A New Framework for Evaluating Straggler Detection Mechanisms in MapReduce

Tien-Dat Phan, Guillaume Pallez, Shadi Ibrahim, Padma Raghavan

A New Framework for Evaluating Straggler Detection Mechanisms in MapReduce

TIEN-DAT PHAN, Inria, Univ Rennes, CNRS, IRISA
GUILLAUME PALLEZ, Inria Bordeaux Sud-Ouest & University of Bordeaux
SHADI IBRAHIM, Inria, IMT Atlantique, LS2N
PADMA RAGHAVAN, Vanderbilt University

Big Data systems (e.g., Google MapReduce, Apache Hadoop, Apache Spark) rely increasingly on speculative execution to mask slow tasks, also known as stragglers, because a job's execution time is dominated by the slowest task instance. Big Data systems typically identify stragglers and speculatively run copies of those tasks with the expectation that a copy may complete faster to shorten job execution times. There is a rich body of recent results on straggler mitigation in MapReduce. However, the majority of these do not consider the problem of accurately detecting stragglers. Instead, they adopt a particular straggler detection approach and then study its effectiveness in terms of performance, e.g., reduction in job completion time, or efficiency, e.g., high resource utilization. In this paper, we consider a complete framework for straggler detection and mitigation. We start with a set of metrics that can be used to characterize and detect stragglers including Precision, Recall, Detection Latency, Undetected Time and Fake Positive. We then develop an architectural model by which these metrics can be linked to measures of performance including execution time and system energy overheads. We further conduct a series of experiments to demonstrate which metrics and approaches are more effective in detecting stragglers and are also predictive of effectiveness in terms of performance and energy efficiencies. For example, our results indicate that the default Hadoop straggler detector could be made more effective. In certain case, Precision is low and only 55% of those detected are actual stragglers and the Recall, i.e., percent of actual detected stragglers, is also relatively low at 56%. For the same case, the hierarchical approach (i.e., a green-driven detector based on the default one) achieves a Precision of 99% and a Recall of 29%. This increase in Precision can be translated to achieve lower execution time and energy consumption, and thus higher performance and energy efficiency; compared to the default Hadoop mechanism, the energy consumption is reduced by almost 31%. These results demonstrate how our framework can offer useful insights and be applied in practical settings to characterize and design new straggler detection mechanisms for MapReduce systems.

CCS Concepts: • **Computer systems organization → Distributed architectures**; • **General and reference →** *Measurement*; *Metrics*;

This work is supported by the ANR KerStream project (ANR-16-CE25-0014-01) and the Stack/Apollo connect talent project. The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.fr/ for details). Part of this work was done when Guillaume Pallez was with Vanderbilt University, and Tien-Dat Phan was visiting Vanderbilt with the support of Vanderbilt Institutional Fund.
Authors' addresses: T-D. Phan, ENS Rennes/IRISA, Campus Universitaire de Beaulieu, 35042 Rennes, France; G. Pallez, Inria Bordeaux Sud-Ouest, 200 avenue de la vieille tour, 33400 Talence, France; S. Ibrahim (corresponding author), Inria Rennes - Bretagne Atlantique, Campus Universitaire de Beaulieu, 35042 Rennes, France; email: shadi.ibrahim@inria.fr; P. Raghavan, Vanderbilt University, Nashville, TN 37235-1826, USA.

2376-3639/20XX/X-ARTXX $15.00
https://doi.org/0000001.0000001

Additional Key Words and Phrases: Modelisation, Performance Evaluation, MapReduce, Hadoop, Speculation, Stragglers, Energy Efficiency

ACM Reference format:
Tien-Dat Phan, Guillaume Pallez, Shadi Ibrahim, and Padma Raghavan. 20XX. A New Framework for Evaluating Straggler Detection Mechanisms in MapReduce. *ACM Trans. Model. Perform. Eval. Comput. Syst.* X, X, Article XX (X 20XX), 22 pages.
https://doi.org/0000001.0000001

1 INTRODUCTION

Many large-scale data analyses rely on MapReduce [7]. Running upon commodity hardware (e.g., in the cloud and large-scale data centers), MapReduce is expected to process Big Data applications correctly and with acceptable response time, against hardware failure and more importantly against performance variability that introduces long execution time: *performance variability in large-scale infrastructures (e.g., clusters, clouds) — stemmed from heterogeneity and resource dynamicity and multi-tenancy — causes a severe slow down in task's runtimes of 700-800%[1].*

Traditionally, Big Data systems (Google MapReduce [7], Apache Hadoop [19, 20], Apache Spark [25]) are designed with hardware failure in mind. In particular, Hadoop [1] tolerates machine failures (crash failures) by re-executing all the tasks of the failed machine by the virtue of data replication. Furthermore, in order to mask the effect of stragglers (tasks performing relatively slower than other tasks, due to the variation in CPU availability, network traffic or I/O contention), MapReduce re-launches other copies of detected stragglers on other machines.

Existing works on mitigating stragglers have largely focused on when and where to schedule speculative copies to improve the overall execution time of the application [26], and on improving the straggler detection mechanism by feeding the detectors with more information. Most of these studies focus on the effectiveness of their detection mechanisms by evaluating the reduction in the job's execution time [1, 2, 5–7, 26] or the reduction in the heavy-tail task execution times [1, 2]. Other studies assess the efficiency of the straggler detection mechanism through evaluation metrics such as the total number of successful speculative copies [2, 22, 23], extra resource usage [2, 22]. This is insufficient, imprecise, and sometimes even results in incorrect interpretations.

To illustrate this observation, assume a scenario where 10% of the tasks are actual stragglers. In this scenario, a detection mechanism that detects as stragglers 50% of the tasks (among which roughly 5% are actual stragglers) may obtain similar performance as a mechanism that only detects 50% of the actual stragglers. However, by only looking at the execution time, how can a system administrator predict the performance of both mechanisms in other scenarios, and how can they choose which detection mechanism is better for their platform?

Our work tackles this issue by introducing a list of evaluation metrics to characterize the straggler detection mechanisms. We further show how one can use those metrics to predict their effectiveness in improving the performance or reducing the energy consumption in MapReduce. Energy consumption is a good metric to study side by side with performance: for one it is a critical concern in current data-center (with energy costs ranging in the millions of dollars [16]), and further for an identical performance, it can measure over-utilization of servers by useless computations. *To the best of our knowledge, no previous studies have worked on assessing, clarifying and analyzing straggler detectors. Furthermore, no previous studies in straggler detection have shown how to use metrics to characterize straggler detection mechanisms, and predict their effectiveness in terms of performance and energy efficiency.*

[1]For simplicity, in this paper Hadoop refers to the MapReduce implementation in Hadoop (Hadoop MapReduce)

Contributions: In this paper, we suggest a new framework that characterizes the straggler detection mechanisms by utilizing a comprehensive list of evaluation metrics, including *Precision*, *Recall*, *Detection Latency*, *Undetected Time*, and *Fake Positive*. While the *Precision* (i.e., percent of actual stragglers among the detected ones) and *Recall* (i.e., percent of actual detected stragglers) are derived from well-used parameters in the detection community (i.e., false negative, true positive and false positive), *Detection Latency* and *Undetected Time* are new metrics which we have introduced to deal precisely with stragglers: different from failures which are very punctual events (either detected or not), stragglers are very lasting events and it is important to consider when they are detected and for how long they run if they are not detected. We then develop an architectural model by which these metrics can be linked to measures of performance including execution time and system energy overheads. We further conduct a series of experiments to demonstrate which metrics and approaches are more effective in detecting stragglers and are also predictive of effectiveness in terms performance and energy efficiencies. We do so through a deployment of Hadoop (state-of-the-art MapReduce implementation) on the Grid'5000 platform [4], i.e., a highly-configurable infrastructure that supports users to perform experiments at large scale. Our results indicate that the default Hadoop straggler detector could be made more effective. In certain case, *Precision* is low and only 55% of those detected are actual stragglers and the *Recall* is also relatively low at 56%. For the same case, the hierarchical approach [14], i.e., a green-driven straggler detection mechanism, achieves a *Precision* of 99% and a *Recall* of 29%. This increase in precision can be translated to achieve lower execution time and energy consumption, and thus higher performance and energy efficiency; compared to the default Hadoop mechanism, execution time and energy consumption are reduced by almost 32% and 31%, respectively. We also show that early launching of speculative copies, by triggering the straggler detection at an early stage of the job execution and providing dedicated resources for them, can strongly impact the performance and energy efficiency. These results demonstrate how our framework can offer useful insights and be applied in practical settings not only to characterize straggler detectors but also in designing new straggler detection mechanisms for MapReduce systems.

Goals: The main focus of this paper is providing an evaluation framework to characterize straggler detection mechanisms in Big Data systems and use this framework to study their efficiency and effectiveness in improving the performance and the energy efficiency.

It is important to note that the metrics we present here are neither limited to the implementations of Hadoop [19, 20] nor specific to the MapReduce programming model: these metrics are meant to be agnostic of the MapReduce frameworks/versions used, where the difference lies in the applied detection algorithm and not in the concept of stragglers. The values for the metrics for a given straggler detection mechanism, will however depend on the programming model of the Big Data system. Moreover, we select and use one simple yet representative MapReduce application, namely *WordCount* in our evaluation. The goal is to ensure full control of the behavior of the application and to avoid any unexpected deviation which can impact the results. However, the evaluation of these metrics can be used with any application.

The rest of the paper is organized as follows: Section 2 presents the background knowledge and the related work. Section 3 describes the proposed evaluating metrics. Section 4 mentions the mathematical intuition for understanding the meaning of our evaluation metrics. The experimental setups will be discussed in details in Section 5. Section 6 demonstrates the practicality of our metrics in characterizing the straggler detection. Section 7 presents the applications of our metrics on understanding the speculation effectiveness in practice. Finally, we conclude the paper in Section 8.

2 BACKGROUND & RELATED WORK

In this section, we briefly mention the MapReduce programming model. Subsequently, the architecture of Hadoop, the popular implementation of MapReduce, is described. Moreover, all technical terms used throughout the paper will be systematically presented. Finally, the related work is discussed at the end of the section.

2.1 Background

2.1.1 MapReduce. MapReduce [7, 12] is a programming model for processing and generating large data sets in large-scale distributed systems. It enables the user customization with easy-to-implement map and reduce functions. The input dataset is divided into small chunks which will be handled by map functions. The reduce functions collect the intermediate results from Map tasks and generate the final result. The underlying runtime system plays the role of parallelizing the computation across large-scale infrastructure, handling failures as well as scheduling inter-machine communication processes.

2.1.2 Hadoop. The Apache Hadoop project [15, 20] is an open-source framework for managing and processing large data sets in clusters. Hadoop MapReduce is an implementation of the Google MapReduce programming model [7]. Usually, Hadoop MapReduce operates on the top of the Hadoop Distributed File System (HDFS) [9], but it can also run on the top of other filesystems such as Amazon S3, Azure Blob Storage, and OpenStack Swift. Hadoop MapReduce adopts a master/slave architecture. The master node, named ResourceManager (RM), is responsible for communicating with the NameNode (i.e., the master node of HDFS) for data access and managing the use of resources across the cluster. The slave nodes, named NodeManagers (NMs), are responsible for launching tasks and monitoring the resource usage per node. In addition, there is a per application component, named ApplicationMaster (AM), which is responsible for requesting the resources of a single job from the RM and monitoring the progress of tasks running on the NMs. The resource management of Hadoop uses *container* as the resource unit. Each container can run at most one task at a time. Every NodeManager, i.e., worker, in Hadoop cluster is configured with a specific number of containers.

2.1.3 Technical Terms and Definitions. In order to make it easy to the readers to follow the paper, we present collectively, in Table 1, all the technical terms used in the paper, as well as their definitions.

2.2 Related work

A large body of research has been dedicated to improve the straggler mitigation in MapReduce. Dean et al. [7] presented a mechanism of backing up slow tasks (stragglers) for improving the job performance. As soon as the normal tasks are all launched, it will start searching for stragglers. A task, which has the progress less than the average progress minus 20%, will be marked as straggler. Zaharia et al. [26] presented a different speculation mechanism which takes into consideration both the progress and the task's elapsed time. These two parameters are used to calculate the progress rate of each task, which represents how fast the execution of the task is. Relying on the progress rates, the task which is expected to finish last will be first duplicated. Ananthanarayanan et al. [2] proposed a new *cause-aware* straggler mitigation mechanism, named Mantri. It keeps monitoring the performance and the resource consumption of the tasks and uses this information for detecting the causes (non-local task, data skew, etc) that originate the slow execution. Based on this information, Mantri schedules speculative tasks only when there is a fair chance to reduce task execution times with a low resource consumption. Chen et al. [5] also considered the impact of

Terms	Definitions
Execution time	The time that a task/job takes from the moment it starts until it finishes.
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 capacity of the cluster. Though tasks are scheduled once resources are freed, Map tasks of the same job have equal work and thus tend to start and finish in "waves" at almost the same times. Accordingly, if the tasks number of a job equals x times the total cluster's 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. By default, speculative copies are launched in the last wave of the job, that is, when there are available resources and all regular tasks have been launched.
Successful speculative copy	The speculative copy that can finish before its original task. This task thus is marked as successful task. 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, thus, gets killed.
Speculative lag	This parameter is used for triggering the straggler detection and handling. The straggler detection 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. The default value for this parameter is 60 seconds.

Table 1. Technical terms and definitions.

data locality and the data skew when detecting the stragglers. Moreover, it detects the stragglers using also progress rate and the network usage within each phase of the execution (e.g., mapping, merging, combining for Map tasks and shuffling, and sorting and reducing for Reduce tasks) to detect the stragglers. Considering that the majority of jobs in production Hadoop clusters are small jobs, Ananthanarayanan et al. [1] presented Dolly, a new approach to handle stragglers. Dolly launches multiple copies (i.e., clones) of tasks belonging to small jobs. After the first clone of a task is complete, the other clones will be killed in order to free the resources. By cloning the small jobs, Dolly results in a significant performance improvement with an acceptable extra resource consumption. Adopting similar idea, Xu et al. [24] proposed two scheduling algorithms, online and offline, using the task-cloning technique for reducing the job execution time. *In contrast to related work, we introduce the first study that characterizes straggler detection mechanisms in MapReduce and provides a sufficient list of metrics to study and compare their effectiveness and efficiency (including the aforementioned ones).*

3 METRICS FOR CHARACTERIZING STRAGGLER DETECTION MECHANISMS

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

Goal	Metric	Description	Related work
Efficiency	Execution time	Measurement of the impact of speculative execution on reducing the execution time	[1, 2, 5, 6, 26]
	Resource consumption	Measurement of the reduction/increase in total resource consumption	[1, 2, 22]
	Heavy-tail reduction	Reduction of the ratio between the longest and average tasks' execution time	[1, 2]
	Wasteful resource occupation	The amount of resource consumed by unsuccessful copies	[26]
Characterizing	# of speculative copies	The number of speculative copies launched throughout the execution	[2, 21, 26]
	# of Successful copies	Total number of speculative copies successfully finish	[2, 21]

Table 2. Existing evaluating metrics.

3.1 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 2 lists the existing metrics which are used in many studies related to straggler mitigation.

While efficiency metrics can measure the impact of a straggler detection 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 of any system.

As an example, we analyzed a one-month trace of three production Hadoop clusters [17] and tried to find the correlation between the number of successful speculative copies and the heavy-tail execution reduction. Although this *number of successful copies* metric is used to explain the impact in reducing the execution time [21] of speculative execution, Table 3 shows that the absolute value of the correlation coefficient only ranges between 0.04-0.14 for the three clusters (while the maximum 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. In addition, we also demonstrate again that it is not sufficient to use this metric for understanding the efficiency of straggler detection in Section 7.3. Thus, we need to consider a complete framework for straggler detection and mitigation. Such a framework can assist a system administrator to predict the performance of straggler detection mechanisms in different scenarios and thus help them to choose the detection mechanism which fits the best their needs.

3.2 Precision, recall, detection latency and undetected time

We now discuss new parameters to characterize stragglers that can be used 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 ($T \in \overline{\texttt{det}}$). When

Cluster	Time	#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

Table 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 execution time for each job. The results show that the correlation is ranked from very weak to no correlation.

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 ($\mathtt{stg} \wedge \overline{\mathtt{det}}$);
- *True Positive*: a detected straggler ($\mathtt{stg} \wedge \mathtt{det}$);
- *False Positive*: a task detected that in the end was not a straggler ($\overline{\mathtt{stg}} \wedge \mathtt{det}$).

Note that *False Negative*, *True Positive* and *False Positive* are parameters that are well-used in the detection community (see for example in failure detection [3, 8]). Based on these, we can define the *precision p* and *recall r* of a straggler detection mechanism, that is:

$$p = \frac{|\mathtt{stg} \wedge \mathtt{det}|}{|\mathtt{det}|} = \frac{True_P}{True_P + False_P} \tag{1}$$

$$r = \frac{|\mathtt{stg} \wedge \mathtt{det}|}{|\mathtt{stg}|} = \frac{True_P}{True_P + False_N}. \tag{2}$$

Simply put, the precision is the number of correct predictions amongst all predictions, and the recall is the ratio of predicted stragglers amongst all stragglers.

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 very lasting event. It seems important to reward detectors that could detect stragglers very early on. *Intuitively, there is a difference between a detector 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, how fast the straggler is detected compared to its usual execution time, that is:

$$\text{Detection Latency} = \frac{\sum_{\mathtt{stg} \wedge \mathtt{det}} \frac{\text{Time to detection}}{\text{Usual execution time}}}{|\mathtt{stg} \wedge \mathtt{det}|} \tag{3}$$

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

$$\text{Undetected Time} = \frac{\sum_{\mathtt{stg} \wedge \overline{\mathtt{det}}} \frac{\text{Execution Time}}{\text{Usual execution time}}}{|\mathtt{stg} \wedge \overline{\mathtt{det}}|} \tag{4}$$

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

4 LINKING STRAGGLER DETECTION METRICS TO PERFORMANCE

In this section, we propose an architectural model for the platform energy consumption and slow-down as a function of its load. We then use this model to illustrate the relationship of the different metrics for straggler detection to performance measures such as execution time and system energy.

4.1 Architectural Model for Performance

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 ct. 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).

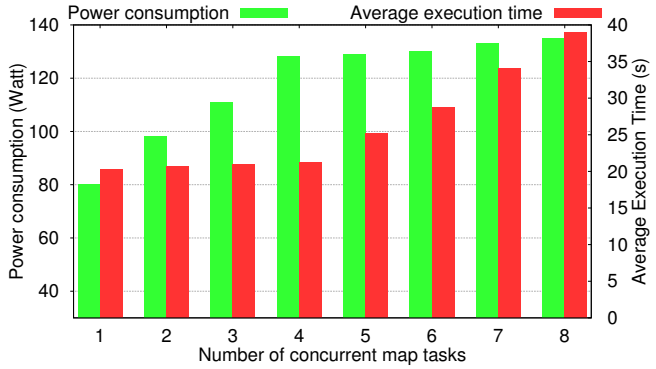

Fig. 1. Observations for both (i) the average task execution time and (ii) power consumption when varying the number of concurrent running tasks: Map task when running *WordCount* application (compute-intensive tasks). Using the model developed in Section 4.1, we obtain: $\mathcal{P}_{\text{static}} = 65W$ and $\mathcal{P}_{\text{dyn}} = 17W$.

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.

Finally, this can be written mathematically, for n the number of tasks running concurrently, the power $\mathcal{P}$ is:

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

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 $ct - 1$ tasks running on them, adding a task on them would not increase the power consumption.

Execution time. We are now interested in evaluating the slowdown incurred by executing multiple tasks on one node. To be able to model how running concurrent tasks impacts the execution time, we need to understand the different bottlenecks:

- Processing power: when two (or more) tasks are executed on a same *core*, they share the processing power, hence inducing a slowdown on the compute part of a task execution.
- Fast storage available: when two (or more) tasks are executed on a same *node*, they share resources such as memory and high-level cache, hence inducing a slowdown on the data-access part of a task execution.

It is hard to give a precise model of the task slowdown due to interference because it mostly depends on the task profiles (e.g., compute-intensive versus data-intensive). Intuitively, core-sharing (hence multi-threading) will impact more compute-intensive applications than node sharing (using multi-core). On the contrary, sharing storage resources will impact more data-intensive applications than core-sharing (we expect the overhead of core-sharing to be minimal).

4.2 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 a very simple application of n tasks, and a very simple architecture with $2n$ nodes each with a single core. We assume that amongst those, we have a ratio α 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 6.2 with the introduction of *Fake Positive* metric).
- We assume that the detection of a non-straggler occurs on average at 50% its execution.

We can evaluate the performance and energy of a task depending on different parameters, namely whether it is a straggler ($\texttt{stg}$) or not ($\overline{\texttt{stg}}$), and whether it was detected as a straggler ($\texttt{det}$) or not ($\overline{\texttt{det}}$). 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: $\mathbb{P}(\texttt{stg} \wedge \texttt{det}) = \mathbb{P}(\texttt{stg})\mathbb{P}(\texttt{det}|\texttt{stg}) = r\alpha$

$$\text{Time Overhead} \approx 1 + \text{Detection Latency}$$

$$\text{Energy cost} \approx (2 + \text{Detection Latency}) \cdot (\mathcal{P}_{\text{static}} + \mathcal{P}_{\text{dyn}})$$

The time overhead is the time it takes to detect the straggler (Detection Latency) and the time for a usual 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.

Non-detected stragglers. They occur with probability: $\mathbb{P}(\texttt{stg} \wedge \overline{\texttt{det}}) = \mathbb{P}(\texttt{stg})\mathbb{P}(\overline{\texttt{det}}|\texttt{stg}) = (1-r)\alpha$

$$\text{Time Overhead} \approx \text{Undetected Time}$$

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

The time overhead is the time it takes for the non-detected straggler to execute. A long-running non-detected straggler can severely prolong the whole job 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: $\mathbb{P}(\overline{\texttt{stg}} \wedge \texttt{det}) = \frac{False_P}{n} = r\alpha\frac{1-p}{p}$

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

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: $\mathbb{P}(\overline{\texttt{stg}} \wedge \overline{\texttt{det}}) = 1 - \mathbb{P}(\texttt{stg} \wedge \overline{\texttt{det}}) - \mathbb{P}(\texttt{stg} \wedge \texttt{det}) - \mathbb{P}(\overline{\texttt{stg}} \wedge \texttt{det}) = 1 - \alpha(1 + \frac{r(1-p)}{p})$

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

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 naive 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.

5 EXPERIMENTAL METHODOLOGY

In this section, we describe the experimental setups used throughout our experiments.

5.1 Testbed

All the experiments conducted throughout the evaluation were executed on Grid'5000 testbed [4]. 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 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 Gbps Ethernet network for intra-cluster communication.

Moreover, each node on this cluster is attached with power monitoring hardware including 2 Power Distribution Units (PDUs), which allow us to acquire fine-grained power consumption information on each node individually during the experiments.

5.2 Platform

We deployed Linux Ubuntu 16.04 LTS on our cluster. Moreover, Hadoop 2.7.3 [20], 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, which is hosting the NameNode and the ResourceManager

(RM) processes. The rest 20 nodes were each running one DataNode process and one NodeManager (NM) process. Each NodeManager node was configured with 8 virtual cores and thus can run a maximum of 8 tasks (Map or Reduce) simultaneously. In terms of the Hadoop file system HDFS, we kept the default setting, where the replication factor was set to 3 and the block size was 64 MB.

5.3 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.

The WordCount 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. Thus, by using WordCount application, we can limit the "real" stragglers to the ones resulted from resource heterogeneity and therefore provide a precise interpretation of the results. There are no other sources of stragglers such as application-specific features (i.e., computation complexity, the size of generated data and computation skew [2, 11, 13]). In addition, the computation time of Map tasks is relatively long, this also reduces the possibility of having stragglers due to non-local executions [10].

The basic characteristics of the WordCount job are presented in Table 4. The cluster capacity is 160 Map tasks. This job, which consists of 320 Map tasks, runs two waves of Map tasks. We chose this setting based on real-life Hadoop production cluster traces [17]. As we observed, there were roughly 25% of the jobs which have at least 2 waves. More importantly, these jobs contribute roughly 90% to the total launched Map tasks. Therefore, our setting can accurately reflect the jobs' execution in real-life Big Data systems. It is important to note that both the WordCount application and the input data are part of the Puma MapReduce benchmark suite [18].

5.4 Straggler injection

We propose a scheme to proactively control the straggler ratio injected throughout the experiments. For our cluster of 20 workers, we divide them into four groups $G_i, i = \overline{1,4}$. 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 $C_j = \{p_1, p_2, p_3, p_4\}$. Throughout our experiments, we vary this straggler injection ratio to present different scenarios covering a broad range of straggler occurrence, which are: $C_1 = (35, 35, 5, 25)$, $C_2 = (25, 25, 25, 25)$, $C_3 = (10, 10, 5, 75)$ and $C_4 = (5, 5, 0, 90)$. Simply put, in configuration C_4, 5% of the nodes have only one active core, 5% have two, and 90% have four. Hence for an identical load on all nodes, 10% of the tasks should be stragglers. Note that the stragglers are created during the whole lifetime of an application.

5.5 Straggler detection mechanisms

Throughout our experiments, we examined three straggler detection mechanisms, two from the literature: *Default* [7] and *LATE* [26] mechanisms. The third mechanism we consider is *Hierarchical* [14], which is a green straggler detection scheme that is applied hierarchically on the top of *Default*. Hereafter, we provide brief descriptions of the three mechanisms.

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

Table 4. Application characteristics and configurations

Default. This is the straggler detection mechanism introduced in [7]. It is based on progress score (PS). The progress score is defined as a 0-to-1 number, which represents the ratio of processed data over the total input data of a task (see Equation 5).

$$\mathrm{PS} = \frac{\mathrm{Size}_{\text{processed data}}}{\mathrm{Size}_{\text{total data}}} \tag{5}$$

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 Equation 6:

$$\mathrm{PS}_j < \left(\frac{\sum_{i=1}^{N} \mathrm{PS}_i}{N} - 0.2\right) \tag{6}$$

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% [7].

LATE. Zaharia et al. [26] 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 7:

$$\mathrm{PR} = \frac{\mathrm{PS}}{\mathrm{t}_{\text{current}} - \mathrm{t}_{\text{start}}} \tag{7}$$

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 $\overline{\mathrm{PR}}$ and the standard deviation SD (as shown in Equations 8 and 9).

$$\overline{\mathrm{PR}} = \frac{1}{N}\sum_{i=1}^{N} \mathrm{PR}_i \tag{8}$$

$$\mathrm{SD} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} (\mathrm{PR}_i - \overline{\mathrm{PR}})^2} \tag{9}$$

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

$$\mathrm{PR}_j < \overline{\mathrm{PR}}(1 - \alpha \times \mathrm{SD}) \tag{10}$$

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). 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 [26]. LATE is used as the default detection mechanism in Hadoop 2.7.3.

Hierarchical. The main goal of the *Hierarchical* detector [14] is to reduce the energy consumption. Therefore, following the discussion in Section 4.2, we considered the following objectives:

(i) to improve the *Precision* of an existing detection mechanism (by detecting less *wrong* stragglers), and
(ii) to improve the *Undetected Time* by focusing on the stragglers on potentially very slow machines when the number of re-execution is limited.

In parallel, it is expected that this comes at a cost of:

(i) a lower *Recall* (while it is supposed to reduce primarily the number of *False Positives*, it expects to also not detect some *True Positives*), and,
(ii) a higher *Detection Latency* (as it adds additional computations to the detection)

***Hierarchical* detector: How it works?** There are diverse reasons that can cause the performance variation, e.g., the node hardware, the software bugs, resource contention, etc. When one of these happens, all the tasks on the node will be most likely affected. Therefore, their performance will degrade. This detection focuses on the stragglers located on the slow nodes.

This strategy works on top of other straggler detection mechanisms. It starts with the list of stragglers detected by the underlying straggler detection layer as input $\mathcal{L}$. Then it trims this list ($\mathcal{L}$) using the following process:

(i) It first extracts the performance information of all running tasks.
(ii) It then evaluates for each task T_i an *approximate* speed (Speed_i) at which it has been running by using information about that task, its progress score, that is the percentage of work done, its input size (we make the approximation that the amount of work to be done is proportional to the size of the input of the task) and the time it has been running ($time - start\ time$):

$$\text{Speed}_i = \frac{\text{PS} \times \text{Size}_{\text{total data}}}{\text{t}_{\text{current}} - \text{t}_{\text{start}}}$$

(iii) Based on this, it evaluates the average performance of each node $\mathcal{N}$:

$$\text{Perf}_{\mathcal{N}} = \frac{\sum_{T_i \in \mathcal{N}} \text{Speed}_i}{|\mathcal{N}|},$$

where without ambiguity $\mathcal{N}$ is also the set of tasks running on node $\mathcal{N}$.
(iv) Finally, it only keeps the subset of $\mathcal{L}$ located on the slowest nodes which have the performance of less than 90% of the cluster average performance (for the detection mechanism *Hierarchical*).

6 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 three straggler detection mechanisms, including *Default* [7], *LATE* [26] and *Hierarchical* using our metrics.

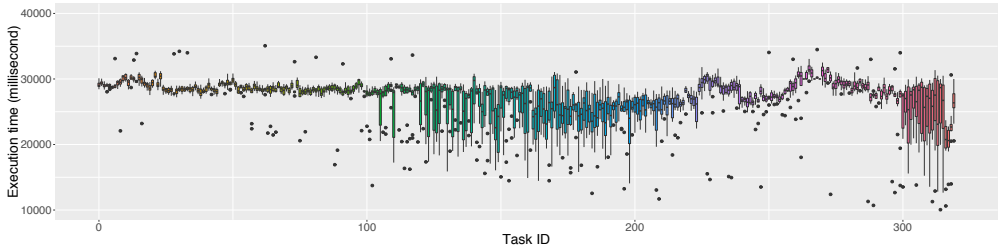

Fig. 2. Distribution of execution times (in milliseconds) per task ID for the WordCount application on a homogeneous platform.

6.1 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 prediction mechanism. According to the straggler definition [7], a straggler is a task that takes more than 1.2 times *the time it usually takes to be executed*. Intuitively, by running enough executions of an application in a homogeneous environment (hence an environment (i) with very few stragglers, and (ii) where the stragglers can be manually detected), then one can compute the usual execution time of each task by taking the X quantile[2] where X can be defined according to those results. By comparing the task execution time to the *time it usually takes to be executed*, 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 [2, 5].

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 2.

In this case, we could then evaluate that the *usual* 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 2, there are 21 outliers. We give in Table 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 *usual execution time*. We put our raw data online for anyone to play with[3]. 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. For instance if we denote by

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

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

$$1.2T_{\text{task}}(1+\varepsilon_{\text{guess}}) \leq s_{\min}T_{\text{task}}(1-\varepsilon_{\text{setting}}) \tag{11}$$

$$T_{\text{task}}(1+\varepsilon_{\text{setting}}) \leq 1.2T_{\text{task}}(1-\varepsilon_{\text{guess}}) \tag{12}$$

[2]Meaning that a proportion X of the execution times were below this value, and $1-X$ above.
[3]https://gitlab.inria.fr/sibrahim/stragglers-detection-evaluation

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%

Table 5. Stragglers ratio on an homogeneous platform for the WordCount application, based on the definition of the usual execution time: *the usual execution time is the time for the X quantile of execution time.*

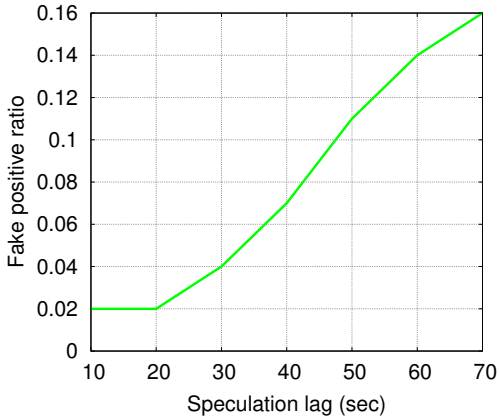

Fig. 3. Impact of *Speculative lag* on the *Fake Positive* ratio with *Default* straggler detection mechanism while running *WordCount* application.

Equation 11 ensures that a straggler is detected as a straggler and Equation 12 ensures that a non-straggler will not be detected as a straggler. Such a value only exists if $\varepsilon_{\text{setting}} \leq \frac{s_{\min}-1}{s_{\min}+1}$ (meaning that the variation due to settings is not too big). In fact, this says that $\varepsilon_{\text{guess}}$ should be lower than $\min\left(\frac{s_{\min}(1-\varepsilon_{\text{guess}})}{1.2} - 1, \frac{0.2-\varepsilon_{\text{guess}}}{1.2}\right)$. This provides a sufficient condition for the imperfection in determining the execution time.

It should be noted that in our case, we noticed that the threshold that we chose (0.5 quantile) allowed to determine closely the expected number of stragglers in homogeneous environment (see Table 5) as well as in all heterogeneous configuration scenarios $C_{i=1..4}$.

6.2 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, a late decision can delay the straggler detection and reduce the chance of the speculative copy to successfully finish.

In Hadoop, there is a parameter, named *Speculative lag*, which specifies the waiting time after the job begins to trigger the straggler detection. This value is set to 60 seconds by default. However, this static parameter does not work well with different applications having different runtime characteristics. Therefore, it is possible for a straggler to be detected too late that its copy has most likely no chance to finish before the original task. We call this *Fake Positive* detection. In the scope of this paper, we define a straggler is *Fake Positive* detected, if it is detected at the moment such that its remaining time is smaller than its *usual execution time*. We consider the *Fake Positive* as part of the *False Positive* detection. Thus, the presence of this metric reduces the value of the *Precision*.

$Cj = (p_1, p_2, p_3, p_4)$	Mechanism	Precision	Recall	Detection Latency (DL)	Fake Positive (FP)	Undetected Time (UT)
$C1 = (35, 35, 5, 25)$	D	0.64	0.48	0.55	0.05	2.10
	H	1.00	0.47	0.65	0.00	2.10
	L	0.83	0.61	0.44	0.04	1.80
$C2 = (25, 25, 25, 25)$	D	0.64	0.46	0.41	0.02	1.80
	H	1.00	0.33	0.57	0.01	1.80
	L	0.82	0.60	0.38	0.04	1.60
$C3 = (10, 10, 5, 75)$	D	0.22	0.41	0.49	0.07	1.90
	H	0.88	0.37	0.55	0.04	1.90
	L	0.24	0.55	0.48	0.06	1.80
$C4 = (5, 5, 0, 90)$	D	0.12	0.50	0.51	0.01	2.10
	H	0.98	0.38	0.53	0.02	2.10
	L	0.31	0.62	0.48	0.03	2.00

Table 6. The characteristics of the three straggler detection mechanisms when running in four different scenarios.

$$\text{Fake\_positive} = \frac{|\text{Fake\_pos}|}{|\text{True\_pos} + \text{False\_pos}|} \tag{13}$$

In order to understand the impact of the *Speculative lag* parameter on causing the *Fake Positive* detection, we conduct a set of experiments monitoring the *Fake Positive* ratio (calculated by the Equation 13). We run *Default* straggler detection while varying the *Speculative lag*. Figure 3 illustrates that the irrelevant default value 60 seconds of the *Speculative lag* parameter can result in a high *Fake Positive* ratio (0.14 in this case). 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. As the result, the *Precision* is reduced down to 0.36 (28% of reduction). 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 by 20 seconds. We believe that this Speculative lag configuration can assure a negligible impact of the Fake Positive ratio on our experimental results.*

6.3 Characterization of the straggler detection mechanisms

We first present the raw results of these experiments in Table 6 before discussing them. In the table, we present the characteristics of the three straggler detection mechanisms: *Default - D*, *LATE - L* and *Hierarchical - H*. The metrics that we use to characterize the mechanisms are: *Precision*, *Recall*, *Detection Latency - DL*, *Undetected Time - UT* and *Fake Positive - FP*. The results depict the characteristics of the three mechanisms in four different scenarios.

In terms of *Default* mechanism, it has quite low *Precision*. In the C_4 scenario, it has the *Precision* of only 0.12, which means there are 88% of detected tasks were not actual stragglers. Regarding the *Recall* metric, its *Recall* has relatively high values in most of the cases. However, the value is still fairly low in some cases (0.41 in the C_3 scenario). This observation suggests that it is potential to much more improve the *Default* mechanism, in both *Precision* and *Recall*.

Considering the *Hierarchical* mechanism, we firstly notice that it has very high *Precision*, up to 1.0 in most of the cases, while the *Precision* of *Default* is very low. On the other hand, it has fairly low *Recall* values compared to *Default*. 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 *LATE* strategy, its advanced detection mechanism results in a better *Precision* as well as *Recall* compared to *Default*. However, in the C_4 scenario, we record a *Precision* of 0.31, which implies that *LATE* mechanism also can still be more improved in terms of *Precision*.

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 usual execution time for the three mechanisms to detect the straggler. This average *Detection Latency* value implies that a speculative copy only has a chance to reduce the execution time of the straggler if the straggler takes at least 130-160% of the usual execution time to finish. This provides useful information for making the straggler handling decisions more efficient. Considering the *Undetected Time* metric, it specifies the impact of non-detected stragglers in causing the long-running tasks to the jobs. We notice that *Hierarchical* mechanism has a fairy close *Undetected Time* compared to *Default*. This is due to its design goal of targeting the potentially longest stragglers.

In brief, using our metrics, we can easily indicate the characteristics of the detection mechanisms. In addition, the results illustrate that the proposed Hierarchical mechanism successfully achieves its design goals. More importantly, our metrics show that there is still room for improving the detection mechanisms.

7 EVALUATION OF SPECULATIVE EXECUTION EFFECTIVENESS

In this section, we verify experimentally the mathematical intuition (Section 4). Precisely, we illustrate how the straggler detection characteristics can be used to indicate the performance and energy consumption of the speculative execution.

7.1 Methodology

For this set of experiments, we use the same infrastructure, platform and application setting as mentioned in Section 5. In this section, we discuss the importance of early speculative copy launching and present the methodology of providing early available resource.

Early launching speculative copies. By default, Hadoop and YARN trigger the speculative execution at the end of the execution, when there are available resources. 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 and duplicated [2, 22, 23]. Therefore, using this late straggler detection and handling cannot exactly reflect the effectiveness of the straggler detection mechanism. By triggering the straggler detection at different stages of the execution, we expect to provide the full information about the speculation efficiency in different scenarios. Moreover, launching the detected stragglers right after the detection instant will comprehensively show the impact of straggler detection mechanisms on the job execution.

Cluster configuration for resource availability. In order to provide the early available resource, we allocate a fraction x of the total resources for launching normal tasks. The speculative copies run on $1 - x$ of reserved resource. This $1 - x$ percent of cluster capacity is evenly reserved across the nodes. Which implies that each node will reserve $1 - x$ percent of its total capacity.

Setup. By varying the reservation resource ratio $x = \{100; 95; 90; 75; 50\}$, our goal is to cover as much as possible the diversity of the scenarios when having different resource quotas for early launching the speculative copies. At $x = 100$, there are no reserved resources for early copy launching. Thus, the copies can only start at the end of the execution. For the other values, the speculative copies are launched on the $1 - x$ percent reserved resource as soon as there are

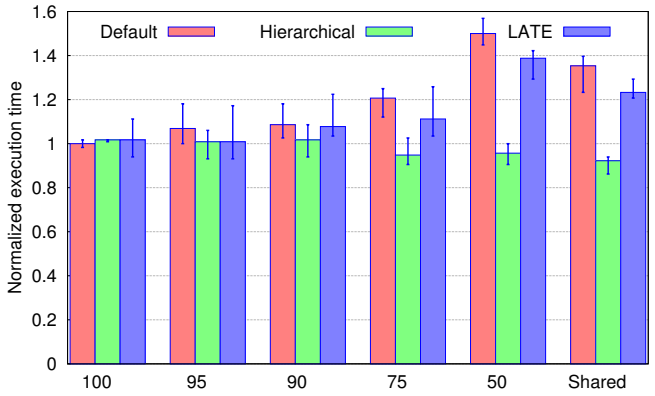

Fig. 4. Execution time comparison

stragglers detected. Moreover, a sharing resource reservation policy, named *Shared*, is also used in the evaluation where the normal task and speculative copies are sharing the free resource from the very beginning of the execution.

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 the scope of this paper, we present the results when running in $C_2 = (25, 25, 25, 25)$ scenario.

7.2 Impact of different resource reservation policies

First of all, we present the results in Figures 4, 5 and 6. The values in Figures 4 and 5 are normalized to the result when using *Default* mechanism in *100* reservation scenario. We can observe that the performance and energy consumption vary amongst different straggler detection mechanisms as well as different resource reservation ratios.

Regarding the 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 *Shared* resource reservation policy with *Hierarchical* straggler detection mechanism (see Figure 4) compared to the *100* scenario. In terms of energy consumption, *Shared* reservation policy can similarly result in a 6% of reduction. This illustrates the importance of providing early and enough resource for improving the efficiency of speculative execution.

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 resources to run and (ii) the normal tasks can take the remaining idle resource and maximize the resource utilization.

In brief, we can conclude that the resource reservation policy can strongly impact the performance and energy efficiency of speculative execution.

7.3 Evaluation of speculation using proposed metrics

In this section, we discuss why the existing metrics cannot provide the relevant explanation for the varied execution behaviors. Subsequently, we demonstrate that our metrics can easily interpret these behaviors.

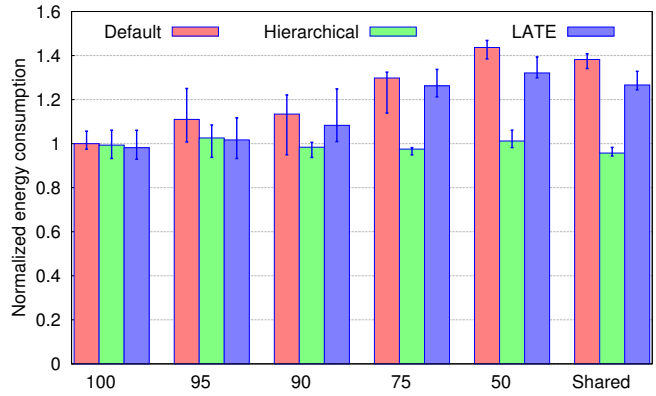

Fig. 5. Energy consumption comparison

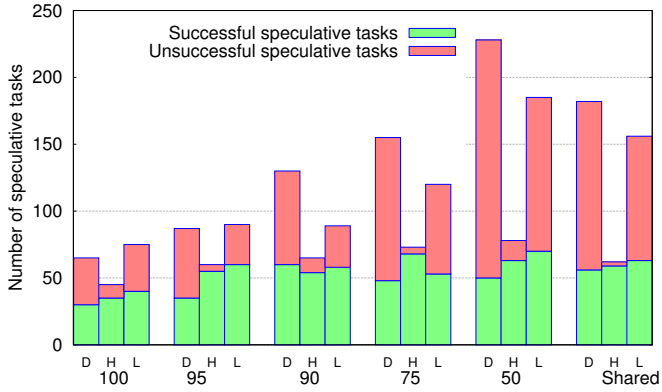

Fig. 6. Number of speculative copies

Misleading existing metrics. Regarding the existing metrics, we consider firstly the *number of speculative copies*. Comparing the results shown in Figure 4, 5 and 6, 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 straggler detection mechanisms' effectiveness. However, the results show that this metric again does not reflect the execution time reduction nor the energy consumption reduction. As an example, 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 and 6 at *90*, *50* and *Shared* scenarios).

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

Table 7. The characteristics of the three straggler detection mechanisms with different resource reservation ratios. The data are calculated by using the information of all the running tasks during the execution. The finish time of killed original tasks are estimated based on the moment they get killed and the progress they reach.

Interpreting the results with our metrics. Hereafter, we illustrate how to use our proposed metrics for understanding the impact of speculative execution on performance and energy consumption.

First, we provide the characteristics of the three straggler detection mechanisms in Table 7. 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 detection mechanism on the cluster's behavior.

As shown in Figure 4, 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 6). These unsuccessful copies compete with the normal tasks and, thus, lead to a significant degradation in performance (roughly 36% degradation in the case of *50*). These copies also result in a high amount of wasteful energy consumption (31% extra energy consumption with *50* policy).

Considering *LATE*, it results in a slightly lower number of unsuccessful copies (see Figure 6), 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 6). More importantly, as *Hierarchical* is designed to target the potentially longest stragglers, it again has mostly closely similar *Undetected Time* compared to *Default*. Thus, the stragglers miss-detected by *Hierarchical* have a small impact on prolonging the execution. Consequently, *Hierarchical* leads to an efficient execution in terms of both performance and energy consumption. Figure 4 and 5 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 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 of our evaluation metrics. Moreover, we have shown that the existing straggler detection mechanisms can be improved to achieve better performance and energy efficiency. Lastly, the Hierarchical detection mechanism can be used in practice for better mitigating the negative impacts caused by stragglers, with a high energy efficiency.

8 CONCLUSION

Speculative execution has emerged as a promising technique for mitigating stragglers. Improving the speculative execution efficiency requires a deep understanding of the straggler detection mechanism characteristics. Unfortunately, using the existing metrics is inefficient and sometimes can lead to imprecise interpretation for understanding the straggler detection effectiveness. In this paper, we demonstrate that these metrics can result in misleading information while evaluating the straggler detection effectiveness. Targeting this issue, we introduce a set of metrics dedicated for characterizing the straggler detection and understanding its inherent attributes. Besides, we present a mathematical intuition for connecting the proposed metrics to their execution characteristics, including performance and energy consumption.

By the means of experiment evaluation, we demonstrate the use of our metrics for characterizing different straggler detection mechanisms. The results show that there is room for improving the existing straggler detection mechanisms targeting higher efficiency. The characteristics, obtained using our metrics, can provide useful hints for improving the efficiency of speculative execution. In considering the future work, we plan to extend our work by using more complex applications (e.g., Cloudburst [13]). Moreover, we are interested in using the characteristics (e.g., *Precision*, *Recall* and *Detection Latency*) of the straggler detection mechanism to adjust the resource reservation ratio for launching speculative copies [27], targeting higher energy efficiency.

REFERENCES

[1] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*. 185–198.

[2] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the outliers in MapReduce clusters using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. 1–16.

[3] Guillaume Aupy, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. 2014. Checkpointing algorithms and fault prediction. *J. Parallel and Distrib. Comput.* 74, 2 (2014), 2048–2064.

[4] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. 2013. Adding Virtualization Capabilities to the Grid'5000 Testbed. In *Cloud Computing and Services Science*. Springer International Publishing.

[5] Qi Chen, Cheng Liu, and Zhen Xiao. 2014. Improving MapReduce performance using smart speculative execution strategy. *IEEE Trans. Comput.* 63, 4 (2014), 29–42.

[6] Jeffrey Dean. 2009. Large-Scale Distributed Systems at Google: Current Systems and Future Directions. In *Keynote speach at the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS'09)*.

[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[8] Ana Gainaru, Franck Cappello, and William Kramer. 2012. Taming of the Shrew: Modeling the Normal and Faulty Behaviour of Large-scale HPC Systems. In *Proceedings of IEEE 26th International Parallel Distributed Processing Symposium (IPDPS'12)*. 1168–1179.

[9] HDFS. 2016. The Hadoop Distributed File System. (2016). https://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html Accessed in July 2019.

[10] Shadi Ibrahim, Hai Jin, Lu Lu, Bingsheng He, Gabirel Antoniu, and Song Wu. 2012. Maestro: Replica-Aware Map Scheduling for MapReduce. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'12)*. 59–72.

[11] Shadi Ibrahim, Hai Jin, Lu Lu, Song Wu, Bingsheng He, and Li Qi. 2010. LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud. In *Proceedings of the 2010 IEEE International Conference on Cloud Computing Technology and Science (CloudCom'10)*. 17–24.

[12] Hai Jin, Shadi Ibrahim, Li Qi, Haijun Cao, Song Wu, and Xuanhua. Shi. 2011. The MapReduce Programming Model and Implementations. *Cloud computing: Principles and Paradigms* (2011), 373–390.

[13] Tien-Dat Phan, Shadi Ibrahim, Gabriel Antoniu, and Luc Bougé. 2015. On Understanding the Energy Impact of Speculative Execution in Hadoop. In *Proceedings of the 2015 IEEE International Conference on Data Science and Data Intensive Systems*. 396–403.

[14] Tien-Dat Phan, Shadi Ibrahim, Amelie Chi Zhou, Guillaume Aupy, and Gabriel Antoniu. 2017. Energy-Driven Straggler Mitigation in MapReduce. In *Proceedings of the 23rd International European Conference on Parallel and Distributed Computing (Euro-Par 2017)*. 385–398.

[15] The Apache Hadoop Project. 2018. (2018). http://hadoop.apache.org

[16] Asfandyar Qureshi. 2010. Power-Demand Routing in Massive Geo-Distributed Systems. In *Ph.D.dissertation, MIT*.

[17] Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. 2013. Hadoop's Adolescence: An Analysis of Hadoop Usage in Scientific Workloads. *Proceedings of the VLDB Endowment* 6, 10 (2013), 853–864.

[18] M Thottethodi, F Ahmad, S Lee, and TN Vijaykumar. 2012. Puma: Purdue MapReduce Benchmarks Suite. *Technical Report, Purdue University* (2012).

[19] MapReduce Tutorial. 2016. (2016). https://hadoop.apache.org/docs/r2.7.3/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html Accessed in July 2019.

[20] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13)*. 5:1–5:16.

[21] Huicheng Wu, Kenli Li, Zhuo Tang, and Longxin Zhang. 2014. A Heuristic Speculative Execution Strategy in Heterogeneous Distributed Environments. In *Proceedings of the 2014 6th International Symposium on Parallel Architectures, Algorithms and Programming*. 268–273.

[22] Huanle Xu and Wing Cheong Lau. 2013. Resource optimization for speculative execution in a MapReduce Cluster. In *Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP'13)*. 1–3.

[23] Huanle Xu and Wing Cheong Lau. 2014. Speculative Execution for a Single Job in a MapReduce-Like System. In *Proceedings of the 2014 7th IEEE International Conference on Cloud Computing*. 586–593.

[24] Huanle Xu and Wing Cheongu Lau. 2015. Task-Cloning Algorithms in a MapReduce Cluster with Competitive Performance Bounds. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems (ICDCS'15)*. 339–348.

[25] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. 10–10.

[26] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. 29–42.

[27] Amelie Chi Zhou, Tien-Dat Phan, Shadi Ibrahim, and Bingsheng He. 2018. Energy-Efficient Speculative Execution Using Advanced Reservation for Heterogeneous Clusters. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP 2018)*. 8:1–8:10.

Received July 2017; revised November 2018; accepted April 2019