Energy-Efficient Speculative Execution using Advanced Reservation for Heterogeneous Clusters

Amelie Chi Zhou, Tien-Dat Phan, Shadi Ibrahim, Bingsheng He

Energy-Efficient Speculative Execution using Advanced Reservation for Heterogeneous Clusters

Amelie Chi Zhou
Shenzhen University
Shenzhen, China

Tien-Dat Phan
Inria, Univ Rennes, CNRS, IRISA
Rennes, France

Shadi Ibrahim
Inria, IMT Atlantique, LS2N
Nantes, France

Bingsheng He
National University of Singapore
Singapore, Singapore

ABSTRACT

Many Big Data processing applications nowadays run on large-scale multi-tenant clusters. Due to hardware heterogeneity and resource contentions, straggler problem has become the norm rather than the exception in such clusters. To handle the straggler problem, speculative execution has emerged as one of the most widely used straggler mitigation techniques. Although a number of speculative execution mechanisms have been proposed, as we have observed from real-world traces, the questions of "when" and "where" to launch speculative copies have not been fully discussed and hence cause inefficiencies on the performance and energy of Big Data applications. In this paper, we propose a performance model and an energy consumption model to reveal the performance and energy variations with different speculative execution solutions. We further propose a window-based dynamic resource reservation and a heterogeneity-aware copy allocation technique to answer the "when" and "where" questions for speculative executions. Evaluations using real-world traces show that our proposed technique can improve the performance of Big Data applications by up to 30% and reduce the overall energy consumption by up to 34%.

1 INTRODUCTION

Many Big Data processing applications nowadays run on large-scale multi-tenant clusters. In such clusters, the heterogeneous hardware models and diverse application categories result in unavoidable performance variability. Despite that there have been many studies aiming to design improved task/job schedulers [10, 11, 22], the unexpected long tails in job execution still exist and have become the norm rather than an exception [8]. These heavy-tailed tasks, i.e., *stragglers*, can severely prolong the job execution time [23]. More importantly, those heavy tails can result in high resource and energy consumption [14] and hence adversely impact the resource utilization and energy efficiency of the cluster.

Much attention and efforts have been paid to mitigate the stragglers of Big Data processing applications, in order to improve their performance and reduce the energy consumption [3, 4, 23]. Amongst the proposed techniques, speculative execution emerged as one of the most widely used straggler mitigation techniques. In practice, it can reduce the average job execution time by 44% [5] compared to the case when speculative execution is disabled. With the speculative execution technique, it is important to decide when and where (which server(s)) to launch a speculative copy of each detected straggler. The answers to these two questions, namely *when* and *where*, can strongly affect the effectiveness of the speculative execution on both performance and energy efficiency.

Concerning the question of *when*, we analyzed the well-known CMU traces [16] to investigate the correlation between the earliness of launching speculative copies and achieving good straggler mitigation results. The earliness of launching copies is defined as $Earliness = 1 - \frac{StartTime_c - StartTime_s}{Average\ Execution\ Time}$, where $StartTime_c$ and $StartTime_s$ are the start time of speculative copies and stragglers, respectively. Average execution time stands for the average execution time of all tasks. We randomly select 200 out of the 1000 jobs in CMU traces and present the observed correlation in Figure 1. On the one hand, the *late* speculation handling (e.g., Hadoop's default scheduler) leads to poor straggler mitigation results as shown by the data plotted on the left top of the figure. This is mainly due to the late launches of copies, which can cause low performance improvement and high resource waste. Ren et al. [16] reported that many reduce tasks are speculated too late, resulting in wasteful speculative copies running for less than 10% of the tasks' average 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 and consume less resources. However, it is not true referring to data plotted on the right top of the figure. Some of these early copies, although are launched right after stragglers, have very bad performance where the execution times of the copies are at least 1.5x longer than the average task execution time. Thus, we conclude that the existing speculative execution techniques with a fixed solution (either early or late) to the "when" question is far from ideal.

Concerning the question of *where*, the heterogeneity problem in large-scale multi-tenant clusters can greatly affect the performance and energy efficiency of Big Data applications. However, this problem has not been well-discussed for speculative execution in existing work [1–4, 7, 18–20, 23]. Heterogeneity in multi-tenant clusters mainly comes from hardware heterogeneity and resource contentions. First, it is common to have different types of hardware in modern commodity-based clusters due to system upgrade for changing user requirements [23]. Second, the resource sharing among multiple users due to resource virtualization can lead to severe contentions and thus cause variant and unpredictable task execution time. Existing studies have demonstrated that those heterogeneities can lead to large performance variances [9, 24] to Big Data applications and affect the straggler handling results. In this paper, we focus on the performance variation of Big Data applications resulted from heterogeneity, while the cause of heterogeneity is not our concern. As has

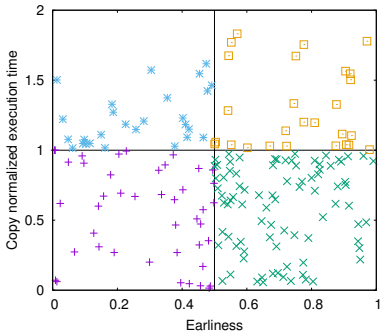


Figure 1: The correlation between the earliness of launching speculative copies and the average execution time of all successful tasks using CMU traces.

been shown in our evaluations, without considering the performance variation in speculative executions, the overall job execution time can increase 26% and the energy consumption increases 28%.

In this work, we take into consideration the deficiencies of existing studies in answering the "when" and "where" questions and introduce a dynamic and heterogeneity-aware speculative execution mechanism. Our goal is to optimize the execution time and energy consumption (i.e., skyline optimization) for Big Data processing applications in multi-tenant clusters by smartly handling stragglers at runtime. Specifically, we first propose a performance model and an energy consumption model to reveal the performance and energy variations with different task and copy allocations, in order to well balance the performance and energy optimization goals. With these models, our heterogeneity-aware copy allocation technique can decide the best locations of launching speculative copies, hence answering the "where" question. Second, we propose a window-based reservation technique to dynamically decide the best timing of launching speculative copies, hence answering the "when" question, by considering the benefits of allocating speculative copies onto the available free slots in the current window. We design a trace-driven simulator to emulate Big Data processing in multi-tenant clusters. Evaluations with real-world traces [16] show that, compared to state-of-the-art speculative execution mechanisms, our technique can improve the performance of Big Data applications by up to 30% and reduce the energy consumption by up to 34%. Overall, we improve the energy efficiency of Big Data applications by up to 130% on heterogeneous clusters. We make the following contributions.

- A performance model and an energy consumption model for Big Data applications to reveal the performance and energy variations with different speculation solutions.
- A window-based slot reservation technique to decide the best timing of launching speculative copies at runtime.
- Evaluations with real-world traces show that our technique can greatly improve the performance and energy efficiency of Big Data applications compared to state-of-the-art speculative execution mechanisms.

The rest of this paper is organized as follows. We introduce the background and related work of this paper in Section 2. We discuss the deficiencies of existing speculative execution mechanisms in Section 3. We give an overview of our design in Section 4 and introduce more details of the proposed techniques in Section 5. We evaluate our design with a trace-driven simulator in Section 6 and conclude this paper in Section 7.

2 BACKGROUND AND RELATED WORK

In this section, we introduce the background and related work on straggler mitigation in Big Data processing systems.

2.1 Stragglers in Big Data Processing Systems

In large-scale Big Data processing systems, performance variations can happen due to many reasons such as heterogeneous hardware, locality problems and resource contention between concurrent jobs [3, 10, 23]. In the following, we do not differentiate the heterogeneities caused by hardware or contention, but use the performance variation of Big Data processing to quantify the overall heterogeneity of a system. Performance variations may lead to the case where some tasks take longer time to finish than the average task execution time. Those slow tasks, i.e., stragglers, can severely degrade the job performance and cause resource waste.

As observed from the CMU traces, many jobs (over 30% of jobs) consist of thousands of tasks, which fully occupy the whole cluster capacity. This high utilization causes severe resource contentions and a larger number of stragglers in the system. Hence, it is important to mitigate stragglers on those highly utilized systems.

2.2 Existing Straggler Mitigation Techniques

Straggler mitigation has attracted much attention due to its high impact on the performance of Big Data applications [23]. We introduce related studies in the following two categories.

Straggler detection. The default straggler detection mechanism adopts the *progress score* (defined as the ratio of processed data over the total input data of a task) to identify slow tasks. Later, Zaharia et al. [23] found that using progress score alone is not enough and *network heterogeneity* can cause 80% more of unnecessary speculation. Accordingly, they propose to adopt the progress rate of a task, which is defined as the percentage of input data processed per second. It alleviates the launching of unnecessary speculation caused by network heterogeneity. Chen et al. [4] introduce new straggler mitigation strategy that considers not only the progress rate but also the data transfer rate of the tasks to detect slow tasks. It also takes into account the data skew for determining the stragglers.

Straggler handling. Dean et al. [5] adopt speculative execution to handle stragglers. With speculative execution, a speculative copy is launched for a straggler on a different machine. Upon the finish of either the straggler or the copy, the other gets killed. It is shown that speculative execution can reduce job execution time by 44% in Google cluster [12]. Ananthanarayanan et al. [3] introduce Mantri, a resource-aware straggler handling mechanism. Mantri schedules speculative tasks only when there is a fair chance to reduce the tasks' execution time with a low resource consumption. Considering that the majority of jobs in production Hadoop clusters are small jobs, Ananthanarayanan et al. [1] present Dolly, a new approach to handle stragglers by launching multiple copies (i.e., clone) of tasks belonging to small jobs. After the first clone of a task is completed, the other clones are killed to free the resources.

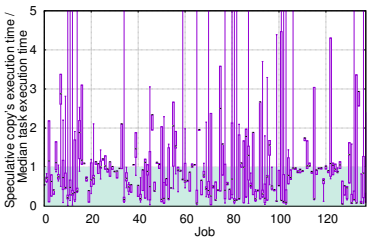


Figure 2: Performance variations of speculative copies in the CMU traces.

In this paper, we focus on designing new speculation mechanism to better mitigate stragglers in highly utilized clusters.

3 DEFICIENCIES OF EXISTING SPECULATIVE MECHANISMS

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

3.1 When to launch? A fixed solution is far from ideal

Ren et al. [16] have shown that the late launching of speculative copies can result in a large number of unsuccessful speculative copies, which are launched too late and are killed before finishing. As a result, many studies aim to introduce early speculation handling solutions [3, 4, 18–20]. However, by analyzing the CMU traces [16], we discover that early speculation handling does not necessarily lead to better straggler mitigation results, as shown in Figure 1. This is mainly because the early allocation can happen at the wrong time. For example, it is possible that the earliest available slot is not suitable for the copy to be launched and leads to a long copy execution time. As shown in Figure 2, we have observed a large variation in speculative copy runtime. This shows that a fixed solution to the question of when to launch a speculative copy is not good enough. The decision has to be made at runtime and considered together with the question of where to launch the copy.

3.2 Where to launch? Heterogeneity has to be considered

Concerning the question of *where* to launch the speculative copies, we need to study how different copy allocation solutions differ on the task performance and energy efficiency. Since the multi-tenant clusters are usually shared by multiple concurrent jobs, the resource contention can lead to performance degradation when collocating tasks with similar critical resource requirement on the same node. For example, Yildiz et al. [21] 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. For each job, we compute the ratios between copy execution time and the median task execution time. The *box-and-whisker* diagrams in Figure 2 show the distribution (the minimum, 25^{th} quantile, median, 75^{th} quantile and the maximum values) of these ratios per job. We find that the difference between the execution time of two copies in the same job can be more than 10x. Regarding the energy consumption, previous studies [14, 15] have shown that there exists a trade-off between the performance and energy consumption when allocating speculative copies to nodes with different numbers of running tasks.

In conclusion, it is important to consider the impact of heterogeneity on performance and energy consumption when making speculative copy allocation decisions. However, this might not be effective if it is done passively as shown in [15]. Hence, this motivates our window-based reservation technique.

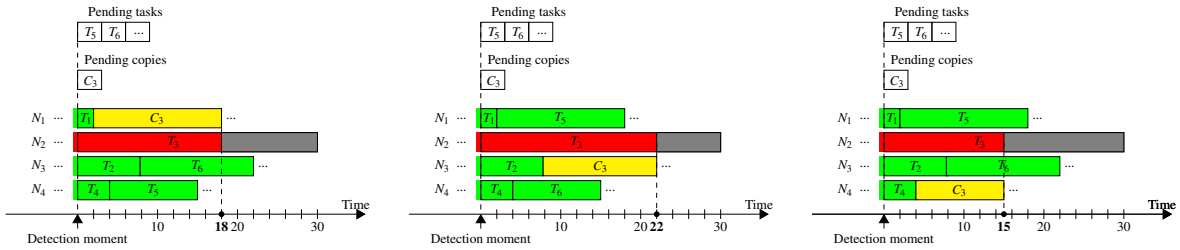
3.3 A motivating example

To better explain the two deficiencies mentioned above, we present an illustrative example as in Figure 3. 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 on the first freed slot in node N_1 (see Figure 3a). The execution time of C_3 on N_1 is 16 and the finish time of T_3 is 18. With late copy allocation, C_3 is launched after all normal tasks in the queue (i.e., T_5 and T_6) have been scheduled (see Figure 3b). Although the execution time of C_3 on N_3 is shorter than on N_1, the finish time of T_3 is late (i.e., 22) due to the late start of the copy. In comparison, we propose a window-based copy allocation mechanism, which adaptively decides the start time and location of C_3 as shown in Figure 3c. The detailed design of this mechanism is introduced in later sections. 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 window-based mechanism also achieves the best result. The resource consumption (in terms of $\#slot \times t$) of the speculative executions with early, late and window-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 efficiency.

4 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 and the number of slots per node equals to the number of cores in the node. Assume the job sizes are large and the cluster is highly utilized. Assume a fraction α 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 efficiency of each job.

To achieve this goal, we propose a task-level speculative scheduler which works together with the built-in job schedulers of Hadoop, such as 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 original tasks and speculative copies, in order to satisfy the performance and energy optimization objectives. The main advantage of designing our speculative scheduler at the task-level is to separate the task



(a) Early copy allocation **(b) Late copy allocation** **(c) Window-based copy allocation**

Figure 3: An example with different speculative mechanisms.

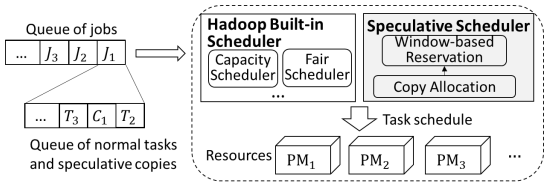


Figure 4: Design Overview

scheduling problem from job scheduling and thus greatly reduce the solution space and problem solving latency. As a result, we are able to optimize the speculative execution at runtime.

Our speculative scheduler has two main components, namely the window-based resource reservation for speculative copies (Section 5.1) and the resource selection component which decides where to schedule speculative copies (Section 5.2). Figure 4 shows an overview of our design. With the Hadoop built-in scheduler working at job-level, in the following, we discuss the details of the two components using a single job for simplicity. However, note that our design works for both single job and multiple jobs scenarios.

5 PROPOSED TECHNIQUES

We divide the speculative handling problem 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 two sub-problems.

5.1 Window-based Resource Reservation

Existing straggler handling mechanisms are triggered once there are stragglers detected and free slots for launching speculative copies. However, in highly utilized clusters, the resource is not always available for launching speculative copies and reservations have to be made in advance to make sure the speculative copies get executed. As has been demonstrated by existing studies [17], static reservation with fixed sizes can easily lead to over- or under-reservation problems. In this paper, we propose a dynamic resource reservation technique, which considers the free resources in the near future (i.e., a time window) to make good resource reservations at runtime. Specifically, we define a time window size w, which is set to be shorter than the average execution time of normal tasks. The average task execution time can be calculated using runtime information collected from the last time window or all previous windows, according to the variance of the system. At the beginning of each time window, we perform straggler detection and store the detected stragglers in a set S. We also have the set of normal tasks to be scheduled 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 running tasks and thus have the estimated number and starting time of freed slots. 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 performance and energy consumption 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 choosing n slots from V. With the large number of stragglers in Big Data applications, the time for exhaustively searching the solution space is prohibitively long. Thus, we propose the following heuristic to quickly obtain a good solution to the problem. First, we order the stragglers according to their *criticalness*, i.e., the residual time for a straggler to finish. The criticalness of a straggler is defined as $(1-progress)\cdot\frac{duration}{progress}$, where $\frac{progress}{duration}$ represents the speed of the straggler. More critical stragglers have better potential of saving resources and thus are given higher priorities when being allocated. A speculative copy is only launched when the residual time of the straggler is longer than the expected execution time of the copy. Second, given the set of ordered stragglers, we sequentially search for a good slot from V for each straggler in S to run the copies, as shown on the left side of Figure 5. When searching for a good slot for each straggler, we consider the impact of different allocation choices to the performance and energy consumption of the job. This searching is processed using our heterogeneity-aware copy allocation technique. As will be introduced in the next subsection, the time complexity of this technique is $O(|V|\times\min\{|S|,|V|\})$. Third, after making copy allocation decisions for all stragglers, the non-reserved slots in V are used for executing the normal tasks in T. The normal tasks are scheduled using the user-defined Hadoop built-in scheduler, as shown on the right side of Figure 5.

The benefits of our design are two-fold. First, compared to real-time reservation methods, it provides the ability to look-ahead when reserving resources and thus offers 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 up-to-date runtime information from the latest time window, and thus are more likely to lead to good results.

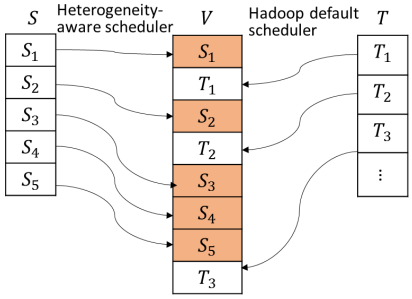


Figure 5: An example of window-based resource reservation. In the current window, S is the set of detected stragglers ordered by criticalness, V is the set of slots to be freed and T is the set of normal tasks to schedule.

Discussion on window size: We limit the window size to be shorter than the average task execution time is to perform one-step look-ahead when estimating the number of free slots. If the window size is too large, the $|V|$ and $|S|$ values are likely to be large. 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 small, the number of allocation choices in one time window is also small. Thus, the overall optimization result can be less effective. In the evaluation section, we dedicate one part to illustrate the impact of different window sizes on the space of choices.

5.2 Heterogeneity-Aware Copy Allocation

Given a list of stragglers S (S_i, where $i = 0, \cdots, k-1$) ordered by criticalness 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 resulted from different copy allocation solutions. Relying on those model-based estimations, we propose a copy allocation heuristic to smartly map speculative copies onto different slots.

5.2.1 Performance Model. The performance of a task (either a normal task or a speculative copy) on a slot can be affected by many factors. Existing performance models [15], which usually predict the task performance using offline profiling, can easily lead to prediction inaccuracies in heterogeneous clusters. In this paper, we mainly consider three factors, namely resource contention, data locality and the capability of slots, which are usually the major impacting factors to task performances. We can divide the execution time of a task into the following three components:

$$t = t_{req} + t_{cont} + t_{data} \tag{1}$$

where t_{req} is the shortest time required to execute the task (with local data and no contention), t_{cont} is the prolonged execution time 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 1. First, for tasks (i.e., Mappers) running on the same slot, t_{req} is almost the same, assuming that the input data is evenly assigned to different tasks. 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 $t_{data} = \frac{D}{B}$, 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 normal 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 discussed in existing studies [23], 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 [23]. 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 currently S_i is the first straggler in S that has not been handled and 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:

$$\Delta t_{ij} = t_{S_i} + t_{V_k} - t_{C_i} - t_{V_j} \tag{2}$$

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 C_i and S_i on slot V_j and V_k, respectively.

5.2.2 Energy Consumption 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^{total}_{static} and the dynamic power consumption P^{total}_{dyna}. 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 [6]. 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 Gbps bandwidth. Then, P^{total}_{dyna} can be defined as the sum of the dynamic power consumption of all active resources as shown below.

$$P^{total}_{dyna} = P^{cpu}_{dyna} + P^{mem}_{dyna} + P^{net}_{dyna} \tag{3}$$

The CPU dynamic power consumption of a running node mainly depends on the number of tasks simultaneously running on the node. We use P_{dyna} to denote the power consumption for one active core and n to denote the number of tasks running on the node. Thus, P^{cpu}_{dyna} can be modeled as below.

$$P^{cpu}_{dyna} = \begin{cases} n \cdot P_{dyna} & \text{for } 0 \le n \le c \\ c \cdot P_{dyna} & \text{for } c < n \end{cases} \tag{4}$$

The dynamic power consumption of the memory is strongly related to the number of data accesses per second [6]. It is 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_{dyna}^{mem} is modeled as the sum of the dynamic memory power of all running tasks [6]. P_{dyna}^{net} can be modeled in a similar way, using the total amount of data transfer of each task [6]. Note that if a task achieves data locality, the dynamic network power consumption of the task is zero.

Similarly, the static power consumption is composed by the idle power consumption of CPU, memory and network resources [6]. We model P_{static}^{total} as the sum of P_{static}^{cpu}, P_{static}^{mem} and P_{static}^{net}, which are considered as architecture constants and can be estimated with hardware specifications or measured through profiling [6]. Thus, the total power consumption P of a running node can be modeled as the sum of P_{static}^{total} and P_{dyna}^{total}, and the power consumption of a slot with a running task can be calculated as the difference of the node power consumption before and after 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 throughput to the power consumption, namely $EE = 1/E$.

Executing a new copy consumes more energy while at the same time saves energy due to 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.

$$\Delta E_{ij} = P_k \cdot \Delta t_{ij} - P_j \cdot t_{C_i} \tag{5}$$

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.

5.2.3 Copy Allocation Heuristic. Given the performance and energy models, we have two metrics $(\Delta t_{ij}, \Delta E_{ij})$ to decide the fitness of each map from straggler S_i to slot V_j. Given any straggler, there can be $m+1$ different copy allocation choices assuming that there are m non-reserved slots in V (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 choose the one with the best performance and energy metrics as the allocated slot. 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. If a slot cannot be found for a straggler S_i, we propose to opportunistically reschedule S_i (Section 5.2.4) with the stragglers scheduled before it, in the hope of optimizing the overall performance and energy consumption. Algorithm 1 presents the general flow of our copy allocation heuristic. The worst-case complexity of this heuristic is $O(|V| \times \min\{|S|, |V|\})$.

5.2.4 Opportunistic Rescheduling. When the *best_map* is not found for a straggler S_i, we try to jointly reschedule the previously scheduled stragglers together with S_i, in order to optimize the overall performance and energy consumption of S_i and the rescheduled straggler. For example, in Figure 5, if no *best_map* can be found for S_2, it is possible that the slot favored by S_2 has already been taken by S_1. Thus, scheduling the copy of S_1 to another slot and allocating the freed slot to S_2 may be better than only launching a copy for S_1. However, when the number of previously scheduled stragglers is high, considering all those candidates for rescheduling is clearly too time consuming. Thus, we propose to select k stragglers that are the most likely to benefit from rescheduling.

Algorithm 1 Speculative copy allocation heuristic.

```
while S is not empty do
    straggler i is the head of S;
    best_fitness = (0,0);
    for slot j in V do
        calculate Δt_ij and ΔE_ij using Equation 2 and 5;
        if (Δt_ij, ΔE_ij) > best_fitness then
            best_map = j;
            best_fitness = (Δt_ij, ΔE_ij);
    if best_map is not found then
        select S' as the set of stragglers to be jointly rescheduled with straggler i;
        reschedule(S', i);
    else
        launch a copy of straggler i in slot best_map;
    remove straggler i from S;
```

Selecting k stragglers: For each straggler S_j $(j < i)$, we define a metric $score_j$ to indicate the potential benefits gained from jointly rescheduling S_j and S_i. $score_j$ is affected by two factors, namely 1) the closeness between the criticalness of S_i and S_j and 2) the preference of S_i on the slot scheduled to S_j. As more critical stragglers have higher potential of saving resources, the first factor indicates the possibility of improving the performance and energy consumption of S_i while not sacrificing too much of those of S_j. We define the closeness as $score_j^1 = \frac{ct_j - ct_i}{ct_j}$, where ct_j and ct_i stand for the criticalness of S_j and S_i calculated in the resource reservation step, respectively. The second factor decides how much benefits S_i can gain by taking over the slot originally assigned to S_j, denoted as V_k. We calculate the benefits on time and energy according to Equation 2 and 5, namely $score_j^2 = \frac{\Delta t_{ik} - \Delta t_{jk}}{\Delta t_{jk}}$ and $score_j^3 = \frac{\Delta E_{ik} - \Delta E_{jk}}{\Delta E_{jk}}$. Overall, $score_j = score_j^1 + score_j^2 + score_j^3$.

We select the top-k stragglers with the highest scores in a set S'. Note that we only select stragglers with positive $score_j^2$ and $score_j^3$ values and the straggler selection is triggered only when $k < i - 1$. Based on sensitivity study, we set k to 3 by default to balance the performance and energy improvements and optimization overhead.

Joint rescheduling: Given the set S', for each selected straggler S'_j, we assign the slot of S'_j to S_i and search for the *best_map* slot from the idle ones for S'_j using the same way as the copy allocation heuristic. We adopt the first found solution as the rescheduling solution. If no solution can be found, no copy is launched for S_i.

6 EXPERIMENTAL EVALUATION

6.1 Setup

Trace-based simulator: Our experiments are conducted with a trace-driven simulator implemented in Java. Our simulator emulates a Big Data processing cluster. There are two main components in the simulator, namely nodes and jobs.

Each node is configured with a specific number of slots, which represents the number of tasks that can concurrently run on the node. When multiple tasks are running at the same time, the CPU, memory and network resources of the node are distributed evenly to those tasks. The same task on different nodes has different performance. This is to emulate the hardware heterogeneity. A job component 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 [16]. For each job, its tasks have specific resource usage characteristics (e.g., CPU, memory and network usages).

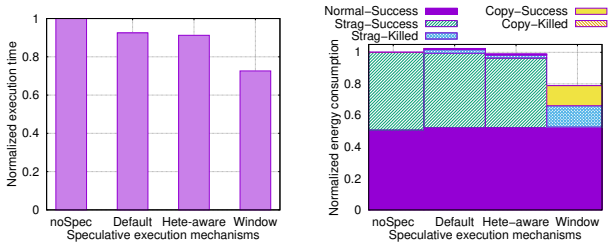
There are three important parameters in our simulation, namely the resource contention degree, hardware heterogeneity degree and straggler ratio. Resource contention degree indicates how much resource contention can affect the job performance. For example, a contention degree of 2.0x means that the execution time of a task becomes twice of its normal execution time due to resource contention. Hardware heterogeneity degree indicates how different the same task can perform on different nodes. For instance, a heterogeneity degree of 3.0x specifies that the same task can have 300% variation in execution time on different nodes across the cluster. The straggler ratio represents the ratio of stragglers among all the tasks. We extract simulation data from the CMU traces collected in January 2012 from a Hadoop production cluster [16]. During the 30 days of execution, more than 1500 jobs (1.5 million tasks) were submitted to the cluster by 78 different users.

Comparison: We compare our reservation-based speculative execution method (denoted as *Window*) with the following methods: (1) *noSpec* means that the speculative execution is disabled. (2) *Default* is the speculative execution mechanism currently used in Hadoop. It allocates speculative copies at the end of job execution, when all normal tasks are launched. (3) *Hete-aware* also allocates speculative copies on available resources at the end of the job execution. However, it takes into account the impact of both the performance and energy consumption of different slots. It uses the same copy allocation heuristic as shown in Algorithm 1.

Configuration: We set the resource contention to 2.5x, straggler ratio to 0.2, hardware heterogeneity to 3.0x and window size to 0.2 by default. We study the impact of these parameters through sensitivity studies. In all experiments, we compare the performance (i.e., makespan) and energy consumption of each job obtained by different mechanisms. All results are normalized to those of *noSpec* if not otherwise specified.

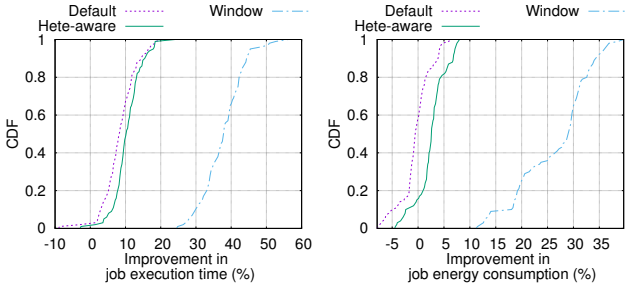
6.2 Comparison Results

Figure 6 shows the normalized average job performance and energy consumption results obtained by all compared mechanisms. We have the following observations. First, as shown in Figure 6a, *Window* obtains the best performance results among all compared mechanisms. Specifically, *Window* reduces the average job execution time by 27%, 22% and 20% compared to *noSpec*, *Default* and *Hete-aware*, respectively. This is mainly due to the window-based resource reservation technique which launches speculative copies at the most appropriate time (e.g., comparing *Window* with *Hete-aware*). Our heterogeneity-aware copy allocation technique can improve the performance of



(a) Normalized execution time **(b) Energy consump. breakdown**

Figure 6: Overall comparison results using different speculative execution mechanisms.



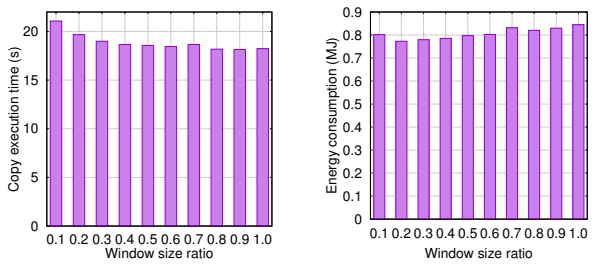
(a) CDF of execution time improvement **(b) CDF of energy consumption improvement**

Figure 7: CDF of execution time and energy consumption improvements of different speculative execution mechanisms with respect to *noSpec*.

speculative copies. *Window* reduces the average execution time of copies by 48% and 41% compared to *Default* and *Hete-aware*, respectively. Second, Figure 6b shows that, *Window* can significantly reduce the average energy consumption of each job by 21%, 23% and 20% compared to *noSpec*, *Default* and *Hete-aware*, respectively. We breakdown the energy consumption according to five task categories, namely successfully finished normal tasks (denoted as *Normal-Success*), successfully finished stragglers (denoted as *Strag-Success*), killed stragglers (denoted as *Strag-Killed*), successful speculative copies (denoted as *Copy-Success*) and killed speculative copies (denoted as *Copy-Killed*). *Window* greatly reduces the total energy consumption of stragglers thanks to the early launch of speculative copies. In comparison, many stragglers of the compared mechanisms are already finished when launching speculative copies. These results emphasize the importance of early speculative copy allocation to the performance and energy optimization in Hadoop clusters.

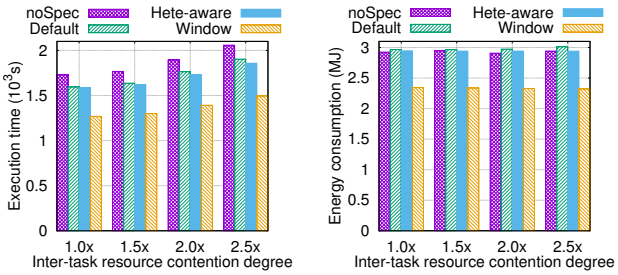
To better understand the outperformance of *Window*, we further study the performance and energy consumption results at the job-level. Figure 7a and 7b depict the CDF of *Window*'s improvements over the other mechanisms on execution time and energy consumption at the job-level. We find that, when looking at each individual job, *Window* still shows good improvement over the other mechanisms. For example, *Window* obtains over 30% reduction in job execution time for more than 90% of the jobs and reduces more than 20% energy consumption for 80% of the jobs compared to *noSpec*.

An interesting finding is that, *Default* and *Hete-aware* can sometimes perform even worse than *noSpec* in both execution time and



(a) Copy execution time **(b) Total energy consumption**

Figure 8: Sensitivity study on the window size parameter.



(a) Average job execution time **(b) Total energy consumption**

Figure 9: Sensitivity study on resource contention degree.

energy consumption for some jobs. This is mainly due to the late launching of speculative copies, which not only prolongs the execution time of stragglers and hence increases job execution time, but also wastes more energy on the killed speculative copies.

To examine the combined improvement of performance and energy consumption, we evaluate the energy efficiency of the compared speculative execution mechanisms. *Window* obtains the best energy efficiency result. Specifically, it increases the energy efficiency by 75%, 70% and 61% compared to *noSpec*, *Default* and *Hete-aware*, respectively. With our heterogeneity-aware copy allocation technique alone, *Hete-aware* increases the energy efficiency by 14% and 9% compared to *noSpec* and *Default*, respectively.

In conclusion, dynamically reserving the right resources can significantly improve the performance and the energy consumption (energy efficiency) of speculative executions.

6.3 Sensitivity Study

We first measure the impact of window size in *Window* on the average copy execution time and energy consumption. Then we evaluate the impact of three system factors, namely resource contention, straggler ratio, and hardware heterogeneity, on the effectiveness of the three speculative execution mechanisms.

6.3.1 Window Size. Window size is an important parameter to *Window*. We vary it from 0.1x to 1.0x of the job's fastest task to study its impact on performance and energy efficiency.

Figure 8 shows the average copy execution time and total energy consumption under different window sizes. With the increase of window size, the copy execution time decreases. This is because with a large window size, *Window* can have more options for the next reservation. Thus, speculative copies have higher chance of executing on relevant slots. However, when the window size increases, the energy consumption also increases in most of the cases. This is mainly because a large window size makes the straggler handling less frequent. Thus, stragglers can run longer and consumes more extra energy. These observations are consistent with our analysis in Section 4. Considering the trade-off between performance and energy consumption, we set the window size to 0.2 by default.

6.3.2 Resource Contention. We vary the contention degree from 1.0x (no contention), 1.5x, 2.0x to 2.5x (severe contention) to simulate the scenario of running different types of applications in the cluster. Figure 9 shows the results of the four methods under different contention degrees. We make the following observations.

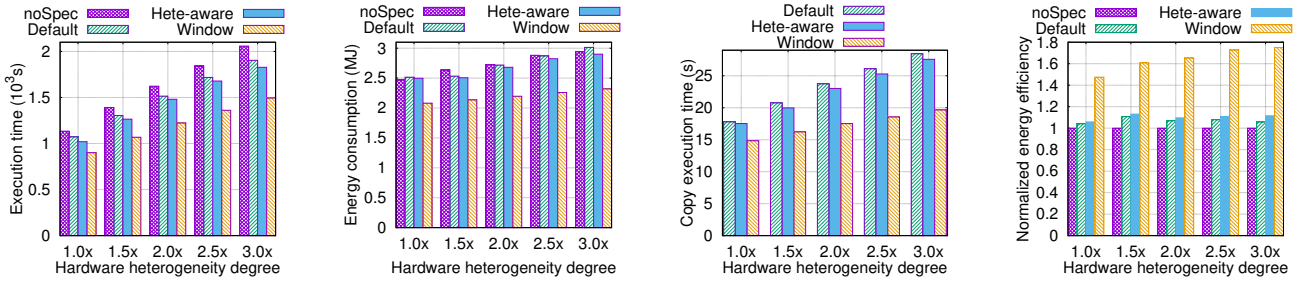
First, when the contention degree increases, the performance improvement of *Window* over the other mechanisms also increases. For example, when the contention degree increases from 1.0x to 2.5x, the reduction on average job execution time increases from 26% to 29%, 20% to 24% and 19% to 22% when comparing *Window* with *noSpec*, *Default* and *Hete-aware*, respectively. Similar trend has also been observed for the energy consumption results. Second, when the contention degree increases, the improvements of *Hete-aware* over *noSpec* and *Default* in performance and energy consumption also increase. For example, when the contention increases from 1.0x to 2.5x, the reduction on average job execution time increases from 8% to 11% when comparing *Hete-aware* with *noSpec*.

These results indicate that our proposed speculative execution mechanism works well with diverse resource contention degrees.

6.3.3 Hardware Heterogeneity. Hardware heterogeneity is a widely recognized problem in many Big Data processing systems [13]. 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. Figure 10 shows the results of the compared mechanisms under different heterogeneity degrees.

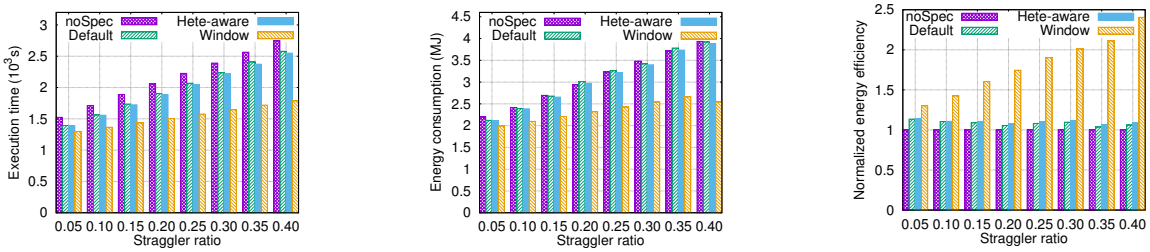
As the hardware heterogeneity degree increases, the straggler problem becomes more severe and *Window* is able to obtain higher improvement in performance and energy consumption compared to the other mechanisms. For example, compared to *Default*, the reduction on average job execution time obtained by *Window* increases from 16% to 22% and the total energy consumption from 17% to 23%. With a closer look at the execution of speculative copies (Figure 10c), we find that the average copy execution time increases much slower in *Window* than in *Default* and *Hete-aware*. *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. In contrast, benefiting from the dynamic resource reservation, *Window* has 40 free slots on average and thus has a better chance of allocating slots with high performance and low power consumption for speculative copies.

Overall, *Window* obtains much higher energy efficiency (up to 75%) compared to the other mechanisms and the improvement increases with the increase of hardware heterogeneity as shown in Figure 10d. This means our speculative execution mechanism works well in commodity clusters with high hardware heterogeneity.



(a) Average job execution time **(b) Total energy consumption** **(c) Average copy execution time** **(d) Normalized energy efficiency**

Figure 10: Sensitivity study on the hardware heterogeneity degree.



(a) Average job execution time **(b) Total energy consumption** **(c) Energy efficiency**

Figure 11: Sensitivity study on the straggler ratio.

6.3.4 Straggler Ratio. By analyzing CMU traces, we observe that the straggler ratio varies from 0.0 to 0.40 per job. In this evaluation, we vary the straggler ratio from 0.05 to 0.40 to study the behavior of the compared mechanisms.

As shown in Figures 11, with the increase of straggler ratio, the improvement of *Window* over the other mechanisms gets higher. For example, *Window* reduces the average job execution time by 6.5%–30% and the energy consumption by 6%–34% compared to *Hete-aware*. *Window* significantly increases the overall energy efficiency when the straggler ratio is high. The increase can be up to 140%, 135% and 130% compared to *noSpec*, *Default* and *Hete-aware*, respectively. This demonstrates that our algorithm can work well in systems with serious straggler problems.

6.4 Evaluation on Underutilized Clusters

Although our speculative execution mechanism is mainly designed for highly utilized clusters, we show that it can also work well as state-of-the-art speculative execution mechanisms on clusters with low utilization.

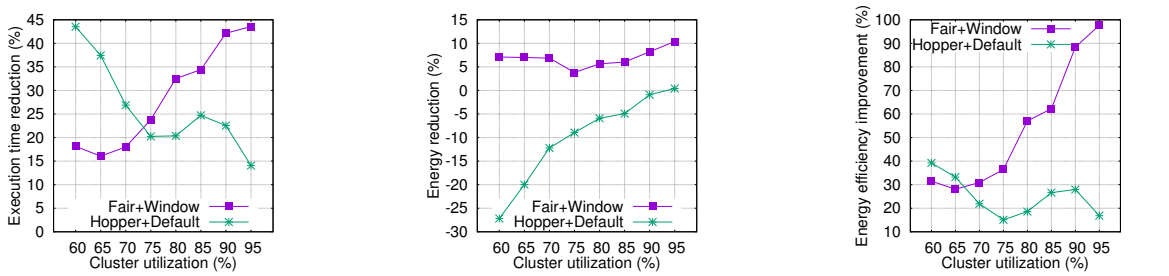
Specifically, we vary the system utilization from 60% to 95%. We take *Window* as a task scheduler and compare it with the default task scheduler of Hadoop, namely *Default*. We adopt two job schedulers for comparison, namely *Fair* [22] and *Hopper* [17]. *Fair* is the *de facto* scheduler in MapReduce production clusters [22]. It allocates free slots evenly among all running jobs. *Hopper* is a recent speculation-aware job scheduler, which is designed specifically for clusters with low utilization. We compare results of using *Window* on top of the *Fair* scheduler (denoted as *Fair+Window*), using *Default* on top of Hopper (denoted as *Hopper+Default*) and using *Default* on top of *Fair* (denoted as *Fair+Default*).

Figure 12 presents the improvements of the performance and energy results of *Fair+Window* and *Hopper+Default* over those of *Fair+Default*. We have the following observations. First, comparing *Fair+Window* with *Fair+Default*, *Fair+Window* is able to obtain better execution time and energy consumption/efficiency results under all system utilization setting. Also, the improvement increases with the increase of cluster utilization. This is mainly due to the in-advance resource reservation of *Window* which works especially well in high-utilized clusters.

Second, comparing *Fair+Window* with *Hopper+Default*, we find that *Fair+Window* obtains similar energy efficiency results as *Hopper+Default* when the system utilization is low ($< 75\%$) and obtains much higher energy efficiency when the system utilization is high ($\geq 75\%$). Note that *Hopper* reserves more than one slots ($2/\beta$ of the total number of normal tasks and we set $\beta = 1.5$) for launching speculative copies when system utilization is low, thus it can obtain good execution time results with extra energy consumption. This is consistent with our observation in Figure 12a and 12b at low system utilization. When there are not enough free slots to allocate $2/\beta$ copies for all stragglers, Hopper allocates free slots to shortest jobs first. As a result, long jobs have to wait until free slots are released and the straggler problem remains unsolved. This explains the observation in Figure 12a where the execution time reduction of *Hopper+Default* decreases when system utilization gets high.

7 CONCLUSION

In many multi-tenant clusters, the straggler problem has become the norm rather than the exception for Big Data applications. Existing studies have proposed a number of straggler handling techniques to address this important problem using speculative executions. However, we have observed from real-world traces that those techniques



(a) Execution time reduction **(b) Energy consumption reduction** **(c) Energy efficiency improvement**

Figure 12: Comparison results on clusters with different utilization degrees.

cannot well address the straggler problem due to inappropriate decisions on "when" and "where" to launch speculative copies. In this paper, we propose a window-based slot reservation technique and a heterogeneity-aware copy allocation technique to answer the "when" and "where" questions for speculative copies. Our techniques are guided by a performance and an energy consumption model to accurately reveal the system performance and energy variations with different speculative execution solutions. Evaluations using real-world traces show that our proposed techniques can greatly improve the performance of Big Data applications and reduce the overall energy consumption. As a future work, we plan to implement our mechanism into real systems to demonstrate its effectiveness.

ACKNOWLEDGMENT

The corresponding author is Shadi Ibrahim (shadi.ibrahim@inria.fr). This work is supported by the ANR KerStream project (ANR-16-CE25-0014-01), the Shenzhen Science and Technology Foundation (Grant No.JCYJ20150529164656096, JCYJ20170302153955969), Guangdong Pre-national Project (2014GKXM054) and Shenzhen University Startup Grant (No.2018062). Bingsheng's work is partially funded by a collaborative grant from Microsoft Research Asia and a NUS start-up grant in Singapore.

REFERENCES

[1] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones. In *NSDI'13*. 185–198.
[2] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. 2014. GRASS: Trimming Stragglers in Approximation Analytics. In *NSDI'14*. 289–302.
[3] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the outliers in map-reduce clusters using Mantri. In *OSDI'10*. 1–16.
[4] Qi Chen, Cheng Liu, and Zhen Xiao. 2014. Improving MapReduce performance using smart speculative execution strategy. *TC* 63 (2014), 29–42.
[5] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *CACM* 51, 1 (2008), 107–113.
[6] Marc Gamell, Ivan Rodero, Manish Parashar, Janine C. Bennett, Hemanth Kolla, Jacqueline Chen, Peer-Timo Bremer, Aaditya G. Landge, Attila Gyulassy, Patrick McCormick, Scott Pakin, Valerio Pascucci, and Scott Klasky. 2013. Exploring Power Behaviors and Trade-offs of In-situ Data Analytics. In *SC'13*. 77:1–77:12.
[7] Peter Garraghan, Xue Ouyang, Paul Townend, and Jie Xu. 2015. Timely Long Tail Identification through Agent Based Monitoring and Analytics. In *ISORC'15*. 19–26.
[8] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *OSDI'16*. 99–115.
[9] Shadi Ibrahim, Bingsheng He, and Hai Jin. 2011. Towards Pay-As-You-Consume Cloud Computing. In *SCC'11*. 370–377.
[10] Shadi Ibrahim, Hai Jin, Lu Lu, Bingsheng He, Gabirel Antoniu, and Song Wu. 2012. Maestro: Replica-Aware Map Scheduling for MapReduce. In *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 *CloudCom'10*. 17–24.
[12] Dean Jeffrey. 2009. Large-Scale Distributed Systems at Google: Current Systems and Future Directions. In *LADIS'09*.
[13] Ripal Nathuji, Canturk Isci, and Eugene Gorbatov. 2007. Exploiting Platform Heterogeneity for Power Efficient Data Centers. In *ICAC'07*. 5–5.
[14] Tien-Dat Phan, Shadi Ibrahim, Gabriel Antoniu, and Luc Bougé. 2015. On Understanding the Energy Impact of Speculative Execution in Hadoop. In *DSDIS'15*. 396–403.
[15] Tien-Dat Phan, Shadi Ibrahim, Amelie Chi Zhou, Guillaume Aupy, and Gabriel Antoniu. 2017. Energy-Driven Straggler Mitigation in MapReduce. In *Euro-Par'17*. 385–398.
[16] Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. 2013. Hadoop's Adolescence: An Analysis of Hadoop Usage in Scientific Workloads. *VLDB* 6, 10 (2013), 853–864.
[17] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. 2015. Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale. In *SIGCOMM'15*. 379–392.
[18] Huicheng Wu, Kenli Li, Zhuo Tang, and Longxin Zhang. 2014. A Heuristic Speculative Execution Strategy in Heterogeneous Distributed Environments. In *PAAP'14*. 268–273.
[19] Huanle Xu and Wing Cheongu Lau. 2014. Speculative Execution for a Single Job in a MapReduce-Like System. In *CLOUD'14*. 586–593.
[20] Huanle Xu and Wing Cheongu Lau. 2015. Task-Cloning Algorithms in a MapReduce Cluster with Competitive Performance Bounds. In *ICDCS'15*. 339–348.
[21] Orcun Yildiz, Matthieu Dorier, Shadi Ibrahim, Robert Ross, and Gabriel Antoniu. 2016. On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems. In *IPDPS'16*. 750–759.
[22] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys '10*. 265–278.
[23] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments. In *OSDI'08*. 29–42.
[24] Amelie Chi Zhou, Bingsheng He, Xuntao Cheng, and Chiew Tong Lau. 2015. A Declarative Optimization Engine for Resource Provisioning of Scientific Workflows in IaaS Clouds. In *HPDC'15*. 223–234.