only 11 and the finish time of T_3 is 15.

From the perspective of resource consumption, the reservation-based mechanism also achieves the best result. The resource consumption (computed as number of slots multiplied by the execution time) of speculative executions with Early, Late and Reservation-based copy allocations are 72, 77 and 70, respectively. This example demonstrates that an adaptive and heterogeneity-aware speculative mechanism can achieve both better performance and energy consumption.

From the example, we conclude that the *when* and *where* questions are actually correlated with each other. We cannot make the best decision when considering them separately.



(a) Early allocation.

(b) Late allocation.

(c) Reservation-based allocation.

Figure 6.3 – An example with different speculative execution mechanisms.

6.2 Design Overview

Consider the scenario where multiple MapReduce jobs are running concurrently in a cluster with N nodes. Each node hosts multiple slots for task executions. Assume the job sizes are large and the cluster is highly utilized. Assume a fraction ϕ of the tasks in each job are stragglers. We study the straggler handling problem under this scenario, with the objective of optimizing both the performance and energy consumption of each job.

To achieve this goal, we propose a task-level speculative copy scheduler which works on top of the built-in schedulers of Hadoop, such as the Capacity and Fair schedulers. During job executions, the Hadoop built-in scheduler is responsible for selecting the next job to schedule in order to satisfy job-level requirements, such as fairness. Our task-level scheduler is responsible for launching regular tasks and speculative copies, in order to satisfy the performance and energy optimization objectives. Our scheduler has two main components, namely (1) the window-based resource reservation which periodically reserves resources to launch early speculative copies (Section 6.3.1) and (2) the resource selection component which decides on which resources to schedule speculative copies (Section 6.3.2). Figure 6.4 shows an overview of our design.

As the Hadoop built-in scheduler working at the job-level, we discuss the design of our scheduler using a single job for simplicity. However, it is important to note that our design works for both single job and multiple jobs scenarios.

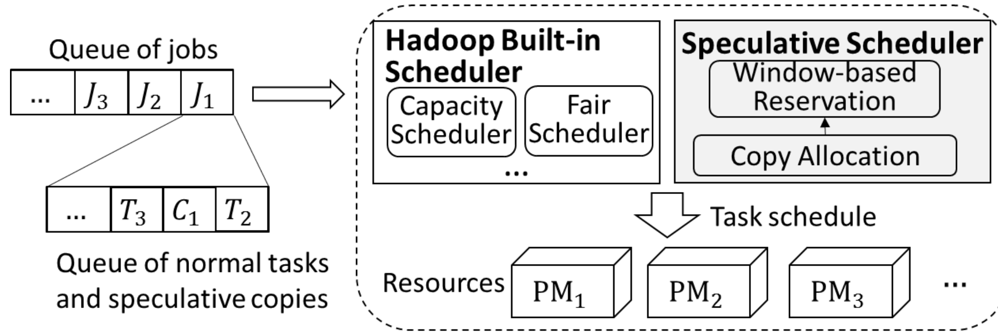


Figure 6.4 – Design overview of the reservation-based speculative execution mechansim.

6.3 Proposed Techniques

As discussed above, the speculative execution problem can be divided into two sub-problems: i) *when* to make reservations for launching speculative copies; and ii) *where* to launch the copies. In the following, we propose a window-based resource reservation technique and a heterogeneity-aware copy allocation technique to solve the when and where sub-problems, respectively.

6.3.1 Window-based Resource Reservation

Existing straggler handling mechanisms are triggered once stragglers are detected and slots are available for launching speculative copies. However, in highly utilized clusters, resources are not always available for launching speculative copies. Therefore, reservations have to be made in advance to make sure the speculative copies get executed. In this study, we propose a dynamic resource reservation technique, which considers the resources potentially free in a near future (*i.e.*, in a time window) to make efficient resource reservation plans at runtime.

Specifically, we define a time window size w, which is set to be shorter than the average task execution time. The average task execution time can be calculated using runtime information collected from the last time window or all previous windows. At the beginning of each time window, we perform straggler detection and store the detected stragglers in a set S. The set of regular tasks to be scheduled is denoted as T. We assume that the execution speed of a task remains constant within a time window and preemption is not allowed. Based on this assumption, we can estimate the finish time of the running tasks. Based on this information, we can estimate when slots are available. Let us denote the set of freed slots ordered by their starting time as V. Then, in each time window, the resource reservation problem is to reserve slots from V for speculative copies of stragglers in S, in order to optimize the overall performance and energy efficiency of the job.

Given $|V|$ free slots and $|S|$ detected stragglers, we can reserve from 0 (do not execute any speculative copy in the current time window) to $\min(|V|, |S|)$ slots (execute as many speculative copies as we can). Thus, the solution space to the resource reservation problem is $\sum_{n=0}^{\min(|V|,|S|)} P(|V|, n)$, where $P(|V|, n)$ stands for the permutation of selecting n slots from V. With the large number of stragglers in Big Data processing systems, the time for exhaustively

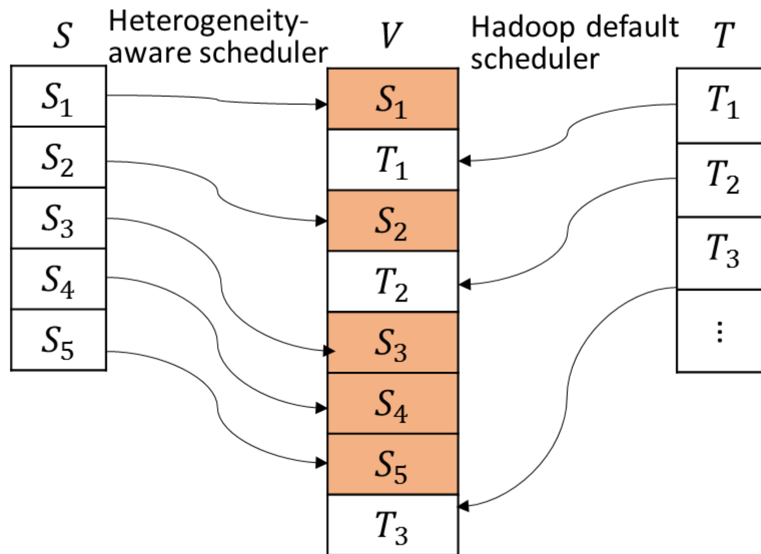


Figure 6.5 – An example of window-based resource reservation. In the current window, S is the set of detected stragglers ordered by criticalness, V is the set of available slots and T is the set of regular tasks to schedule.

searching the solution space is prohibitively long. Thus, we propose the following heuristic to quickly obtain a good solution to the problem.

- We first order the stragglers according to their *criticalness*. A critical straggler is a straggler which has better potential of saving resources. It is given higher priorities when being considered for copy allocations. This was discussed in detail in Chapter 5. Note that a speculative copy is only launched when the residual time of the straggler is longer than the expected execution time of the copy.

- Given the set of ordered stragglers, we sequentially search for a slot from V for each straggler in S to run the copies, as shown on the left side of Figure 6.5. When searching for a slot for each straggler, we consider the impact of different allocation choices to the performance and energy efficiency of the job. This searching is processed using our heterogeneity-aware copy allocation technique. The target of this searching is to bi-optimize the overall energy consumption and performance when launching speculative copies. As will be introduced in the next subsection, the time complexity of this technique is $O(|V|^{\min\{|S|,|V|\}})$.

- After making copy allocation decisions for all speculative copies, the non-reserved slots in V are used for executing the regular tasks in T. Regular tasks are scheduled using the default Hadoop built-in scheduler, as shown on the right side of Figure 6.5.

The benefits of our window-based design are two-fold. First, compared to real-time reservation methods, it provides the ability of looking-ahead when reserving resources and thus provides a larger space of choices for copy allocations. Second, compared to offline optimization methods, our resource reservation and copy allocation decisions are made periodically using the up to date runtime information from the latest time window, and thus are more likely to lead to better optimization results.

Discussion on window size. The reason why we constrain the window size to be shorter than the average task execution time is to perform one-step look-ahead when estimating the number of available slots. If the window size is too large, the $|V|$ and $|S|$ values are also likely to be large and thus the optimization overhead is large for one time window. Also, our assumption of constant task speed in one window is more reasonable with a small window size. However, if the window size is too small, the number of available slots in one time window is also small. As a result, space of choices is also small. Thus, the improvement, which is brought by our mechanism, can be reduced. In the evaluation section, we dedicate one part to illustrate the impact of different window sizes on the space of choices.

6.3.2 Heterogeneity-Aware Copy Allocation

Given a list of ordered stragglers S (S_i, where $i = 0, \cdots, k-1$) and a list of slots V (V_i, where $i = 0, \cdots, m-1$) to be freed in the current window, we need to decide which speculative copies of stragglers should be launched and where to launch them. In order to answer the *where* question, we first present a performance and an energy model to estimate the job performance and energy consumption yielded by different copy allocation solutions. Relying on those model-based estimations, we propose a copy allocation heuristic to smartly map speculative copies onto different slots.

Performance Model. The performance of a task (either a regular task or a speculative copy) on a slot can be affected by multiple factors. In this study, we mainly consider three factors, namely resource contention, data locality and capability of slots, which are usually the major impacting factors to task performance. We can divide the execution time of a task into the following three components:

$$t = t_{\text{req}} + t_{\text{cont}} + t_{\text{data}} \tag{6.2}$$

where t_{req} is the shortest time required to execute the task (with local data and no contention), t_{cont} is the execution time penalty of the task due to resource contention and t_{data} is the time spent on loading data from remote location if data locality is not achieved.

Given a mapping of straggler S_i to slot V_j, we estimate the performance of the speculative copy of S_i, denoted as C_i, using Equation 6.2. First, for Map tasks running on the same slot, t_{req} is almost the same, assuming that the input data is evenly assigned to different Map tasks. For Reduce tasks running on the same slot, we assume that the t_{req} depends linearly on the input size. Second, as the resource contention is mainly caused by concurrent tasks, the t_{cont} value of C_i can be considered as the same as that of the latest task running on V_j. Lastly, the t_{data} component can be easily estimated as $t_{\text{data}} = \frac{D}{B}$, where D is the size of the input data to be loaded by C_i and B is the network bandwidth between the location where the data resides and V_j.

With the above analysis, we can estimate the execution time of C_i using the runtime history of the latest non-straggling task (either a regular task or a speculative copy) executed on V_j, denoted as T_h. There are three cases for estimation.

- Case 1: if C_i and T_h execute either (1) with input data locally available or (2) without input data locally available, then we have $t_{C_i} = t_{T_h}$.

- Case 2: if C_i runs with data locally available and T_h does not, then we have $t_{C_i} = t_{T_h} - \frac{D_{T_h}}{B}$.
- Case 3: if T_h runs with data locally stored and C_i has to fetch input data from remote node, then we have $t_{C_i} = t_{T_h} + \frac{D_{C_i}}{B}$.

As has been discovered in existing studies [131], the ratio of local task over the number of total tasks is small in real clusters. For example, 58% of Facebook's jobs have only 5% tasks that are local tasks [131]. As a result, tasks most likely run with remote data. Thus, case 1 is the most common one in real clusters, which makes the performance estimation for copy allocations light-weight.

Assume that S_i is currently the first straggler in S that has not been handled. S_i runs on slot V_k. As stragglers are ordered according to their criticalness, we estimate that by handling S_i, the job execution time can be reduced by:

$$\Delta t_{ij} = t_{S_i} + t_{V_k} - t_{C_i} - t_{V_j} \tag{6.3}$$

where t_{S_i} and t_{C_i} are the estimated execution time of straggler S_i and its copy, respectively. t_{V_j} and t_{V_k} are the starting time of slot V_j and V_k, namely the starting time of C_i and S_i, respectively.

Energy Model. To estimate the energy consumption of executing C_i on V_j, we first propose the node-level power model. The power consumption of a running node is composed of two parts, namely the fixed static power consumption $P^{\text{total}}_{\text{static}}$ and the dynamic power consumption $P^{\text{total}}_{\text{dyn}}$. The dynamic power consumption is proportionally related to the usage of resources, including the number of active cores, the number of memory accesses and the amount of data transferred through the network [38]. Assume a node is equipped with a CPU of c cores, a memory with m GB capacity and a Network Interface Card (NIC) providing b Gb/s bandwidth. Then, $P^{\text{total}}_{\text{dyn}}$ can be defined as the sum of the dynamic power consumption of all active resources as shown below.

$$P^{\text{total}}_{\text{dyn}} = P^{\text{cpu}}_{\text{dyn}} + P^{\text{mem}}_{\text{dyn}} + P^{\text{net}}_{\text{dyn}} \tag{6.4}$$

The CPU dynamic power consumption of a running node mainly depends on the number of tasks simultaneously running on the node. We use P_{dyn} to denote the power consumption for one active core and n to denote the number of tasks running on the node. Thus, $P^{\text{cpu}}_{\text{dyn}}$ can be modeled as shown in Equation 6.5. This is the second term of the power consumption model mentioned in Chapter 4.

$$P^{\text{cpu}}_{\text{dyn}} = \begin{cases} n \times P_{\text{dyn}} & \text{for } 0 \leq n \leq c \\ c \times P_{\text{dyn}} & \text{for } c < n \end{cases} \tag{6.5}$$

The dynamic power consumption of the memory is strongly related to the number of data accesses per second [38]. It has been observed that this value can be estimated using the runtime history of tasks of the same job. The dynamic memory power consumption of a task is modeled as its average data accesses per second multiplied by the energy consumption of

one data access. $P_{\text{dyn}}^{\text{mem}}$ is modeled as the sum of the dynamic memory power of all running tasks [38].

$P_{\text{dyn}}^{\text{net}}$ can be modeled in a similar way, using the total amount of data transfer of each task [38]. Note that if a task achieves data locality, then the dynamic network power consumption of the task is zero.

Finally, the static power consumption is composed by the idle power consumption of CPU, memory and network resources [38]. We model $P_{\text{static}}^{\text{total}}$ as the sum of $P_{\text{static}}^{\text{cpu}}$, $P_{\text{static}}^{\text{mem}}$ and $P_{\text{static}}^{\text{net}}$, which are considered as architecture constants and can be estimated with hardware specifications or measured through profiling [38]. Thus, the total power consumption P of a running node can be modeled as the sum of $P_{\text{static}}^{\text{total}}$ and $P_{\text{dyn}}^{\text{total}}$. The power consumption of a running task can be computed as the difference of the node power consumption between after and before running the task.

The energy consumption E of a node is its power integrated over time and thus can be modeled as $E = \int_0^T P(t)dt$. We use T to denote the execution time of tasks running on the node. The energy efficiency EE is defined as the ratio of the performance to the energy consumption, where the performance is defined as the number of jobs finished per second.

Executing a new copy consumes more energy, whereas it saves at the same time energy by shortening the execution time of the straggler task. We can formulate the saved energy consumption caused by launching a copy of straggler S_i on slot V_j as follows.

$$\Delta E_{ij} = P_k \times \Delta t_{ij} - P_j \times t_{C_i} \tag{6.6}$$

where P_k and P_j are the power consumption of slots V_k and V_j which execute the straggler S_i and copy C_i, respectively.

Copy Allocation Heuristic. Given the performance and energy models, we define two metrics $(\Delta t_{ij}, \Delta E_{ij})$ to decide the fitness of each allocation of straggler S_i to slot V_j. Let us assume that there are m reserved slots in V. Given any straggler, there can be $m+1$ different copy allocation choices (*i.e.*, choosing one of the m slots to launch a copy and do not launch any copy). The objective of our copy allocation heuristic is to find the allocation decision that maximizes both the performance and energy metrics for each straggler in S.

Following the order of stragglers sorted in the window-based resource reservation step, we iteratively search all the slots in V and select a copy allocation with the highest value of $(\Delta t, \Delta E)$. When comparing two sets of metrics, *e.g.*, $M_1 = (\Delta t_1, \Delta E_1)$ and $M_2 = (\Delta t_2, \Delta E_2)$, we use the skyline comparison to decide which one is better. Specifically, we define $M_1 > M_2$ only when both $\Delta t_1 > \Delta t_2$ and $\Delta E_1 > \Delta E_2$ are satisfied. Algorithm 2 presents the general flow of our copy allocation heuristic. The worst-case complexity of this heuristic is $O(|V|^{\min\{|S|,|V|\}})$.

6.4 Methodology

In the following, we present the experimental methodology of our evaluation. We start with a detailed description of the discrete-event simulation. Then, we list the speculative execution mechanisms under study. Finally, we discuss the configurations that are used throughout the evaluation.

```
1  while S is not empty do
2  |  Straggler i is the head of S;
3  |  best_fitness = (0,0);
4  |  foreach Slot j in V do
5  |  |  calculate Δt_ij and ΔE_ij using Equations 6.3 and 6.6;
6  |  |  if (Δt_ij, ΔE_ij) > best_fitness then
7  |  |  |  best_slot = j;
8  |  |  |  best_fitness = (Δt_ij, ΔE_ij);
9  |  |  end
10 |  end
11 |  launch a copy of Straggler i in Slot best_slot;
12 |  remove Straggler i from S;
13 end
```

Algorithm 2: Copy allocation heuristic.

Discrete-event Simulation. We decide to adopt the simulation approach for evaluating our window-based resource reservation mechanism. The simulation approach allows us to dynamically tune diverse parameters (*e.g.*, hardware heterogeneity, straggler ratio, *etc.*) in order to comprehensively evaluate our solution under different scenarios. Regarding the simulator, the straggler mitigation techniques must be accurately simulated with this simulator. Besides, the simulator should be able to extract essential parameters from production traces for the simulation. Many of existing simulators are capable of simulating a large-scale Big Data processing system (*e.g.*, MRPerf [112], MRSG [65], SimMapReduce [102], MR-Sim [47]). However, none of them provides fully these functionalities, which are essential for our evaluation. Due to the absence of such a simulator, we decide to implement a new discrete-event simulator. It is important to notice that this simulator was designed to fully satisfy the aforementioned functionalities with minor implementation overhead. Additionally, it can be used to evaluate upcoming straggler mitigation solutions, which requires a comprehensive evaluation in large-scale simulation with different parameters.

We develop a our simulator in Java with 2000 lines of code. There are two essential components in the simulator, including nodes and jobs. Each node is configured to have a specific number of slots, which represents the number of tasks the node can run concurrently. A node is set with a specific execution time ratio. This execution time ratio determines how long a task takes to complete on this node. For instance, if a node is set with the execution time ratio of 2.0x, the tasks running on this node may take 2.0x to finish, compared to the average task execution time. By varying the value of this ratio across nodes, we can emulate hardware heterogeneity in terms of performance in typical Big Data processing clusters.

Moreover, a node is equipped with a given number of CPU cores, a given capacity of memory as well as network bandwidth. When it executes concurrently multiple tasks, these resources are distributed across the tasks evenly. For instance, a node executes two concurrent tasks, which heavily consume network to transfer data. Each task runs with 50% of maximum network bandwidth. As a result, these two tasks take 2 times longer to finish compared to the case when each of them runs separately. This feature emulates resource contention in Big Data processing systems. We discuss in detail this feature below.

The implementation of job object implementation simulates Big Data job in real systems.

A job contains a list of tasks. Moreover, it has specific parameters, which can be customized. For instance, we can tune the ratio of stragglers when running a job. These parameters can be manually set, or extracted from the production traces, *e.g.*, CMU traces [93]. For each job, its tasks have specific resource usage characteristics (*e.g.*, CPU, memory and network usages).

There are three important parameters in our simulation: the resource contention degree, the hardware heterogeneity degree and the straggler ratio. The resource contention degree parameter indicates how much resource contention can affect the job performance. For instance, two data-intensive applications may greatly suffer from network contention, which leads to a high performance slowdown. If nodes have a low network bandwidth and tasks transfer a large amount of data, this slowdown is increased. The hardware heterogeneity degree parameter indicates how differently the same task can perform on different nodes. The straggler ratio parameter indicates the percentage of the total tasks can be stragglers while running.

We use the traces [93] collected in January 2012 of a Hadoop production cluster from CMU to extract the basic parameters: job arrival rate, job sizes, task average execution time, straggler ratio, *etc*. This Hadoop production cluster consists of 64 nodes. Each node in the cluster has a 2.8 GHz two quad-core CPUs, 16 GB of memory, 10 Gb/s Ethernet network connection, and four 7200 RPM SATA disk drives. During the 30 days execution, more than 1500 jobs (1.5 million tasks) were submitted to the cluster by 78 different users.

Compared Straggler Handling Mechanisms. In our evaluation, our reservation-based speculative execution mechanism is denoted as *Window*. *noSpec* is used to denote the case when speculative execution mechanism is disabled. The state-of-the-art speculative execution mechanism, which is currently used in Hadoop, is denoted as *Default*. This mechanism allocates speculative copies at the end of the job execution, when all regular tasks have been launched.

In addition, we include in our evaluation another speculative copy allocation method, called *Hete-aware*. This mechanism also allocates speculative copies on available resources at the end of the job execution. However, it takes into account the impact on both the performance and energy consumption of different slots. It uses the same copy allocation heuristic as shown in Algorithm 2. Its goal is to bi-optimize the performance and energy consumption of speculative copies.

Configurations. We divide our evaluation into two parts. The first part is used to compare different speculative execution mechanisms. In this evaluation, we set the resource contention to 2.5x, straggler ratio to 0.2, hardware heterogeneity to 3.0x and window size to 0.2. The second part of the evaluation focuses on examining the impact different values of these parameters on the behaviors of our proposed speculative execution mechanism.

Metrics. The execution time metric is calculated for each job. It is determined by the amount of time from the moment the job's first task starts until its last task finishes. The energy consumption metric is the sum of energy consumption of a job's tasks and copies. The energy consumption of a task or copy is determined by its power consumption multiplied by it execution time. The detailed energy and power consumption models are discussed in Section 6.3.2.

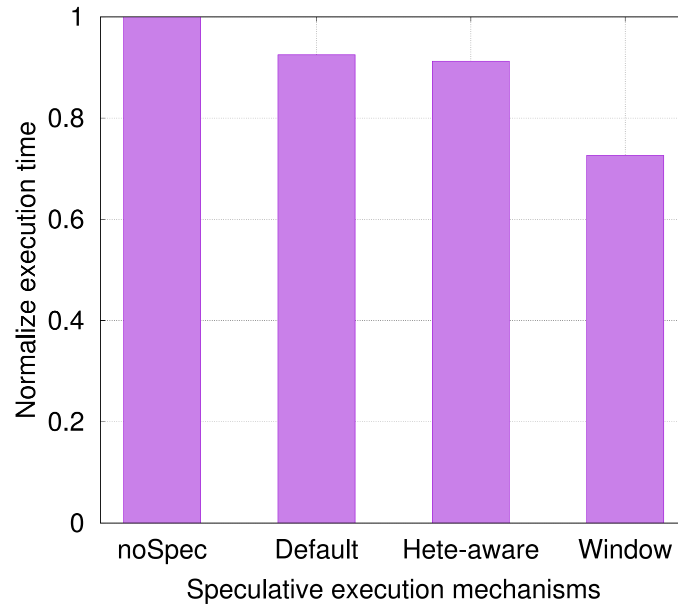


Figure 6.6 – Normalized execution times when using different speculative execution mechanisms.

In our experiment, when the results are normalized, they are normalized to those of *noSpec*.

6.5 Experimental Evaluation

In this section, we evaluate the effectiveness of our speculative execution mechanism by comparing it with state-of-the-art speculative execution mechanisms. Then, we perform sensitivity studies to evaluate the impact of various parameters on the effectiveness of our speculative execution mechanism.

6.5.1 Comparison of Different Speculative Execution Mechanisms

In this section, we start with a comparison in overall job performance and energy consumption. Then, we take a deeper look at the behaviors of different speculative execution mechanisms to better understand the results.

Overall Performance Comparison. Figure 6.6 shows the normalized average job execution times obtained with different speculative execution mechanisms. We have the following observations.

First, our speculative execution mechanism obtains the best results with respect to execution time. Specifically, *Window* reduces the average job execution time by 27.4%, 21.5% and 20.4% compared to *noSpec*, *Default* and *Hete-aware*, respectively. This is because early speculative copies can better mitigate stragglers. Moreover, reserving the appropriate resources provides speculative copies with better performance. On average, the execution times of

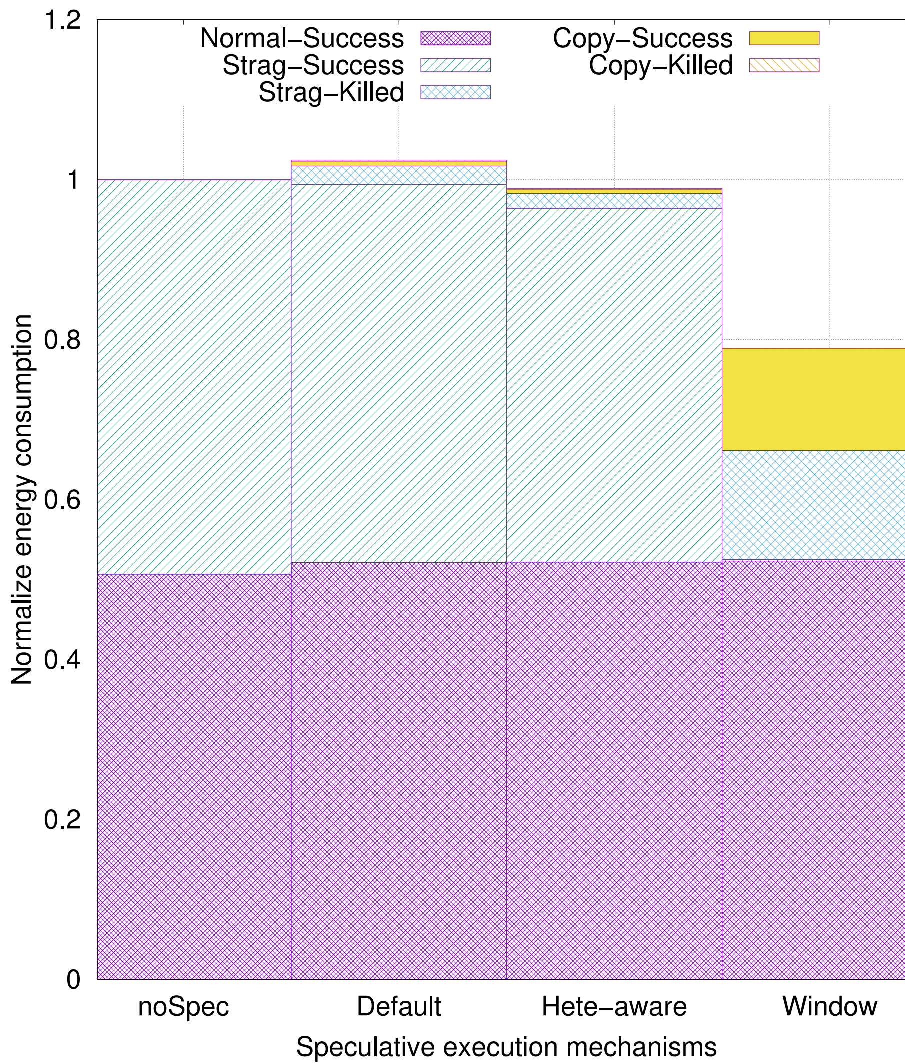


Figure 6.7 – Energy consumption breakdown with different speculative execution mechanisms.

speculative copies when using *Default* and *Hete-aware* are respectively 48% and 41% higher compared to *Window*. In the sensitivity evaluation, we discuss this in more detail.

Second, *Default* and *Hete-aware* can improve execution time. However, these improvements are modest in comparison with *Window*. Specifically, *Default* and *Hete-aware* can reduce the average execution time by 7.5% and 8.8% compared to *noSpec*, respectively. This is because both *Default* and *Hete-aware* launch speculative copies at the end of the execution. At this stage, stragglers have been running for a long time. Even if speculative copies are successful, the time reduction is significantly smaller. These results emphasize on the importance of early speculative execution in mitigating stragglers.

Third, we notice that the execution time reductions of *Default* and *Hete-aware* are very close. Specifically, the difference is only 1.3% between *Default* and *Hete-aware*. *Hete-aware* is designed targeting a lower energy consumption while maintaining performance comparable to *Default*.

Figure 6.8 – CDF of execution time improvements of different speculative execution mechanisms with respect to *noSpec*.

Energy Consumption Comparison. Figure 6.7 represents the average energy consumption when using different speculative execution mechanisms. Overall, we notice that *Window* can significantly reduce the energy consumption by 21.1%, 23.0% and 20.0% compared to *noSpec*, *Default* and *Hete-aware*, respectively.

To better understand this high reduction, we provide the detailed energy breakdown with different speculative execution mechanisms. There are five different task categories: (1) normal tasks which successfully finish (denoted as *Normal-Success*); (2) stragglers which successfully finish (denoted as *Strag-Success*); (3) stragglers which get killed (denoted as *Strag-Killed*); (4) successful speculative copies (denoted as *Copy-Success*); and (5) speculative copies which get killed (denoted as *Copy-Killed*).

We notice that *Window* significantly reduces the energy consumption of stragglers. These stragglers are shortened thanks to early speculative copies. However, the speculative copies consume extra energy and contribute to the total energy consumption. Nonetheless, the increase (*i.e.*, in energy consumed by successful speculative copies) are compensated by the large reduction in the energy consumption of stragglers. These results emphasize on the importance of early speculative copies allocation in reducing a large amount of energy consumed by stragglers.

Regarding *Hete-aware*, we notice that it has slightly better energy consumption compared to *Default*. The energy reduction is up to 5% in this case.

Performance and Energy Consumption at the Job-level. Figure 6.8 and 6.9 depict the CDF of the reductions in execution time and energy consumption at the job-level. First, we notice that *Window* still shows good improvement over the other mechanisms. For example, as shown in Figure 6.8, *Window* obtains over 30% reduction in job execution time for more than 90% of the jobs when compared to *noSpec*. Similar trend is recorded in the energy consumption comparison. *Window* can reduce more than 20% energy consumption in 80% of the jobs, compared to *noSpec*.

Second, *Default* and *Hete-aware* again have close improvements in both execution time and energy consumption. This explains their close results in both average execution time

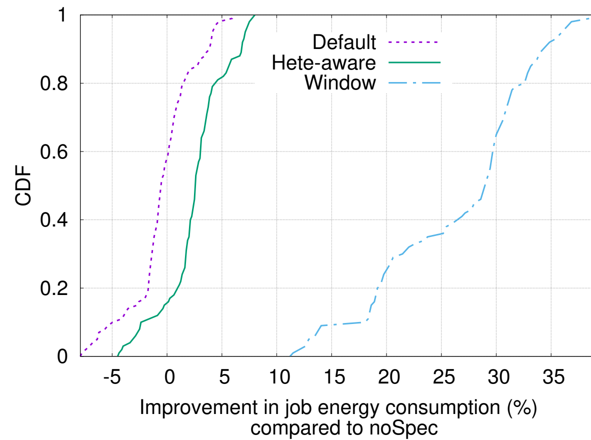


Figure 6.9 – CDF of energy consumption improvements of different speculative execution mechanisms with respect to *noSpec*.

and average energy consumption.

Third, we notice that *Default*, and sometimes *Hete-aware*, can have negative improvements, in both execution time and energy consumption for some jobs. This is due to their way of launching speculative copies at the end of the job execution. Late speculative copies may get killed as they cannot finish before the stragglers which have been running for a long time. Once the speculative execution is not successful for one job, the unsuccessful speculative copies occupy slots for long time and affect the execution of the next jobs. As a result, this negative impact may be amplified.

Energy Efficiency Comparison. Figure 6.10 represents the comparison in energy efficiency with different speculative execution mechanisms. The energy efficiency is calculated as throughput divided by energy consumption. The throughput is defined as the number of jobs finished per second. We observe that our speculative execution mechanism, *i.e.*, *Window*, results in significantly higher energy efficiency. Specifically, it has 75%, 70% and 61% higher energy efficiency compared to *noSpec*, *Default* and *Hete-aware*, respectively.

Regarding *Defaut*, late speculative execution only improves the energy efficiency by 5% compared to *noSpec*. With *Hete-aware*, thanks to the heterogeneity-aware copy allocation method, it has 9% higher energy efficiency compared to *Default*. In comparison with *noSpec*, it obtains an improvement of 14% in energy efficiency.

Conclude. We observe that dynamically reserving resources makes speculative execution more effective. This leads to significant performance improvement and energy consumption reduction. It results in a substantial increase in energy efficiency. In contrast, launching speculative copies at the end of the execution limits the potential benefits of speculative execution, as stragglers have been running and consuming energy for a long time. In a high-utilization cluster, dynamic resource reservation is the key to better answer both questions of *when* and *where* in order to achieve the maximum benefits.

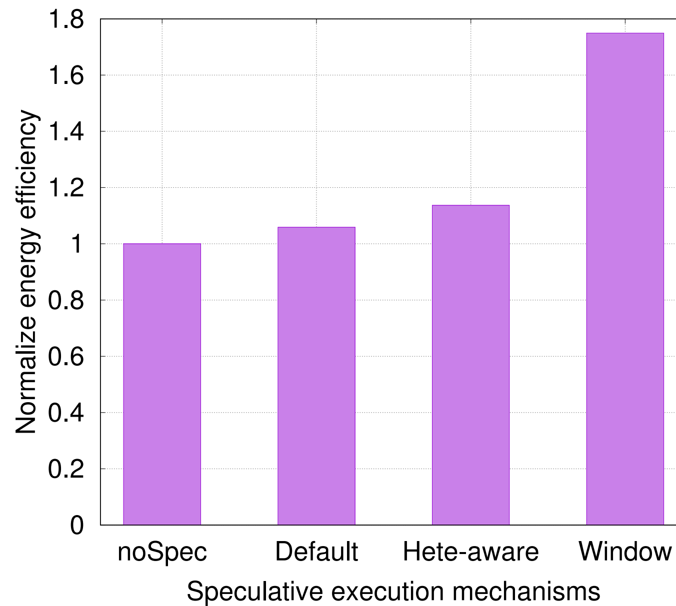


Figure 6.10 – Energy efficiency with different speculative execution mechanisms.

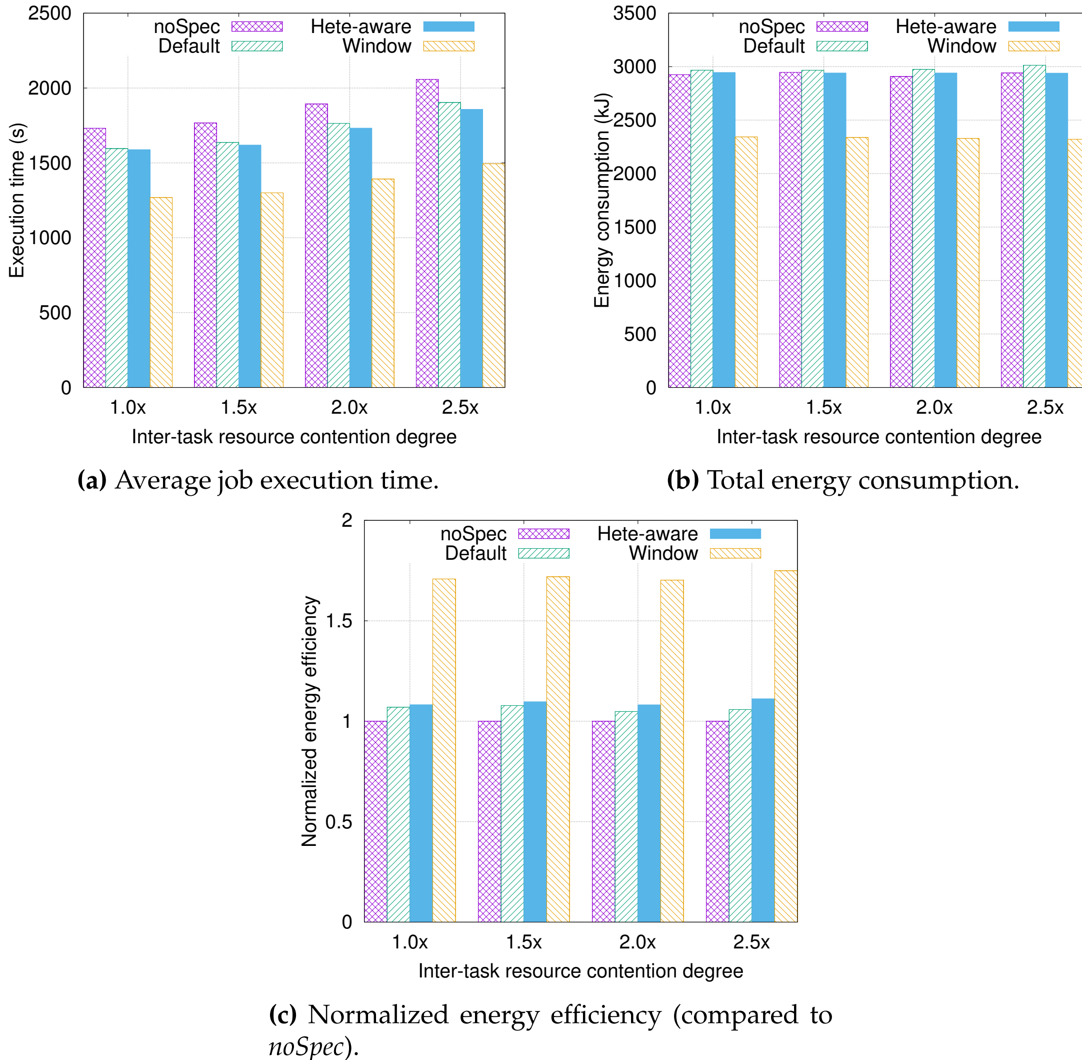
6.5.2 Sensitivity Study

In this subsection, we evaluate four speculative execution mechanisms in improving the performance and energy efficiency while tuning different parameters, including the resource contention, straggler ratio, hardware heterogeneity and window size. Those results can be used as guidance for users to select the best parameters which suit with their needs.

Resource Contention. We manually change the resource contention degree between different jobs to simulate the scenario of running different types of applications in the cluster. Specifically, we vary the contention degree from 1.0x (no contention), 1.5x, 2.0x to 2.5x (severe contention). For example, when the parameter is 1.5x, collocating different tasks can result in a maximum slowdown of 1.5x, compared to the task normal execution time. In Big Data processing systems, it is common for this contention to happen [6, 126]. Figure 6.11 shows the results of the four mechanisms under different contention degrees. We make the following observations.

First, when the contention degree increases, the average job execution time also increases for the four mechanisms. However, *Window* keeps outperforming the other mechanisms, and the performance improvement increases with the increase of the contention degree, from 27.4%, 21% and 16% compared to *noSpec*, *Default* and *Hete-aware*, respectively. Similar trend is observed for energy consumption. In all cases, *Window* performs the best among all compared algorithms, with 21%, 23.5% and 21% reduction compared to the total energy consumption of *noSpec*, *Default* and *Hete-aware*, respectively.

Second, we notice that *Hete-aware* performs better with respect to the increase of resource contention. When contention increases, the differences in performance and energy consumption also increase. As a result, the heterogeneity-aware copy allocation method of *Hete-aware* can benefit more from these increasing differences.



(a) Average job execution time.

(b) Total energy consumption.

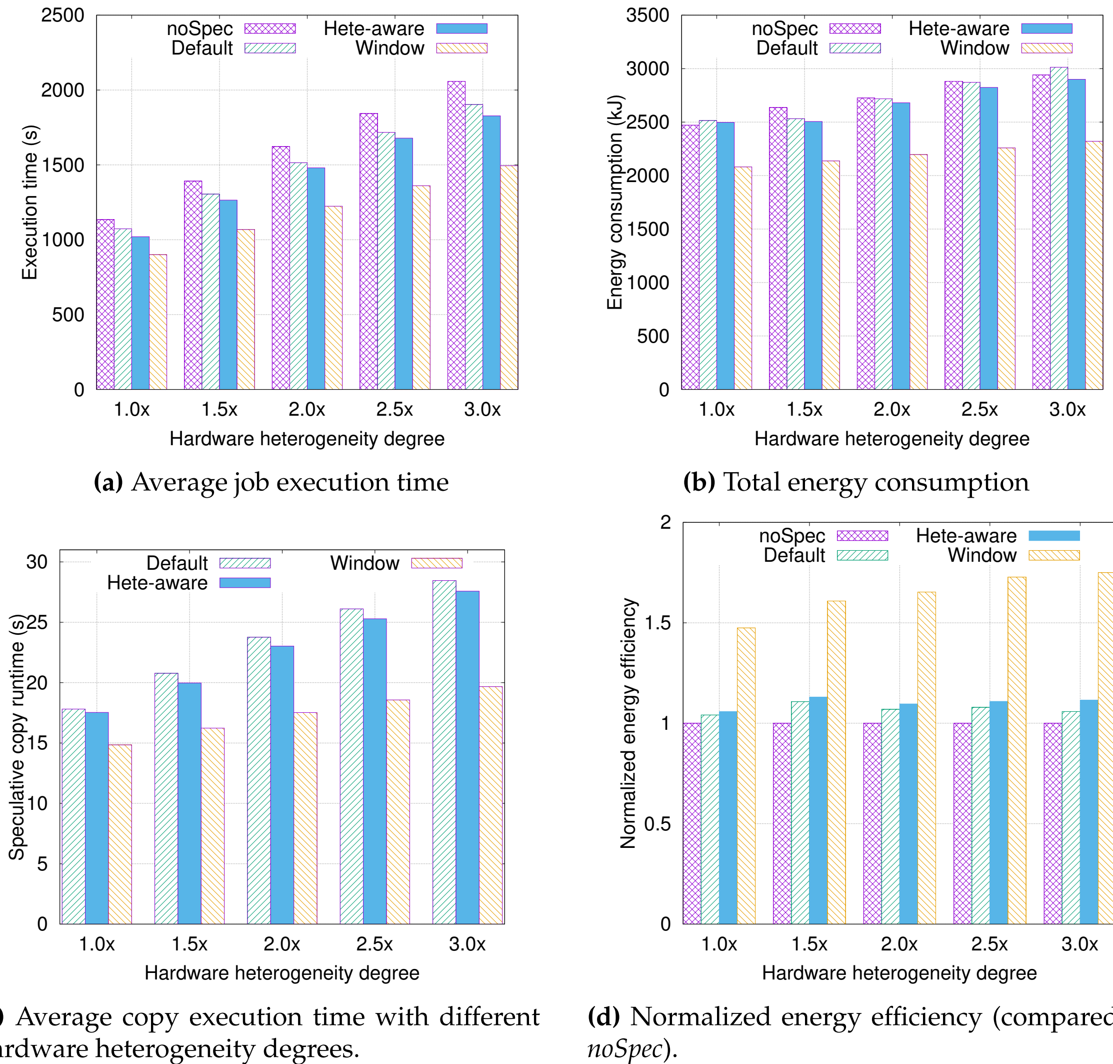
(c) Normalized energy efficiency (compared to *noSpec*).

Figure 6.11 – Sensitivity study on resource contention degree.

Regarding the energy efficiency, *Window* increases the overall energy efficiency by up to 75% compared to *noSpec* as shown in Figure 6.11c. In comparison with *Default* and *Hete-aware*, *Window* has up to 70% and 63% higher energy efficiency, respectively.

These results indicate that our proposed speculative execution mechanism works well with diverse resource contention degrees. Besides, *Hete-aware* results in higher improvement compared to *Default* along with the increase of resource contention degree.

Hardware Heterogeneity. Hardware heterogeneity is yet another major feature of Big Data processing systems. Nathuji *et al.* [83] record that 90% of Big Data processing systems upgrade their infrastructures within 2 years with new machines. This leads to an inevitable hardware heterogeneity. In this evaluation, we vary the hardware heterogeneity degree from 1.0x, 1.5x, 2.0x, 2.5x to 3.0x to simulate different types of commodity clusters. For example, when the parameter is 1.5x, the performance of the same task running on different nodes can



(a) Average job execution time

(b) Total energy consumption

(c) Average copy execution time with different hardware heterogeneity degrees.

(d) Normalized energy efficiency (compared to *noSpec*).

Figure 6.12 – Sensitivity study on the hardware heterogeneity degree.

vary 1.5 times of its normal execution time. Figure 6.12 shows the results of the compared mechanisms under different heterogeneity degrees.

As the hardware heterogeneity degree increases, the straggler problem becomes more important. The *Window* mechanism is able to obtain higher improvement in performance and energy compared to the other mechanisms. For example, *Window* reduces the average job execution time by 16%–22% and the total energy consumption by 17%–23% compared to *Default*. When taking a closer look at the execution of speculative copies (Figure 6.12c), we find that the average execution time of speculative copies increases much slower in *Window* than in *Default*, when the hardware heterogeneity degree increases. This is mainly due to a heterogeneity-aware resource reservation of *Window* to early launch speculative copies.

Hete-aware also performs better with the increase of hardware heterogeneity degree. However, we observe that this increase is small compared to *Window*. This is due to the small number of available slots when *Hete-aware* allocates copies. We observe that when

Hete-aware copy allocation method allocates speculative copies, the average number of free slots is less than 5. This small set of slots might not contain the slots with good performance and energy consumption. This limits the number of options for *Hete-aware*, and thus hinders the effectiveness of *Hete-aware*. In contrast, *Window* is not limited by this as it can dynamically reserve the relevant slots for speculative copies. As a result, the speculative copies are launched on the slots with high performance and low energy consumption.

With respect to the energy efficiency, *Window* again appears to have significantly high energy efficiency. It can increases the overall energy efficiency by up to 75% compared to *noSpec* as shown in Figure 6.12d. This increase in energy efficiency increases when the hardware heterogeneity degree increases. This result demonstrates that our speculative execution mechanism works even better with higher hardware heterogeneity degree. We observe the same trend with the *Hete-aware* speculative execution mechanism.

Straggler Ratio. In Big Data processing systems, the occurrence of stragglers is common. By analyzing the traces from CMU, we observe that the straggler ratio, *i.e.*, ratio of the number of stragglers over the number of total tasks, varies from 0.0 to 0.40 per job. In this evaluation, we vary the straggler ratio from 0.05 up to 0.40 to study the behavior of four speculative execution mechanisms.

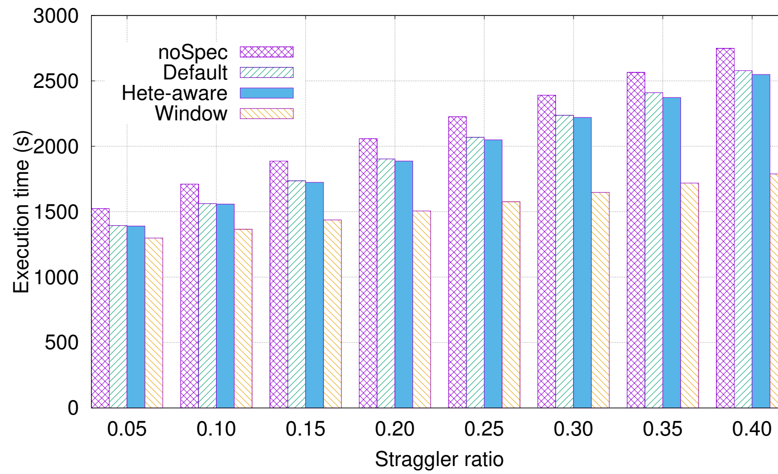
Figures 6.13 and 6.14 depict the results of these mechanisms. With the increase of straggler ratio, *Window* can reduce the average job execution time and total energy consumption more than the other mechanisms. For example, *Window* reduces the average execution time by 6.5%–30% and the energy consumption by 6%–34% compared to *Hete-aware*. This means that our algorithm can work better in systems with serious straggler problems. This is because *Window* provides timely resources to mitigate more stragglers. As a result, the higher the ratio of stragglers is, the better the improvement *Window* can achieve. Accordingly, *Window* can significantly increase the overall energy efficiency. As shown in Figure 6.14, the increase can be up to 140%, 135% and 130% compared to *noSpec*, *Default* and *Hete-aware*, respectively. More importantly, the increase in energy efficiency substantially improves with the increase of straggler ratio.

In contrast, the improvements when using *Hete-aware* or *Default* do not show significant increase when the straggler ratio increases. This is because both *Hete-aware* and *Default* mitigate stragglers at the end of job execution. A large number of stragglers, which run and finish before the final wave, are not mitigated by these mechanisms.

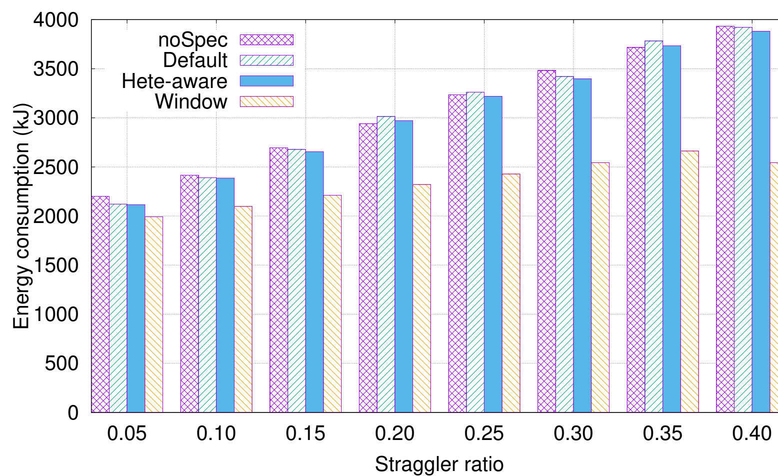
To conclude, *Window* can successfully mitigate a large number of stragglers in highly utilized cluster.

Window Size. In contrast with the aforementioned parameters, window size is a specific parameter of *Window*. This parameter allows users to define the amount of time between two consecutive resource reservations. This parameter is calculated per job. In this evaluation, the window size is computed as the ratio between the window time and the execution of the fastest task of the job. We vary it from 0.1 to 1.0 to study the impact on performance and energy efficiency.

We can observe on Figure 6.15 that a larger window size leads to a better performance of speculative copies. With the window size of 0.1, speculative copies take 16% longer time to finish compared to the results when window size is set at 1.0. This is because a large window



(a) Average job execution time



(b) Total energy consumption

Figure 6.13 – Sensitivity study on the straggler ratio: Comparison on execution time and energy consumption.

time makes the reservation look further into future. This increases the number of resources which will be available within the next window time. Accordingly, the reservation has more options to select when launching speculative copies.

However, the energy consumption does not show the same trend with the increase of the window size parameter. We observe that energy consumption only decreases when increasing the window size from 0.1 to 0.2. The energy reduction is 4% in this case. From that value of the window size, a larger window sizes results in a higher energy consumption. For instance, the energy consumption increases by 9% when increasing the window size from 0.2 to 1.0. This increase can be explained when we take a deeper look at the impact of window size on detection and handling latency.

Figure 6.16 depicts the average detection and handling latency when increasing the window size parameter. We observe that the latency increases linearly with the increase of win-

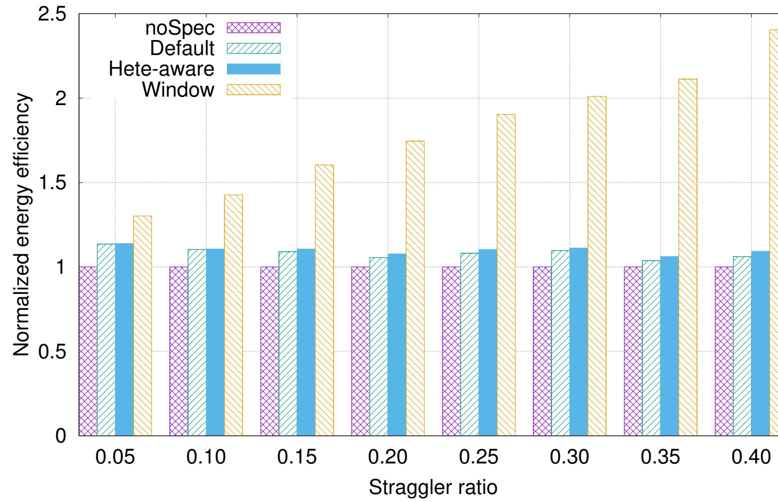
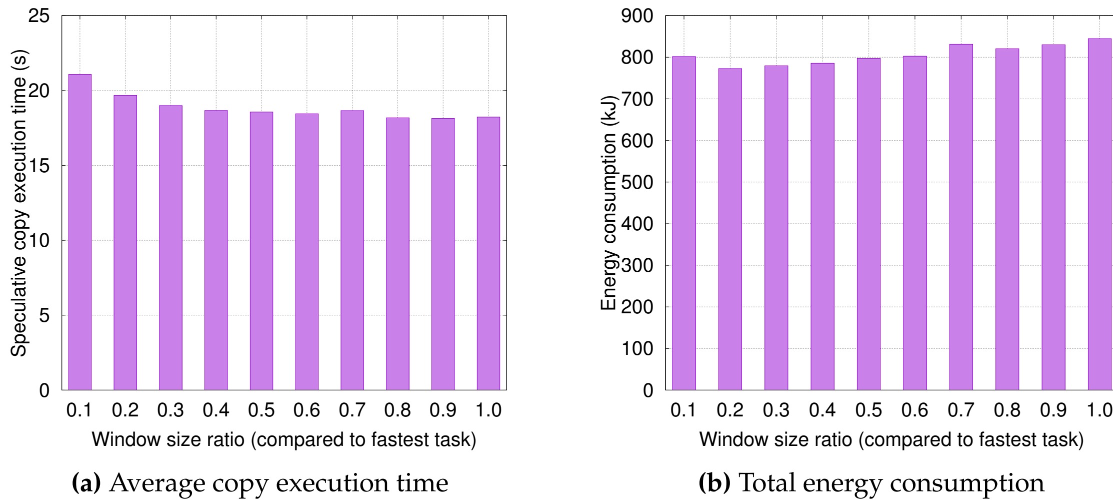


Figure 6.14 – Sensitivity study on the straggler ratio: Comparison on energy efficiency.



(a) Average copy execution time

(b) Total energy consumption

Figure 6.15 – Sensitivity study on the window size parameter.

dow size. The total latency increases 560% when increasing the window size from 0.1 to 1.0. This is because a larger window size decreases the frequency of handling straggler actions. For instance, a window size of 0.1 triggers the straggler detection and resource reservation 10 times more compared to a window size of 1.0. A larger window size further delays both straggler detection and the straggler handling. As a result, stragglers can run and consume energy for longer times before they are mitigated. In contrast, this larger window time can improve the performance of speculative copies of *Window*. This trade-off results in the lowest energy consumption at window size of 0.2 in our evaluation.

To conclude, the window size parameter can impact performance and energy consumption. A relevant value of window size can result in the best performance and energy consumption, depending on the characteristics of cluster and attributes of running applications.

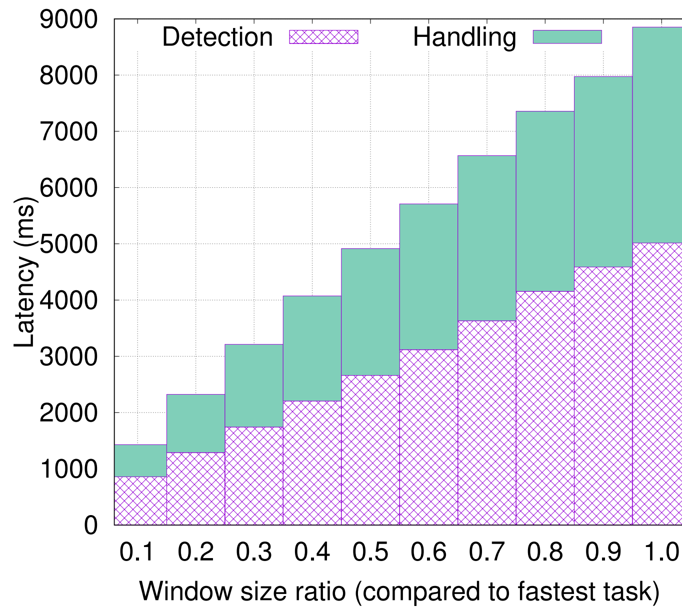


Figure 6.16 – Straggler detection and straggler handling latency of the window-based approach.

6.6 Conclusion

As having shown, how to answer the *when* and *where* questions strongly affects the performance and energy consumption. In this chapter, we aim to provide a speculative execution mechanism which can answer these two questions jointly. On the one hand, our speculative execution mechanism is equipped with a resource reservation mechanism. This mechanism searches for resources which are soon available. These resources are reserved for launching speculative copies as soon as they get freed. On the other hand, our speculative execution mechanism adopts a window-based scheduling technique. With this technique, the resource reservation is triggered at the beginning of each window time. Only the resources, which are expected to be freed within the current window time, are considered. The size of this window time can be dynamically changed to determine the straggler handling latency.

Through a detailed evaluation using discrete-event simulations, we observe that our speculative execution mechanism offers a significant performance increase and energy consumption reduction. As a result, it leads to a substantial improvement in energy efficiency. Moreover, we provide a detailed sensitivity study to illustrate the impacts of different parameters on how our speculative execution mechanisms and state-of-the-art mechanisms work.

Chapter 7

Conclusion

Contents

7.1 Achievements **116**
7.1.1 Characterizing the Impact of Straggler Mitigation on Performance and Energy Consumption 116
7.1.2 Measuring and Enabling Energy Efficiency of Straggler Detection . . 117
7.1.3 Bringing Energy-awareness to Straggler Handling 118
7.1.4 Energy-efficient Straggler Handling Mechanism 118
7.2 Perspectives **119**
7.2.1 Prospects Related to the *Hierarchical* Straggler Detection Mechanism 119
7.2.2 Prospects Related to Our Straggler Handling Mechanisms 120

ENERGY consumption has become the major concern in operating large-scale Big Data processing systems. Moreover, this concern is quickly becoming crucial as Big Data processing infrastructures are relentlessly expanding to cope with the ever-growing data size. In parallel, the increasingly large scale of Big Data infrastructures raises another emerging issue which cannot be ignored, namely performance variability. This performance variability is in turn responsible for a large number of stragglers, which have significantly longer execution time compared to the average task execution time. Stragglers can have a significantly negative impact on the performance and energy consumption of Big Data processing systems. In response, much attention has been paid to mitigate stragglers. Unfortunately, these studies do not pay enough attention on the additional energy consumption of their straggler mitigation techniques. As a result, it may lead to a high energy cost.

1. Current straggler detection mechanisms are equipped with simple algorithm in order to quickly detect stragglers at runtime. These simplistic straggler detection mechanisms may make inaccurate detection decisions. For instance, they can overly detect

normal tasks as stragglers. The speculative copies of the overly detected normal tasks are most likely to get killed. This results in a large amount of extra energy consumption.

2. Heterogeneity exists in large-scale Big Data processing systems. Actually, it is the original cause of performance variability. It strongly affects the executions of Big Data applications. In other words, executing the same task on different nodes lead to different performance and energy consumption results. The same phenomenon can be observed when allocating speculative copies to different locations. Unfortunately, existing straggler handling mechanisms do not pay enough attention to this. As a result, they may lead to a high energy consumption.

3. By default, regular tasks have higher priority compared to speculative copies. Consequently, speculative copies are left in the queue until all regular tasks are launched. In the context of Big Data processing systems, many Big Data jobs can consist of thousands of tasks. Thereby, Big Data infrastructures may have to execute them in multiple waves. In this case, speculative copies are not launched until the final wave. As a result, stragglers are left to be running for a long time, while consuming a large amount of wasteful energy. Existing straggler mitigation techniques do not take in to consideration the impact of this scenario. Thus, the improvement that they can bring is significantly reduced.

Targeting these issues, our work improves the energy efficiency of straggler mitigation through a number of contributions. Hereafter, we collectively describe these contributions.

7.1 Achievements

7.1.1 Characterizing the Impact of Straggler Mitigation on Performance and Energy Consumption

Much attention has been paid to mitigate stragglers in Big Data processing systems. To detect stragglers, straggler detection mechanisms are typically equipped with simple detection algorithm in order to quickly detect straggler at runtime. To handle stragglers, speculative execution is a widely-used technique. A copy of the detected straggler is launched with the expectation that it can finish earlier and reduce the long execution time of that straggler.

Even though benefits exist, launching speculative copies is not cost-free due to the extra energy consumed by them. Even worse, speculative copies may get killed if they cannot finish earlier than the stragglers. In this case, speculative copies result in a high energy cost while bringing no performance improvement. To conclude, speculative execution can strongly impact the performance and energy consumption of Big Data processing systems. Unfortunately, very few of existing work can provide a detailed understanding of this aspect. In response, we present an in-depth study to understand the impact of speculative execution on performance and energy consumption in Big Data processing systems.

Understanding the Impact of Speculative Execution on Performance and Energy Consumption of Hadoop Clusters. By means of experiments, we investigate the impact of

speculative execution on the performance and the energy consumption. A set of experiments are conducted on large-scale clusters with three representative Big Data applications. We consider three different clusters: (1) homogeneous cluster; (2) cluster with nodes having different CPU powers; and (3) cluster consisting of nodes with different network bandwidths. Three applications are executed with and without speculative execution enabled on these clusters. The results are then analyzed to provide more insights into the behaviors of speculative execution in different environments with different applications.

We observe that speculative execution can sometimes result in performance improvement, sometimes lead to performance degradation. In the case that speculative execution improves performance, it is due to a high number of successful speculative copies. This partially stems from the inaccurate detection of straggler detection mechanism in Hadoop. In terms of energy consumption, a performance improvement does not entail a proportional energy reduction. This is due to the extra energy consumed by speculative copies. In the case of performance degradation, the ratio of unsuccessful speculative copies is high. These copies result in a higher energy cost. Finally, we observe that there exists a fundamental trade-off between performance and energy consumption while allocating speculative copies to different nodes.

7.1.2 Measuring and Enabling Energy Efficiency of Straggler Detection

As we have shown, inaccurate detection leads to unnecessary speculative copies, which are most likely to get killed. These killed copies result in a high energy cost. In an attempt to improve the energy efficiency of straggler mitigation, we first of all address the straggler detection phase.

Dedicated Metrics to Characterize Straggler Detection Mechanisms. As a first stepping stone towards improving the energy efficiency of straggler detection, we introduce a set of dedicated metrics to characterize and evaluate straggler detection mechanisms. This set includes some well-known metrics, *i.e.*, *Precision* and *Recall*, as well as novel metrics, which we have introduced to specifically deal with stragglers: *Detection Latency*, *Undetected Time* and *Fake Positive*. In order to demonstrate the use of our metrics, we conduct a set of experiments on Grid'5000. Successively, we explain in detail the methods to calculate the metrics, as well as the important parameters which can be tuned by users to suit their scenarios. Finally, we introduce a mathematical intuition to link each metric to the resulting performance and energy consumption. Finally, our proposed metrics allow users to easily and effectively characterize and evaluate straggler detection mechanisms.

Evaluating State-of-the-art Straggler Detection Mechanisms. Using the proposed metrics, we conduct a set of experiments to evaluate two state-of-the-art straggler detection mechanisms, *i.e.*, *Default* and *LATE*. The goal of this evaluation is to provide more insights into these widely-used straggler detection mechanisms. This information is important to either (1) users who want to select the relevant straggler detection mechanisms for their systems, or (2) researchers who aim to improve these straggler detection mechanisms. Our evaluation results indicate that these two mechanisms could be significantly improved. Regarding *Default*, it has low *Precision*. In some cases, the *Precision* is only 12%. This means 88% of detected stragglers were not actually stragglers. Regarding *LATE*, although it has higher

Precision compared to *Default*, the values of *Precision* are still low in general. These results motivated us to introduce a new straggler detection mechanism towards a higher *Precision*.

Hierarchical Straggler Detection: A Green Straggler Detection Mechanisms. This straggler detection mechanism adopts a hierarchical approach to detect straggler. It works as a secondary straggler detection layer on the top of regular straggler detection mechanisms. Our straggler detection mechanism detects stragglers by considering them at the node-level. Specifically, only stragglers that run on slow nodes (*i.e.*, nodes with performances below a threshold, configurable by users) are finally kept in the detected straggler list. Using our characterizing metrics, *Hierarchical* is shown to have very high *Precision*. In many cases, the values of *Precision* are 100%. Correspondingly, *Hierarchical* can significantly reduce the number of killed copies by up to 100%. Thus, the wasteful energy consumed by these copies is also reduced. In brief, *Hierarchical* achieves a higher energy efficiency compared to state-of-the-art straggler detection mechanisms. This straggler detection mechanism is implemented in Java with roughly 2000 lines of code, in the Hadoop stable versions 1.2.1 and 2.7.3.

7.1.3 Bringing Energy-awareness to Straggler Handling

Once detected, stragglers are usually handled with speculative execution technique. As shown, allocating speculative copies to different locations results in different performance and energy consumption. By taking this into account, one can improve the overall energy consumption while handling stragglers.

Energy-aware Speculative Copy Allocation. By means of experiments, we demonstrate that different speculative copy allocations can result in considerably different outcomes. In some cases, two different copy allocations result in similar performance, but have a large difference in energy consumption, by up to 25%. Targeting this issue, we introduce a new straggler handling mechanism, which is equipped with an energy-aware copy allocation method. This copy allocation method takes into account the impact on both performance and energy consumption of different speculative copies allocations. Thus, it allocates speculative copies to the resources which result in less energy consumption. This speculative copy allocation method is evaluated with different Big Data applications on Grid'5000. The results indicate that it can greatly improve the energy efficiency while guaranteeing comparably good performance, compared to state-of-the-art speculative copy allocation methods. It is implemented in the Hadoop stable version 1.2.1 with more than 1500 line of Java code.

7.1.4 Energy-efficient Straggler Handling Mechanism

Task scheduling in general aims at answering two classical questions: *where* to allocate the task and *when* to launch the task. Handling straggler using speculative execution basically shares a similar story, which is how to answer two questions of *where* and *when* to launch speculative copies. Regarding the question of *when*, speculative execution technique is strongly driven by the resource availability. Without available resources, speculative copies cannot be launched and have to wait until the release of resources. This makes speculative copies to be launched late. A late speculative copy has less chance to successfully finish. Even if it can successfully finish, the improvement that it brings is smaller as the straggler

has been running for long time and consuming a large amount of energy. With respect to the question of *where*, the resource unavailability again limits the possibilities that speculative copy allocation method can have. Thus, it reduces the benefits brought by speculative execution.

Energy-efficient Straggler Handling Adopting Resource Reservation Approach. In the context of Big Data processing systems, it is usual that jobs include thousands of tasks [93]. These large jobs usually have to execute in multiple waves to be finished. By default, regular tasks have higher priority with respect to speculative copies. As a result, speculative copies are starved waiting for resources until the last regular task is launched. In this scenario, the resource unavailability makes the straggler handling problem more challenging. Tackling this problem, we introduce a novel straggler handling mechanism, which jointly answers *where* and *when* questions to improve energy efficiency. Specifically, this straggler handling mechanism is equipped with a performance model and an energy consumption model. The goal is to bi-optimize both performance and energy consumption. Therefore, these two models are used to indicate a suitable location for speculative copies to be launched, satisfying the performance and energy consumption bi-optimization goal. On the other hand, our straggler handling mechanism adopts a window-based approach to dynamically decide the best timing for launching speculative copies. At the beginning of each window time, the straggler handling mechanism looks for upcoming free resources within the window time. Accordingly, the best resources are reserved for the speculative copies. Once resources are freed, corresponding speculative copies are launched. Through a set of evaluations using trace-driven simulation, our straggler handling mechanism is shown to significantly improve the energy efficiency by up to 61%.

7.2 Perspectives

Our work opens a number of perspectives. In this section, we discuss in details the most promising ones. We separate these perspectives into two sections: (1) directions addressing straggler detection phase; and (2) potential contributions regarding our straggler handling mechanisms.

7.2.1 Prospects Related to the *Hierarchical* Straggler Detection Mechanism

In Big Data processing systems, there exists two major categories of jobs. On the one hand, interactive jobs consist of several up to tens of tasks [4]. These small interactive jobs have strict response time requirements. On the other hand, batch data analysis jobs typically consist of thousands of tasks [31]. These batch jobs have mostly loose response time requirements.

With the presence of stragglers, the execution time of interactive jobs may be significantly increased. This long execution time can violate the strict response time requirement. Consequently, this job category demands effective straggler mitigation, with respect to performance improvement. Considering long-running batch jobs, the pressure on early finishing is significantly smaller. With this job category, straggler mitigation should rather focus on reducing the energy consumption of extremely long-running stragglers.

Adaptive Straggler Detection Mechanism for Big Data Processing Systems. With *Hierarchical* straggler detection mechanism, the slow-node parameter β can be easily tuned. A high value of β equals to a loose detection criteria, which means more stragglers are detected. In contrast, a small β makes *Hierarchical* detect only stragglers which have significantly longer execution time, compared to the average task execution time. In a system that executes multiple jobs, *Hierarchical* should be used with relevant value of β, depending on the job characteristics. High values of β can be used when detecting stragglers on interactive jobs with strict response time requirements. For this setting, more speculative copies are launched to better improve the job performance. This may lead to extra energy consumed by unnecessary copies. However, as these interactive jobs are very small, the number of unnecessary copies is expected to be also small. On the other hand, small values of β are used when detecting stragglers of long-running batch jobs. Small β reduces the number of inaccurate detection. Thereby, it reduces the energy cost on unnecessary copies. The major challenge for this approach is how to adaptively determine the optimal value of β for each job, given its size and its response time requirement. Therefore, a model which accurately exposes the trade-off between energy consumption and performance is needed. Finally, it requires an algorithm to determine the value of β which leads to the lowest energy consumption while satisfying the job's response time requirement.

7.2.2 Prospects Related to Our Straggler Handling Mechanisms

In general, the questions of *When* and *Where* are key questions for task scheduling problem. In Chapter 5 and Chapter 6, we introduce new mechanisms to comprehensively answer these questions with respect to the speculative copy scheduling problem, towards a higher energy efficiency. These mechanisms can also be applied to answer similar scheduling problems in Big Data processing systems. Hereafter, we discuss two promising directions.

A Resource Reservation Approach for Failure Recovery in Big Data Processing Systems. Failure is an inherent feature while operating large-scale Big Data processing systems [35]. Therefore, Big Data processing frameworks are equipped with fault tolerance mechanisms [31]. Once a failed task is detected, a new instance of this task is launched on the earliest available resource [113]. On the one hand, resources are not always available. Thereby, this failed task may need to wait for long time for free resources [125, 127]. This can delay the failure recovery process. On the other hand, the earliest resource might not be the most appropriate to execute the failed task. As a result, this task can show low performance and high energy consumption. Our resource reservation mechanism can comprehensively solve this problem. With our mechanism, we can proactively reserve resources for early re-launching failed tasks. Our mechanism is aware of the performance and energy consumption of the resources and the availability of resources. Thus, it can select resources for early re-launching failed tasks, with high performance and low energy consumption.

Energy-efficient Speculative Execution for Approximate Applications. Approximation applications accept inaccurate output within a error-bound (*e.g.*, machine learning applications [71]). For this type of application, speculative execution is used not only to mitigate stragglers but also to increase the chance of achieving higher accurate output. A speculative copy and its straggler can be killed if the current job's output satisfies the error-bound. This

scenario happens if a speculative copy does not have short enough execution time. In this case, speculative copy results in extra energy consumption while having zero contribution to improving the output accuracy. This type of killed speculative copies further increases the killed ratio. This in turn results in lower energy efficiency of speculative execution. At this point, our speculative execution mechanism can be applied to this type of applications in order to improve energy efficiency. The major challenge to adopt our mechanism is that it requires a model to predict the output accuracy of a job. Using the information indicated by this model, the speculative execution mechanism knows whether speculative copies can finish before the job's output satisfies the error-bound or not. If yes, speculative copies can be launched to improve further the output accuracy. Otherwise, speculative copies should not be launched, as they most likely get killed.

Bibliography

[1] F. Ahmadand, S. Lee, M. Thottethodi, *et al.*, "PUMA: Purdue MapReduce benchmarks suite", Purdue University, Tech. Rep., 2012. [Online]. Available: http://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1438&context=ecetr/.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques*. Addison Wesley, 1986.

[3] H. Amur, J. Cipar, V. Gupta, *et al.*, "Robust and flexible power-proportional storage", in *ACM Symposium on Cloud Computing (SoCC '10)*, 2010, pp. 217–228.

[4] G. Ananthanarayanan, A. Ghodsi, S. Shenker, *et al.*, "Effective straggler mitigation: attack of the clones", in *USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, 2013, pp. 185–198.

[5] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, *et al.*, "GRASS: trimming stragglers in approximation analytics", in *USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, 2014, pp. 289–302.

[6] G. Ananthanarayanan, S. Kandula, A. Greenberg, *et al.*, "Reining in the outliers in MapReduce clusters using Mantri", in *USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, 2010, pp. 1–16.

[7] D. P. Anderson, J. Cobb, E. Korpela, *et al.*, "SETI@ home: an experiment in public-resource computing", *Communications of the ACM (CACM)*, vol. 45, no. 11, pp. 56–61, 2002.

[8] T. W. Anderson, T. W. Anderson, T. W. Anderson, *et al.*, *An introduction to multivariate statistical analysis*. Wiley, 1958, vol. 2.

[9] C. Anglano, J. Brevik, M. Canonico, *et al.*, "Fault-aware scheduling for bag-of-tasks applications on desktop grids", in *ACM/IEEE International Conference on Grid Computing (GRID '06)*, 2006, pp. 56–63.

[10] C. Anglano and M. Canonico, "Scheduling algorithms for multiple bag-of-task applications on desktop grids: a knowledge-free approach", in *IEEE International Parallel and Distributed Processing Symposium (IPDPS '08)*, IEEE, 2008, pp. 1–8.

[11] M. Armbrust, A. Fox, R. Griffith, *et al.*, "A view of cloud computing", *Communications of the ACM (CACM)*, vol. 53, no. 4, pp. 50–58, 2010.

[12] M. J. Atallah, R. Cole, and M. T. Goodrich, "Cascading divide-and-conquer: a technique for designing parallel algorithms", *SIAM Journal on Computing (SICOMP)*, vol. 18, no. 3, pp. 499–532, 1989.

[13] G. Aupy, Y. Robert, F. Vivien, *et al.*, "Checkpointing algorithms and fault prediction", *Journal of Parallel and Distributed Computing (JPDC)*, vol. 74, no. 2, pp. 2048–2064, 2014.

[14] A. T. Bates, *Technology, e-learning and distance education*. Routledge, 2005.

[15] F. Bonomi, R. Milito, J. Zhu, *et al.*, "Fog computing and its role in the Internet of Things", in *Workshop on Mobile Cloud Computing (MCC '12)*, 2012, pp. 13–16.

[16] T. Bray, J. Paoli, C. M. Sperberg-McQueen, *et al.*, "Extensible markup language (XML)", *World Wide Web Journal (WWW)*, vol. 2, no. 4, pp. 27–66, 1997.

[17] D. J. Brown and C. Reams, "Toward energy-efficient computing", *Communications of the ACM (CACM)*, vol. 53, no. 3, pp. 50–58, 2010.

[18] P. Buneman, S. Davidson, G. Hillebrand, *et al.*, "A query language and optimization techniques for unstructured data", in *ACM SIGMOD International Conference on Management of Data (SIGMOD '96)*, 1996, pp. 505–516.

[19] P. Carbone, A. Katsifodimos, S. Ewen, *et al.*, "Apache Flink: stream and batch processing in a single engine", *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, pp. 28–38, 2015.

[20] M. Cardosa, A. Singh, H. Pucha, *et al.*, "Exploiting spatio-temporal tradeoffs for energy-aware MapReduce in the cloud", in *IEEE International Conference on Cloud Computing (CLOUD '11)*, 2011, pp. 251–258.

[21] H. Chen, R. H. Chiang, and V. C. Storey, "Business intelligence and analytics: from big data to big impact", *Management Information Systems Quarterly (MISQ)*, vol. 36, no. 4, pp. 1165–1188, 2012.

[22] Q. Chen, C. Liu, and Z. Xiao, "Improving MapReduce performance using smart speculative execution strategy", *IEEE Transactions on Computers (TC)*, vol. 63, no. 4, pp. 29–42, 2014.

[23] Y. Chen, S. Alspaugh, D. Borthakur, *et al.*, "Energy efficiency for large-scale MapReduce workloads with significant interactive analysis", in *ACM European Conference on Computer Systems (EuroSys '12)*, 2012, pp. 43–56.

[24] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in Big Data systems: a cross-industry study of MapReduce workloads", *Proceedings of the VLDB Endowment (VLDB)*, vol. 5, no. 12, pp. 1802–1813, 2012.

[25] Y. Chen, A. Ganapathi, and R. H. Katz, "To compress or not to compress - compute vs. IO tradeoffs for MapReduce energy efficiency", in *ACM SIGCOMM Workshop on Green Networking (Green Networking '10)*, 2010, pp. 23–28.

[26] Y. Chen, L. Keys, and R. H. Katz, "Towards energy efficient MapReduce", EECS Department, University of California, Berkeley, Tech. Rep., 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-109.html.

[27] D. Cheng, C. Jiang, and X. Zhou, "Heterogeneity-aware workload placement and migration in distributed sustainable datacenters", in *IEEE International Parallel and Distributed Processing Symposium (IPDPS '14)*, 2014, pp. 307–316.

[28] V. K. Chippa, D. Mohapatra, A. Raghunathan, *et al.*, "Scalable effort hardware design: exploiting algorithmic resilience for energy efficiency", in *Annual Design Automation Conference (DAC '10)*, 2010, pp. 555–560.

[29] K. Choi, R. Soma, and M. Pedram, “Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 24, no. 1, pp. 18–28, 2005.

[30] E. F. Codd, “Relational database: a practical foundation for productivity”, *Communications of the ACM (CACM)*, vol. 25, no. 2, pp. 109–117, 1982.

[31] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters”, *Communications of the ACM (CACM)*, vol. 51, no. 1, pp. 107–113, 2008.

[32] G. DeCandia, D. Hastorun, M. Jampani, *et al.*, “Dynamo: Amazon’s highly available key-value store”, *ACM SIGOPS Operating Systems Review (OSR)*, vol. 41, no. 6, pp. 205–220, 2007.

[33] W. Deng, F. Liu, H. Jin, *et al.*, “Lifetime or energy: consolidating servers with reliability control in virtualized cloud datacenters”, in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom ’12)*, 2012, pp. 18–25.

[34] M. D. Dikaiakos, D. Katsaros, P. Mehra, *et al.*, “Cloud computing: distributed Internet computing for IT and scientific research”, *IEEE Internet Computing (IC)*, vol. 13, no. 5, pp. 10–13, 2009.

[35] F. Dinu and T. E. Ng, “Understanding the effects and implications of compute node related failures in Hadoop”, in *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC ’12)*, 2012, pp. 187–198.

[36] D. Florescu and D. Kossmann, “Rethinking cost and performance of database systems”, *ACM Sigmod Record*, vol. 38, no. 1, pp. 43–48, 2009.

[37] A. Gainaru, F. Cappello, and W. Kramer, “Taming of the shrew: modeling the normal and faulty behaviour of large-scale HPC systems”, in *IEEE International Parallel and Distributed Processing Symposium (IPDPS ’12)*, 2012, pp. 1168–1179.

[38] M. Gamell, I. Rodero, M. Parashar, *et al.*, “Exploring power behaviors and trade-offs of In-situ data analytics”, in *IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’13)*, 2013, pp. 1–12.

[39] P. Garraghan, X. Ouyang, P. Townend, *et al.*, “Timely long tail identification through agent based monitoring and analytics”, in *IEEE International Symposium on Real-Time Distributed Computing (ISORC ’15)*, 2015, pp. 19–26.

[40] L. Gillam and M. Zakarya, “Energy efficient computing, clusters, grids and clouds: a taxonomy and survey”, *Sustainable Computing: Informatics and Systems*, vol. 14, pp. 13–33, 2017.

[41] I. Gog, M. Schwarzkopf, A. Gleave, *et al.*, “Firmament: fast, centralized cluster scheduling at scale”, in *USENIX Symposium on Operating Systems Design and Implementation (OSDI ’16)*, 2016, pp. 99–115.

[42] Í. Goiri, K. Le, M. E. Haque, *et al.*, “GreenSlot: scheduling energy consumption in green datacenters”, in *IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’11)*, 2011, pp. 1–11.

[43] I. Goiri, K. Le, T. D. Nguyen, *et al.*, “GreenHadoop: leveraging green energy in data-processing frameworks”, in *ACM European Conference on Computer Systems (EuroSys ’12)*, 2012, pp. 57–70.

[44] T. Gunarathne, T.-L. Wu, J. Qiu, *et al.*, "MapReduce in the clouds for science", in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom '10)*, 2010, pp. 565–572.

[45] *The Apache Hadoop project*. [Online]. Available: https://hadoop.apache.org/ (visited on 2017-06-25).

[46] J. Hamilton, *Cost of power in large-scale data centers*, 2015. [Online]. Available: http://perspectives.mvdirona.com/2008/11/cost-of-power-in-large-scale-data-centers/.

[47] S. Hammoud, M. Li, Y. Liu, *et al.*, "MRSim: a discrete event based MapReduce simulator", in *International Conference on Fuzzy Systems and Knowledge Discovery (FSKD '10)*, vol. 6, 2010, pp. 2993–2997.

[48] *HDFS architecture guide*. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html (visited on 2017-07-15).

[49] A. Holmes, *Hadoop in practice*. Manning Publications, 2012.

[50] D. Huang, X. Shi, S. Ibrahim, *et al.*, "MR-scope: a real-time tracing tool for MapReduce", in *ACM International Symposium on High Performance Distributed Computing (HPDC '10)*, 2010, pp. 849–855.

[51] S. Ibrahim, B. He, and H. Jin, "Towards pay-as-you-consume cloud computing", in *IEEE International Conference on Services Computing (SCC '11)*, 2011, pp. 370–377.

[52] S. Ibrahim, H. Jin, L. Lu, *et al.*, "Handling partitioning skew in MapReduce using LEEN", *Peer-to-Peer Networking and Applications*, vol. 6, no. 4, pp. 409–424, 2013.

[53] S. Ibrahim, H. Jin, L. Lu, *et al.*, "LEEN: locality/fairness-aware key partitioning for MapReduce in the cloud", in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom '10)*, 2010, pp. 17–24.

[54] S. Ibrahim, H. Jin, L. Lu, *et al.*, "Maestro: replica-aware map scheduling for MapReduce", in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '12)*, 2012, pp. 435–442.

[55] S. Ibrahim, D. Moise, H.-E. Chihoub, *et al.*, "Towards efficient power management in MapReduce: investigation of CPU-frequencies scaling on power efficiency in Hadoop", in *Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC '14)*, 2014, pp. 147–164.

[56] S. Ibrahim, T.-D. Phan, A. Carpen-Amarie, *et al.*, "Governing energy consumption in Hadoop through CPU frequency scaling: an analysis", *Future Generation Computer Systems (FGCS)*, vol. 54, no. C, 2016.

[57] M. Isard, M. Budiu, Y. Yu, *et al.*, "Dryad: distributed data-parallel programs from sequential building blocks", in *ACM European Conference on Computer Systems (EuroSys '07)*, 2007, pp. 59–72.

[58] D. Jeffrey, "Large-scale distributed systems at Google: current systems and future directions", in *ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS '09)*, 2009.

[59] Y. Jégou, S. Lantéri, J. Leduc, *et al.*, "Grid'5000: a large scale and highly reconfigurable experimental grid testbed", *International Journal of High Performance Computing Applications (IJHPCA)*, vol. 20, no. 4, pp. 481–494, 2006.

[60] H. Jin, S. Ibrahim, T. Bell, *et al.*, "Cloud types and services", in *Handbook of Cloud Computing*. Springer, 2010, ch. 14, pp. 335–355.

[61] H. Jin, S. Ibrahim, T. Bell, *et al.*, "Tools and technologies for building clouds", in *Cloud Computing*. Springer, 2010, ch. 1, pp. 3–20.

[62] H. Jin, S. Ibrahim, L. Qi, *et al.*, "The MapReduce programming model and implementations", in *Cloud Computing: Principles and Paradigms*. Wiley, 2011, ch. 14, pp. 373–390.

[63] R. T. Kaushik and M. Bhandarkar, "GreenHDFS: towards an energy-conserving, storage-efficient, hybrid Hadoop compute cluster", in *USENIX International Conference on Power Aware Computing and Systems (HotPower '10)*, 2010, pp. 1–9.

[64] J. Kim, J. Chou, and D. Rotem, "Energy proportionality and performance in data parallel computing clusters", in *International Conference on Scientific and Statistical Database Management (SSDBM '11)*, 2011, pp. 414–431.

[65] W. Kolberg, P. D. B. Marcos, J. C. Anjos, *et al.*, "MRSG–a MapReduce simulator over SimGrid", *Parallel Computing*, vol. 39, no. 4, pp. 233–244, 2013.

[66] D. Kondo, A. A. Chien, and H. Casanova, "Resource management for rapid application turnaround on enterprise desktop grids", in *ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis (SC '04)*, 2004, pp. 17–30.

[67] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems", *IEEE Transactions on Software Engineering (TSE)*, no. 1, pp. 23–31, 1987.

[68] M. Kryczka, R. Cuevas, C. Guerrero, *et al.*, "A first step towards user assisted online social networks", in *ACM Workshop on Social Network Systems (SNS '10)*, 2010, pp. 1–6.

[69] D. Laney, *3D data management: controlling data Volume, Velocity and Variety*, 2001. [Online]. Available: https://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf.

[70] W. Lang and J. M. Patel, "Energy management for MapReduce clusters", *Proceedings of the VLDB Endowment (VLDB)*, vol. 3, no. 1-2, pp. 129–139, 2010.

[71] P. Langley and H. A. Simon, "Applications of machine learning and rule induction", *Communications of the ACM (CACM)*, vol. 38, no. 11, pp. 54–64, 1995.

[72] G. Lee, B.-G. Chun, and H. Katz, "Heterogeneity-aware resource allocation and scheduling in the cloud", in *USENIX Conference on Hot Topics in Cloud Computing (HotCloud '11)*, 2011, pp. 1–5.

[73] L. Lei, T. Wo, and C. Hu, "CREST: towards fast speculation of straggler tasks in MapReduce", in *IEEE International Conference on e-Business Engineering (ICEBE '11)*, 2011, pp. 311–316.

[74] J. Leverich and C. Kozyrakis, "On the energy (in)efficiency of Hadoop clusters", *ACM SIGOPS Operating Systems Review (OSR)*, vol. 44, no. 1, pp. 61–65, 2010.

[75] Y. Li, Q. Yang, S. Lai, *et al.*, "A new speculative execution algorithm based on C4.5 decision tree for Hadoop", in *International Conference of Young Computer Scientists, Engineers and Educators (ICYCSEE '15)*, 2015, pp. 284–291.

[76] L. Lu, H. Jin, X. Shi, *et al.*, "Assessing MapReduce for internet computing: a comparison of Hadoop and BitDew-MapReduce", in *ACM/IEEE International Conference on Grid Computing (GRID '12)*, 2012, pp. 76–84.

[77] J.-Z. Luo, J.-H. Jin, A.-B. Song, *et al.*, "Cloud computing: architecture and key technologies", *Journal of China Institute of Communications*, vol. 32, no. 7, pp. 3–21, 2011.

[78] A. Malik, A. Malik, K. Hiekkanen, *et al.*, "Impact of privacy, trust and user activity on intentions to share Facebook photos", *Journal of Information, Communication and Ethics in Society*, vol. 14, no. 4, pp. 364–382, 2016.

[79] *MapReduce tutorial*. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html (visited on 2017-06-30).

[80] L. Mashayekhy, M. M. Nejad, D. Grosu, *et al.*, "Energy-aware scheduling of MapReduce jobs", in *IEEE International Congress on Big Data (BigData Congress '14)*, 2014, pp. 32–39.

[81] L. Mashayekhy, M. M. Nejad, D. Grosu, *et al.*, "Energy-aware scheduling of MapReduce jobs for Big Data applications", *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 26, no. 10, pp. 2720–2733, 2015.

[82] O. A. Mukhanov, "Energy-efficient single flux quantum technology", *IEEE Transactions on Applied Superconductivity (TAS)*, vol. 21, no. 3, pp. 760–769, 2011.

[83] R. Nathuji, C. Isci, and E. Gorbatov, "Exploiting platform heterogeneity for power efficient data centers", in *IEEE International Conference on Autonomic Computing (ICAC '07)*, 2007, pp. 1–10.

[84] M. Odersky, P. Altherr, V. Cremet, *et al.*, "An overview of the Scala programming language", École Polytechnique Fédérale de Lausanne, Tech. Rep., 2004. [Online]. Available: https://infoscience.epfl.ch/record/52656/files/ScalaOverview.pdf.

[85] A.-C. Orgerie, M. D. d. Assuncao, and L. Lefevre, "A survey on techniques for improving the energy efficiency of large-scale distributed systems", *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, pp. 1–31, 2014.

[86] X. Ouyang, P. Garraghan, D. McKee, *et al.*, "Straggler detection in parallel computing systems through dynamic threshold calculation", in *IEEE International Conference on Advanced Information Networking and Applications (AINA '16)*, 2016, pp. 414–421.

[87] R. Patgiri and A. Ahmed, "Big Data: the V's of the game changer paradigm", in *IEEE International Conference on High Performance Computing and Communications (HPCC '16)*, 2016, pp. 17–24.

[88] P. Pawluk, B. Simmons, M. Smit, *et al.*, "Introducing STRATOS: a cloud broker service", in *IEEE International Conference on Cloud Computing (CLOUD '12)*, 2012, pp. 891–898.

[89] T.-D. Phan, S. Ibrahim, G. Antoniu, *et al.*, "On understanding the energy impact of speculative execution in Hadoop", in *IEEE International Conference on Data Science and Data Intensive Systems (DSDIS '15)*, 2015, pp. 396–403.

[90] T.-D. Phan, S. Ibrahim, A. Zhou, *et al.*, "Energy-driven straggler mitigation in MapReduce", in *International European Conference on Parallel and Distributed Computing (Euro-Par '17)*, 2017, pp. 385–398.

[91] *Powered by Hadoop*. [Online]. Available: http://wiki.apache.org/hadoop/PoweredBy/ (visited on 2017-06-18).

[92] A. Qureshi, "Power-demand routing in massive geo-distributed systems", PhD thesis, Massachusetts Institute of Technology, 2010.

[93] K. Ren, Y. Kwon, M. Balazinska, *et al.*, "Hadoop's adolescence: an analysis of Hadoop usage in scientific workloads", *Proceedings of the VLDB Endowment (VLDB)*, vol. 6, no. 10, pp. 853–864, 2013.

[94] S. I. Resnick, *Heavy-tail phenomena: Probabilistic and statistical modeling*. Springer, 2007.

[95] P. Roy, R. Ray, C. Wang, *et al.*, "ASAC: automatic sensitivity analysis for approximate computing", in *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '14)*, 2014, pp. 95–104.

[96] P. Russom, *Big Data analytics*, 2011. [Online]. Available: https://vivomente.com/wp-content/uploads/2016/04/big-data-analytics-white-paper.pdf.

[97] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance", *Proceedings of the VLDB Endowment (VLDB)*, vol. 3, no. 1-2, pp. 460–471, 2010.

[98] M. C. Schatz, "CloudBurst: highly sensitive read mapping with MapReduce", *Bioinformatics*, vol. 25, no. 11, pp. 1363–1369, 2009.

[99] L. Shen, T. Abdelzaher, and M. Yuan, "TAPA: temperature aware power allocation in data center with MapReduce", in *IEEE International Green Computing Conference and Workshops (IGCC '11)*, 2011, pp. 1–8.

[100] M. Silberstein, A. Sharov, D. Geiger, *et al.*, "Gridbot: execution of bags of tasks in multiple grids", in *ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, 2009, 11:1–11:12.

[101] Z. D. Stephens, S. Y. Lee, F. Faghri, *et al.*, "Big Data: astronomical or genomical?", *PLOS Biology*, vol. 13, no. 7, pp. 1–11, 2015.

[102] F. Teng, L. Yu, and F. Magoulès, "SimMapReduce: a simulator for modeling MapReduce framework", in *International Conference on Multimedia and Ubiquitous Engineering (MUE '11)*, 2011, pp. 277–282.

[103] E. Thereska, A. Donnelly, and D. Narayanan, "Sierra: practical power-proportionality for data center storage", in *ACM European Conference on Computer Systems (EuroSys '11)*, 2011, pp. 169–182.

[104] P. Thinakaran, J. R. Gunasekaran, B. Sharma, *et al.*, "Phoenix: a constraint-aware scheduler for heterogeneous datacenters", in *IEEE International Conference on Distributed Computing Systems (ICDCS '17)*, 2017, pp. 977–987.

[105] A. Thusoo, Z. Shao, S. Anthony, *et al.*, "Data warehousing and analytics infrastructure at Facebook", in *ACM International Conference on Management of Data (SIGMOD '10)*, 2010, pp. 1013–1020.

[106] A. Toshniwal, S. Taneja, A. Shukla, *et al.*, "Storm@ Twitter", in *ACM International Conference on Management of Data (SIGMOD '14)*, 2014, pp. 147–156.

[107] N. Vasić, M. Barisits, V. Salzgeber, *et al.*, "Making cluster applications energy-aware", in *ACM Workshop on Automated Control for Datacenters and Clouds (ACDC '09)*, 2009, pp. 37–42.

[108] V. K. Vavilapalli, A. C. Murthy, C. Douglas, *et al.*, "Apache Hadoop YARN: yet another resource negotiator", in *ACM Annual Symposium on Cloud Computing (SoCC '13)*, 2013, pp. 1–16.

[109] S. Venkataramani, A. Raghunathan, J. Liu, *et al.*, "Scalable-effort classifiers for energy-efficient machine learning", in *Annual Design Automation Conference (DAC '15)*, 2015, pp. 1–6.

[110] R. L. Villars, C. W. Olofson, and M. Eastwood, *Big Data: what it is and why you should care?*, 2011. [Online]. Available: http://www.tracemyflows.com/uploads/big_data/idc_amd_big_data_whitepaper.pdf.

[111] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of Amazon EC2 data center", in *IEEE International Conference on Computer Communications (INFOCOM '10)*, 2010, pp. 1–9.

[112] G. Wang, A. R. Butt, P. Pandey, *et al.*, "A simulation approach to evaluating design decisions in MapReduce setups", in *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '09)*, 2009, pp. 1–11.

[113] T. White, *Hadoop: The definitive guide*. O'Reilly Media, 2012.

[114] K. Wiley, A. Connolly, J. P. Gardner, *et al.*, "Astronomy in the cloud: using MapReduce for image coaddition", in *Annual Conference on Astronomical Data Analysis Software and Systems (ADASS '11)*, 2011, pp. 93–96.

[115] K. Wilson, *Microsoft Office 365*. Springer, 2014.

[116] T. Wirtz and R. Ge, "Improving MapReduce energy efficiency for computation intensive workloads", in *IEEE International Green Computing Conference and Workshops (IGCC '11)*, 2011, pp. 1–8.

[117] H. Wu, K. Li, Z. Tang, *et al.*, "A heuristic speculative execution strategy in heterogeneous distributed environments", in *IEEE International Symposium on Parallel Architectures, Algorithms and Programming (PAAP '14)*, 2014, pp. 268–273.

[118] X. Wu, X. Zhu, G. Q. Wu, *et al.*, "Data mining with Big Data", *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 26, no. 1, pp. 97–107, 2014.

[119] H. Xu and W. C. Lau, "Optimization for speculative execution in Big Data processing clusters", *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 2, pp. 530–545, 2017.

[120] H. Xu and W. C. Lau, *Optimization for speculative execution of multiple jobs in a MapReduce-like cluster*, 2014. [Online]. Available: https://arxiv.org/pdf/1406.0609.pdf.

[121] H. Xu and W. C. Lau, "Task-cloning algorithms in a MapReduce cluster with competitive performance bounds", in *IEEE International Conference on Distributed Computing Systems (ICDCS '15)*, 2015, pp. 339–348.

[122] H. Xu and W. C. Lau, "Resource optimization for speculative execution in a MapReduce cluster", in *IEEE International Conference on Network Protocols (ICNP '13)*, 2013, pp. 1–3.

[123] H. Xu and W. C. Lau, “Speculative execution for a single job in a MapReduce-like system”, in *IEEE International Conference on Cloud Computing (CLOUD '14)*, 2014, pp. 586–593.

[124] Y. Xu and S. Mao, “A survey of mobile cloud computing for rich media applications”, *IEEE Wireless Communications*, vol. 20, no. 3, pp. 46–53, 2013.

[125] O. Yildiz, S. Ibrahim, T. A. Phuong, *et al.*, “Chronos: failure-aware scheduling in shared Hadoop clusters”, in *IEEE International Conference on Big Data (BigData '15)*, 2015, pp. 313–318.

[126] O. Yildiz, M. Dorier, S. Ibrahim, *et al.*, “On the root causes of cross-application I/O interference in HPC storage systems”, in *IEEE International Parallel and Distributed Processing Symposium (IPDPS '16)*, 2016, pp. 750–759.

[127] O. Yildiz, S. Ibrahim, and G. Antoniu, “Enabling fast failure recovery in shared Hadoop clusters: towards failure-aware scheduling”, *Future Generation Computer Systems (FGCS)*, vol. 74, pp. 208–219, 2017.

[128] M. Zaharia, D. Borthakur, J. Sen Sarma, *et al.*, “Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling”, in *ACM European Conference on Computer Systems (EuroSys '10)*, 2010, pp. 265–278.

[129] M. Zaharia, M. Chowdhury, T. Das, *et al.*, “Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing”, in *USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*, 2012, pp. 15–28.

[130] M. Zaharia, M. Chowdhury, M. J. Franklin, *et al.*, “Spark: cluster computing with working sets”, in *USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10)*, 2010, pp. 1–7.

[131] M. Zaharia, A. Konwinski, A. D. Joseph, *et al.*, “Improving MapReduce performance in heterogeneous environments”, in *USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, USENIX Association, 2008, pp. 29–42.

[132] A. C. Zhou, B. He, and S. Ibrahim, “A taxonomy and survey of scientific computing in the cloud”, in *Big Data: Principles and Paradigms*. Morgan Kaufmann, 2016, ch. 18, pp. 431–455.

[133] M. Zwolenski and L. Weatherill, “The digital universe: rich data and the increasing value of the Internet of Things”, *Australian Journal of Telecommunications and the Digital Economy*, vol. 2, no. 3, pp. 1–9, 2014.

Résumé en français

Contexte

Nous sommes entrés dans l'ère de Big Data où la taille des données qui sont générées, capturées et traitées quotidiennement augmente de manière effrénée. Selon une étude récente d'International Data Corporation [133], la quantité totale de données générée jusqu'en 2017 est approximativement de 16 zettaoctets. Dans des unités plus familières, ce montant correspond à 16 billions de gigaoctets ou à 16 milliards de téraoctets. De plus, cette étude montre que la taille des données double tous les deux ans. En conséquence, la quantité totale de données générées devrait atteindre 180 zettaoctets d'ici 2025.

Pour extraire de la valeur d'un tel volume de données, de nouveaux paradigmes de traitement sont apparus [31, 57]. Parmi ceux-ci, MapReduce [31] est devenu le modèle de programmation de référence pour le traitement Big Data en raison de son évolutivité et de sa facilité d'utilisation. Hadoop, une implémentation *open source* de MapReduce, est actuellement utilisée pour le traitement Big Data dans de nombreuses institutions, sociétés et universités [91, 113]. Par exemple, Yahoo ! traite des centaines de petaoctets de données annuellement en utilisant Hadoop [131]. Récemment, Spark a émergé en tant que système de traitement de données à faible latence en exploitant le traitement de données en mémoire [130].

Pour faire face à la demande élevée de calcul et de stockage lors du traitement Big Data, ces systèmes sont généralement déployés dans des infrastructures à grande échelle, par exemple des *clouds* publics ou des datacenters privés [77]. À titre d'exemple, Facebook exploite des *datacenters* regroupant plus de 60.000 machines pour traiter des centaines de téraoctets de données quotidiennement [105].

À une telle échelle, l'hétérogénéité des ressources est inévitable. Elle apparaît à différents niveaux des systèmes. Au niveau du matériel, plusieurs générations de matériel coexistent dans les infrastructures des clouds. Par conséquent, les utilisateurs n'ont aucun contrôle sur les matériels qui leur sont attribués [83]. Au niveau de l'application, le matériel est réparti physiquement entre différents utilisateurs. De ce fait, les ressources allouées à une application ne garantissent pas de fournir des performances constantes pendant la durée de vie de cette application [97, 111]. Cette hétérogénéité, à son tour, aboutit à une évidente *variabilité de performance* [131]. D'autre part, les infrastructures à grande échelle se composent de milliers de machines qui consomment collectivement une énorme quantité d'énergie, ce qui entraîne

un énorme coût opérationnel [46]. Par exemple, la consommation annuelle d'électricité des datacenters de Google dépasse 1.120 GWh, ce qui correspond à une facture d'électricité de 67 M $ [92].

À l'avenir, la variabilité de performance et la consommation d'énergie continueront d'être des préoccupations majeures pour la conception et l'exploitation des systèmes de traitement Big Data [74, 97]. L'échelle des infrastructures sous-jacentes doit augmenter pour faire face à l'augmentation implacable de la taille des données. Cette échelle croissante augmentera non seulement la variabilité de performance mais aussi la consommation d'énergie. À titre indicatif, les besoins en énergie pour le fonctionnement des systèmes de traitement Big Data devraient atteindre l'equivalent de la production d'une centrale nucléaire moyenne [82].

Dans le contexte du Big Data, un calcul se compose généralement d'un très grand nombre de tâches élémentaires. La *performance* d'un calcul est déterminée par la fin de sa dernière tâche. En raison de la variabilité élevée de performance, les temps d'exécution des tâches peuvent varier de manière importante au sein du même calcul. Même si les temps d'exécution d'un grand nombre de tâches restent proches du temps d'exécution moyen, certains d'entre eux peuvent présenter une très grande déviation. Il n'est pas rare dans la pratique d'observer certaines tâches avec des temps d'exécution jusqu'à huit fois plus longs que le temps d'exécution moyen [6]. Ce phénomène est appelé *distribution heavy-tail* [94]. Il a un impact négatif sur la performance du calcul [131]. Dans le domaine du Big Data, ces tâches nuisibles sont appelées *stragglers*.

Il existe un grand nombre de travaux consacrés à la réduction de la fréquence d'apparition de stragglers [27, 41, 104]. Cependant, la variabilité de performance entraine l'apparition de stragglers inattendus. Dans la pratique, il a été démontré que ces stragglers ont un impact majeur sur la performance [131]. En conséquence, la prévention des stragglers est un objectif crucial pour améliorer les performances des grands systèmes de traitement Big Data.

De nombreuses techniques de *prévention des stragglers* ont été introduites [4, 6, 31, 131]. Généralement, elles se composent de deux phases : la *détection des stragglers* et le *traitement des stragglers*. Dans la phase de détection, les tâches lentes (*e.g.*, tâches avec la vitesse ou le progrès en dessous de la moyenne) sont marquées comme stragglers [4, 6, 31, 131]. Ensuite, les stragglers détectés sont traités en utilisant la technique de *kill-restart* [6] ou la technique d'*exécution spéculative* [31]. Dans la première approche, le straggler est arrêté et puis relancé à partir de son état initial avec l'espoir qu'il puisse terminer plus rapidement. Dans la deuxième approche, une *copie* du straggler est lancée en parallèle avec le straggler. Dès lors que l'un d'entre eux achève son traitement, il est marqué comme *réussi* et l'autre est immédiatement *supprimé*. Cette copie est appelée *copie spéculative* dans le sens où il n'y a aucune garantie qu'elle puisse se terminer avant le straggler. L'exécution spéculative a été utilisée dans de nombreux environnements de traitement Big Data, tels que Hadoop et Spark [130, 131]. Par exemple, Google mentionne que l'exécution spéculative améliore les performances des calculs jusqu'à 44% [31].

Comme nous l'avons mentionné, la consommation d'énergie est une préoccupation majeure pour l'exploitation des systèmes de traitement Big Data. Malheureusement, l'exécution spéculative a un coût énergétique élevé, même si elle peut apporter une amélioration significative de la performance. En effet, l'énergie économisée en raccourcissant le temps d'exécution du straggler peut ne pas compenser l'énergie supplémentaire consommée par la copie spéculative. Pire encore, les copies spéculatives peuvent ne pas terminer avant les stragglers

et être supprimées. Dans la pratique, les techniques actuelles de prévention des stragglers ont encore un ratio élevé de copies spéculatives supprimées. Dans certains cas, il peut s'agir de 80% des copies spéculatives [93].

Plusieurs causes sont à l'origine de ce problème. Premièrement, les mécanismes actuels de détection des stragglers sont équipés d'algorithmes de filtrage simples afin de détecter rapidement les stragglers durant l'exécution [4, 6, 31, 131]. Ceux-ci peuvent entraîner des décisions de détection inexactes. Par exemple, ils peuvent détecter de trop nombreux en tant que stragglers. Par conséquent, des *copies spéculatives inutiles* sont lancées, et de ce fait les performances et la consommation d'énergie en souffrent.

Deuxièmement, la répartition des copies spéculatives au sein de l'infrastructure peut avoir un impact négatif. La variabilité de performance peut encore une fois avoir une incidence sur la performance et la consommation d'énergie des différentes répartitions de copies spéculatives. Malheureusement, les mécanismes existants de traitement de stragglers ne tiennent pas compte de cela [4, 6, 31, 131]. Ils peuvent déployer les copies spéculatives sur des ressources non appropriées, sur lesquelles les copies spéculatives auront de faibles performances et une forte consommation d'énergie.

Troisièmement, la mise en œuvre habituelle de l'exécution spéculative différencie les tâches régulières des copies spéculatives. Une fois que les ressources sont disponibles, les tâches régulières sont prises d'abord en considération, car ayant une priorité plus élevée, avant de prendre en compte les copies spéculatives. Par conséquent, les copies spéculatives ont seulement la possibilité de s'exécuter à la fin de l'exécution du calcul, lorsque toutes les tâches régulières ont été lancées [113]. Ce long délai peut rendre l'exécution spéculative moins efficace, car les stragglers continuent de s'exécuter et consomment de l'énergie pendant une période prolongée.

L'objet de cette thèse est d'améliorer les techniques de prévention des stragglers pour d'optimiser les performances des calculs et la consommation d'énergie.

Contributions

Dans cette thèse, nous introduisons la notion d'*efficacité de performance-énergie* d'un système. Il est défini comme la paire (P, E), où P représente le temps d'exécution du système et E représente sa consommation d'énergie. Un système est défini comme plus efficace dans ce sens si son temps d'exécution P est plus court et sa consommation d'énergie E est plus petite. Dans le cadre de cette thèse, nous utilisons le terme d'*efficacité énergétique* pour cette notion d'efficacité.

Cette thèse vise à offrir une meilleure compréhension de l'impact des techniques de prévention des stragglers, à la fois sur la performance et la consommation d'énergie. L'objectif est de proposer de nouvelles solutions pour améliorer l'*efficacité énergétique* de ces techniques dans les systèmes de traitement Big Data.

Résumé des contributions. Nous commençons par caractériser l'impact de la prévention des stragglers sur la performance et la consommation d'énergie des systèmes de traitement Big Data. Nous observons que l'efficacité énergétique des techniques actuelles de prévention des stragglers pourrait être considérablement améliorée. Cela motive une étude détaillée des deux phases : *détection des stragglers* et *traitement des stragglers*.

En ce qui concerne la détection de straggler, nous introduisons un cadre novateur pour caractériser et évaluer les mécanismes de détection des stragglers existants. Nous proposons un nouveau mécanisme de détection de straggler. Ce mécanisme de détection est implémenté dans Hadoop et se révèle avoir une efficacité énergétique plus élevée par rapport aux mécanismes les plus récents. En ce qui concerne le traitement des stragglers, nous présentons une nouvelle méthode de répartition des copies spéculatives qui prend en compte l'impact de l'hétérogénéité des ressources sur la performance et la consommation d'énergie. Enfin, nous introduisons un nouveau mécanisme éconergétique pour gérer les stragglers. Ce mécanisme fournit plus de ressources disponibles pour lancer des copies spéculatives en utilisant une réservation dynamique de ressources. Il est démontré qu'elle améliore considérablement l'efficacité énergétique en utilisant une simulation.

Caractériser l'impact de techniques de prévention des stragglers sur la performance et consommation d'énergie

De grands efforts ont été consacrées à l'amélioration des techniques de prévention de straggler en ce qui concerne la performance. Cependant, peu de travail se concentre sur la compréhension des implications de ces techniques sur la performance et la consommation d'énergie des systèmes de traitement Big Data. Dans cette thèse, nous appuyons sur Grid'5000 [59], une infrastructure configurable qui permet d'effectuer des expériences scientifiques à grande échelle. En utilisant Grid'5000, nous menons un ensemble d'expériences pour évaluer l'impact de techniques de prévention de stragglers sur la performance et la consommation d'énergie de Hadoop. Notre étude révèle que les techniques de prévention de stragglers peuvent parfois augmenter, parfois réduire la consommation d'énergie de Hadoop. Dans le premier cas, l'augmentation de la consommation d'énergie provient en partie de l'inexactitude des mécanismes existants de détection de stragglers utilisés dans Hadoop. Cela conduit à un grand nombre de copies spéculatives inutiles, ce qui entraîne des performances inférieures et une consommation d'énergie plus élevée. Dans le second cas, la consommation d'énergie du système est globalement réduite, car la consommation d'énergie supplémentaire introduite par les copies spéculatives est compensée par l'énergie économisée en raccourcissant l'exécution des applications. De plus, nous montrons que la consommation d'énergie supplémentaire varie selon les applications. Elle est déterminée par trois facteurs principaux : le temps d'exécution de copies spéculatives, le temps d'inactivité des machines et la répartition de copies spéculatives. Ce travail a conduit à une publication à la conférence DIDIS 2015 (voir [89]).

Mesure et activation de l'efficacité énergétique de la détection des stragglers

En dépit d'un grand nombre d'études visant à améliorer les mécanismes de détection de stragglers. Les évaluer précisément reste un défi à cause de l'absence de mesure spécifique. En réponse à ce défi, nous présentons un cadre étendu pour caractériser et évaluer les mécanismes de détection de stragglers. Nous commençons par un ensemble de mesure, spécialement conçues pour caractériser les mécanismes de détection de stragglers. Ensuite, nous développons un modèle architectural par lequel ces mesures peuvent être utilisées pour estimer la performance et la consommation d'énergie. Nous menons en outre une série d'expériences sur Grid'5000 pour caractériser les mécanismes existants de détection de stragglers.

Les résultats indiquent que les mécanismes existants [31, 131] pourraient significativement être améliorés. Dans certains cas, seulement 12% des tâches détectées sont des stragglers réels. En conséquence, un grand nombre de copies spéculatives inutiles sont lancées. Ces copies entraînent à leur tour un énorme gaspillage d'énergie. Cela illustre l'inefficacité énergétique des mécanismes existants de détection de stragglers.

Ces résultats nous motivent à introduire un mécanisme de détection de stragglers économe en énergie, appelé *Hierarchical*. Il fonctionne comme une couche secondaire de filtrage au dessus des autres mécanismes de détection de stragglers. Étant donné que les stragglers sont principalement causés par l'architecture des nœuds de calcul [6], *Hierarchical* ne considère que les tâches sur les noeuds lents, les nœuds avec une performance bien inférieure à la moyenne. Nous mettons en œuvre ce mécanisme de détection de stragglers dans Hadoop et l'évaluons à l'aide de tests représentatifs MapReduce [1]. Les résultats montrent que *Hierarchical* peut réduire considérablement l'énergie gaspillée sur des copies spéculatives supprimées, tout en maintenant une bonne performance par rapport aux mécanismes de détection de stragglers les plus récents. Ce travail a abouti à une publication lors de la conférence Euro-Par 2017 (voir [90]).

Méthode d'allocation de copie spéculative vers une haute efficacité énergétique

Les choix des ressources pour lancer les copies spéculatives conduisent à des résultats différents de performance et de consommation d'énergie. Malheureusement, très peu d'études en tiennent compte. Dans ce travail, nous présentons un nouveau mécanisme de traitement de stragglers équipé d'une méthode éconergétique pour allouer des copies spéculatives. Cette méthode d'attribution traite en priorité les stragglers les plus *critiques*, ceux qui devraient avoir les plus longs temps restants. Les copies de ces stragglers critiques ont une plus grande chance de se terminer avant eux. Ainsi, ces copies peuvent significativement réduire le temps d'exécution et la consommation élevée d'énergie de ces stragglers. En outre, nous présentons un modèle de performance et un modèle de consommation d'énergie. Ces deux modèles mesurent le compromis entre la performance et la consommation d'énergie lors de la répartition de copies spéculatives entre différentes ressources. Ils sont utilisés pour guider notre méthode d'allocation de copie spéculative aux ressources appropriées, ce qui peut entraîner de meilleures performances avec une consommation d'énergie plus faible. Cette méthode d'allocation de copie spéculative est implémentée dans Hadoop. Elle peut fonctionner avec n'importe quel mécanisme de détection de stragglers fourni par Hadoop. Nous évaluons notre méthode d'allocation de copie spéculative sur Grid'5000 [59] en utilisant trois applications représentatives de MapReduce [1]. Les résultats expérimentaux montrent qu'elle peut réduire la consommation d'énergie tout en garantissant des performances comparables aux méthodes les plus récents d'allocation de copies spéculatives.

Un mécanisme de réservation pour améliorer l'efficacité énergétique du traitement des stragglers

Le problème de savoir *quand* lancer les copies spéculatives est crucial. Lancer une copie spéculative trop tard ne lui laisse aucune chance de terminer plus tôt que le straggler. Cependant, le lancement des copies le plus tôt possible sans considérer la question de l'emplacement des copies peut également entraîner de mauvais résultats. La raison en est à nouveau

l'hétérogénéité, comme indiqué ci-dessus. Le lancement d'une copie spéculative sur la première ressource disponible peut laisser inutilisées certaines ressources à venir qui offrent de meilleures performances avec une consommation d'énergie plus faible. Par conséquent, répondre en harmonie aux questions *quand* et *où* est la clé pour obtenir de meilleures performances et réduire la consommation d'énergie.

Dans ce travail, nous introduisons un nouveau mécanisme de traitement de stragglers qui adopte une approche basée sur les réservations pour fournir dynamiquement les ressources pertinentes au moment opportun. Notre objectif est d'optimiser la performance et la consommation d'énergie durant l'exécution. Tout d'abord, nous proposons un nouveau modèle de performance qui repose sur l'historique d'exécution pour estimer les temps d'exécution de nouvelles tâches ou des copies spéculatives. Nous introduisons également un nouveau modèle de consommation d'énergie qui tient compte de l'impact de la contention des ressources tout en associant différentes tâches. Ces deux modèles sont utilisés pour estimer la performance et la variation d'énergie des répartitions différentes de tâches et de copies spéculatives, afin d'atteindre l'objectif d'optimisation de la performance et de l'énergie. Cette information nous aide à sélectionner les meilleurs ressources pour lancer des copies spéculatives, et donc répondre à la question *où*. Deuxièmement, nous proposons une technique de réservation basée sur des fenêtres de temps pour sélectionner dynamiquement le meilleur moment pour lancer des copies spéculatives. Ceci répond à la question de *quand*. Notre solution proposée est évaluée à l'aide d'un ensemble de simulations. Les résultats montrent qu'elle offre une amélioration significative de la performance et de l'efficacité énergétique. Ce travail a été partiellement réalisé au cours d'un stage de 3 mois à l'Université Nationale de Singapour.

Publications

Articles de journaux internationaux

- Shadi Ibrahim, **Tien-Dat Phan**, Alexandra Carpen-Amarie, Houssem-Eddine Chihoub, Diana Moise, Gabriel Antoniu. *Governing Energy Consumption in Hadoop through CPU Frequency Scaling : an Analysis*. In the Journal of Future Generation Computer Systems (**FGCS**), *Vol 54(C), January 2016*. Impact factor 2016 : 3.997.

Communications en conférences internationales

- **Tien-Dat Phan**, Shadi Ibrahim, Gabriel Antoniu, Luc Bougé. *On Understanding the Energy Impact of Speculative Execution in Hadoop*. In Proceeding of the 2015 IEEE International Conference on Data Science and Data Intensive Systems (**DSDIS '15**), *Sydney, December 2015*.

- **Tien-Dat Phan**, Shadi Ibrahim, Amelie Chi Zhou, Guillaume Aupy, Gabriel Antoniu. *Energy-Driven Straggler Mitigation in MapReduce*. In Proceedings of the 2017 International European Conference on Parallel and Distributed (**Euro-Par '17**), *Santiago de Compostela, August 2017*. CORE Rank A (acceptance rate 28%).

Posters en conférences internationales

- **Tien-Dat Phan**. *Green Big Data Processing in Large-scale Clouds : Towards Energy Efficient Speculative Execution in Hadoop*. In the PhD Forum of the 2016 IEEE International Parallel & Distributed Processing Symposium (**IPDPS '16**), *Chicago, May 2016.*