Energy-Driven Straggler Mitigation in MapReduce

Tien-Dat Phan, Shadi Ibrahim, Amelie Zhou, Guillaume Aupy, Gabriel Antoniu

Energy-Driven Straggler Mitigation in MapReduce

Tien-Dat Phan[1], Shadi Ibrahim[2](✉), Amelie Chi Zhou[2], Guillaume Aupy[3], and Gabriel Antoniu[2]

[1] ENS Rennes / IRISA Rennes, France
tien-dat.phan@irisa.fr
[2] Inria Rennes - Bretagne Atlantique Research Center, Rennes, France
{shadi.ibrahim,chi.zhou,gabriel.antoniu}@inria.fr
[3] Inria Bordeaux - Sud-Ouest Research Center, Bordeaux, France
guillaume.aupy@inria.fr

Abstract. Energy consumption is an important concern for large-scale data-centers, which results in huge monetary cost for data-center operators. Due to the hardware heterogeneity and contentions between concurrent workloads, straggler mitigation is important to many Big Data applications running in large-scale data-centers and the speculative execution technique is widely-used to handle stragglers. Although a large number of studies have been proposed to improve the performance of Big Data applications using speculative execution, few of them have studied the energy efficiency of their solutions. In this paper, we propose two techniques to improve the energy efficiency of speculative executions while ensuring comparable performance. Specifically, we propose a hierarchical straggler detection mechanism which can greatly reduce the number of killed speculative copies and hence save the energy consumption. We also propose an energy-aware speculative copy allocation method which considers the trade-off between performance and energy when allocating speculative copies. We implement both techniques into Hadoop and evaluate them using representative MapReduce benchmarks. Results show that our solution can reduce the energy waste on killed speculative copies by up to 100% and improve the energy efficiency by 20% compared to state-of-the-art mechanisms.

Keywords: MapReduce; Energy efficiency; Straggler mitigation, Detection; Copy allocation

1 Introduction

Energy consumption has started to severely constrain the design and the way that data-centers are operated. Energy bill has become a substantial part of the monetary cost for data-center operators (e.g., the annual electricity usage and bill are over 1,120 GWh and $67 M for Google, and over 600 GWh and $36 M for Microsoft [13]). Moreover, as a result of the explosion of Big Data and applications becoming more data-intensive, it is natural for data-center operators

to extend their infrastructure with more machines, which are energy-hungry. This makes energy consumption a major concern for Big Data systems [7, 11].

In parallel, the increasing scale of data-centers results in a noticeable performance variation in operations [21, 22]. This is due to: i) the hardware heterogeneity caused by the gradual scaling out of data-centers [10], and ii) the dynamic resource allocation when adopting the virtualization technique to collocate different users [20]. The performance variation results in a large number of stragglers, i.e., tasks that take significantly longer time to finish than the normal execution time (e.g., 700-800% slower [1]). Since the job execution time is determined by the latest task, stragglers can severely prolong the job execution time. Speculative execution is a widely-used straggler mitigation technique to improve the performance of jobs. It launches a speculative copy for each straggler upon its detection. As soon as the straggler or the copy finishes, the other one is killed and the task is considered finished. Nonetheless, using speculative execution is not always beneficial. For example, Ren et al. [14] have shown that speculative execution can reduce the task execution time in *21%* of the time while the unsuccessful speculative copies consume more than *40%* extra resources. Thus, there exists a trade-off between performance gain and extra energy/resource consumption when using speculative execution [12].

Existing speculative execution mechanisms cannot achieve good trade-off between performance and energy efficiency. *First*, existing speculative execution mechanisms detect as many stragglers as possible in order to cut the jobs' heavy-tails. This policy is good for improving the performance, but can cause much extra energy consumption. *Second*, different speculative copy allocation decisions can result in different performance and energy consumption results [12]. For example, launching speculative copies on nodes with a small number of running tasks can result in short task execution time but leads to a high power consumption (refer to Section 3). Unfortunately, existing copy allocation methods do not consider this aspect. In this paper we make the following contributions.

- We introduce a novel straggler detection mechanism to improve the energy efficiency of speculative execution. The goal of this detection mechanism is to identify critical stragglers which strongly affect the job execution times and reduce the number of killed speculative copies which lead to energy waste. This hierarchical straggler detection mechanism can work as a secondary layer on top of any existing straggler detection mechanisms (Section 5).
- We propose an energy-aware copy allocation method to reduce the energy consumption of speculative execution. The core of this allocation method is a performance model and an energy model which expose the trade-off between performance and energy consumption when scheduling a copy (Section 6).
- We evaluate our hierarchical detection mechanism and energy-aware copy allocation method on the Grid'5000 [8] testbed using three representative MapReduce applications. Experimental results show a good reduction in the resource wasted on killed speculative copies and an improvement in the energy efficiency compared to state-of-the-art mechanisms (Section 7).

2 Related Work

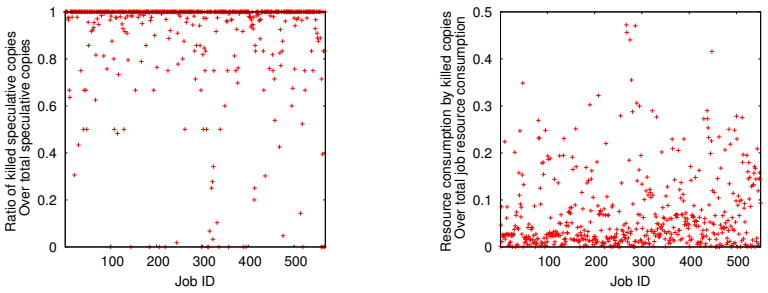
There is a rich body of research on straggler mitigation in MapReduce [4, 9].
Straggler Detection in MapReduce. Dean *et al.* [4] presented a straggler detection mechanism based on progress score, a 0-to-1 number represents the ratio of processed data over the total input data. A task, which has a progress score less than the average progress score minus 20%, is marked as a straggler. This mechanism has shown a reduction to the job execution times by 44%. Zaharia *et al.* [20] noticed that the progress score alone does not accurately reflect how fast a task runs as different tasks start at different times. Therefore, they present a new detection mechanism (i.e., *LATE*) which takes into consideration both the progress score and the elapsed time (i.e., the time each task takes from the moment it starts). These two parameters are used to calculate the progress rate of each task. In practice, this straggler detection mechanism can reduce the job execution times by a factor of 2. Recent studies [3, 5, 2, 6, 17, 18] have shown that there still exist several reasons that lead to incorrect straggler detections, including data locality and task execution skew. Ananthanarayanan *et al.* [2] proposed a *cause-aware* straggler detection mechanism. It keeps monitoring the performance and resource consumption of tasks and uses this information to infer the causes of slow task executions (e.g., non-local task and data skew). *Our hierarchical straggler detection mechanism complements these mechanisms to enforce identifying the most critical stragglers and hence reduce the extra energy consumption imposed by speculating non-critical ones.*
Straggler Handling in MapReduce. Ren *et al.* [15] proposed a speculation-aware scheduler, named Hopper. Hopper reserves spare resources to run speculative copies whenever needed. Ananthanarayanan *et al.* [1] presented Dolly, a straggler handling approach which launches multiple copies (i.e., clones) for each task when starting MapReduce applications. *While previous studies mainly answer the question of **when** to allocate the resources to speculative copies, this paper tackles the problem of **where** to allocate speculative copies. In particular, it leverages the heterogeneity of resources (in terms of performance and energy) to reduce the energy consumption of MapReduce applications.*

3 On The Energy Inefficiency of Speculative Execution

In this section, we discuss the energy inefficiency of the default speculative execution mechanism in Hadoop.

3.1 Huge Energy Waste Due to Unsuccessful Speculative Copies

Speculative execution is initially designed to handle stragglers and improve job performance. The common wisdom applied in existing straggler detection mechanisms is to detect as many stragglers as possible in order to cut the heavy-tails in job execution. For example, *Default* [4] decides a task with progress less than 80% of the average progress as straggler. *LATE* [20] marks the tasks with speed



(a) Ratio of killed copies (b) Resource on killed copies

Fig. 1. Production Hadoop cluster trace analysis: (a) More than 65% of the jobs have zero successful speculative copies; (b) The resource consumption caused by the unsuccessful copies can be substantial. In some cases, it can reach 40% of the total resource consumption.

less than the mean speed minus the standard deviation as stragglers. *Mantri* [2] considers tasks with 1.5x times longer execution time than average execution time as stragglers. To understand the energy efficiency of these straggler detection mechanisms, we have analyzed one month traces (October 2012) collected from a Hadoop production cluster in CMU [14]. Figure 1 shows the ratio of killed speculative copies, i.e., unsuccessful copies, over all copies for each Hadoop job, as well as the ratio of resources consumed by the killed copies over the total resource consumption of a job. We observe that many speculative copies are unsuccessful and are wasting a lot of resources. For example, among the total 568 jobs, there are 370 jobs which have speculative execution with no successful copy. For some jobs, the killed copies consumed more than 40% of the job's total resource consumption. The large number of unsuccessful speculative copies is mainly due to the late detection and the wrongly detected stragglers. To conclude, the philosophy of detecting as many stragglers as possible in speculative execution is no longer optimal from the energy perspective.

3.2 Speculative Copy Allocation Matters

We have observed that there is a trade-off between the performance and energy consumption for tasks executing on different nodes, according to the current status of the nodes. Figure 2 shows the average task execution time and the energy consumption of a node when executing different numbers of tasks concurrently. The application used is WordCount and the number of cores in the node is four. For example, we find that when the number of concurrent tasks is three, we can obtain the lowest energy consumption, without sacrificing too much the performance. Thus, allocating speculative copies to different locations, which may have different numbers of running tasks, can result in different performance and energy consumption results. Unfortunately, existing copy allocation meth-

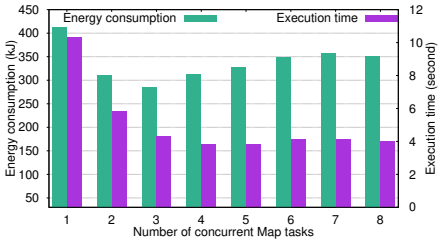


Fig. 2. Variability in execution times and energy consumptions with different numbers of concurrent Map tasks for WordCount application.

ods have ignored such a trade-off. For example, *Default* [4] follows the simple FCFS policy to allocate copies to the first freed slot, without considering any of the performance and energy objectives. In *Mantri*, the task placement is mainly based on the performance objective which ensures that a copy is more likely to finish earlier than the original task. In order to improve the energy efficiency of speculative execution, it is important to take into consideration the impact of different copy allocation decisions to the overall energy consumption.

Based on the two observations, in the following sections, we present a novel straggler detection mechanism and a smart speculative copy allocation method, in order to improve the energy efficiency of speculative executions.

4 Architectural Model

Considering the straggler mitigation problem in a cluster, we provide the following models to describe the energy and performance behaviors of tasks running in the cluster.

Power and energy model. For any node in the cluster, we assume there are c cores which support t threads each. The power consumption of a running node is composed of two parts, namely the fixed static power consumption $\mathcal{P}_{\text{static}}$ and the dynamic power consumption proportionally related to the number of active cores. We use $\mathcal{P}_{\text{dyn}}$ to denote the power consumption resulted by activating one core and n to denote the number of tasks running on the node. Then the total power consumption $\mathcal{P}$ of a running node can be modeled as in Equation 1.

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

The energy consumption E of a node is its power integrated over time and thus can be modeled as $E = \int_0^T \mathcal{P}(t)dt$. 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 $EE = 1/E$.

Average slowdown factor and interference model. We model the slowdown to a task caused by interference between concurrent tasks running on the same

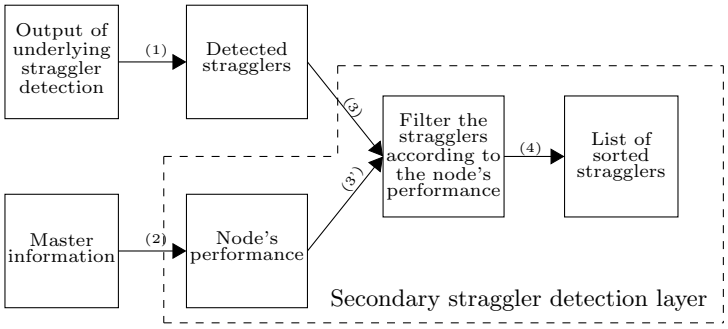


Fig. 3. Hierarchical straggler detection architecture.

node using the average slowdown factor α defined for a node. We observe that α equals to one when the number of concurrent tasks is less than the number of cores c. This is because each task can be executed on a dedicated core and there is hardly any interference between the tasks. When the number of tasks increases beyond c, the interference also increases. Denote the number of running tasks as n, then the average slowdown factor α can be calculated as below.

$$\alpha = \begin{cases} 1 & \text{for } 1 \leq n \leq c \\ \frac{n}{c} & \text{for } c < n \leq ct \end{cases} \tag{2}$$

5 Hierarchical straggler detection mechanism

In this section, we present the architecture of our hierarchical straggler detection mechanism. Our hierarchical mechanism works as a secondary layer on top of an existing straggler detection mechanism. The goal of this detection layer is to select the *critical stragglers*, i.e., the long-running stragglers which strongly affect the job execution time, from the list of stragglers detected by an existing detection mechanism. The secondary detection layer considers the stragglers at the node-level. That means, it detects only the stragglers on very slow nodes. The reason for this strategy is that most stragglers are caused by node-level problems, such as a node with worn-out hardware and node-level resource contentions which lead to slow tasks [2]. We identify all the nodes with performance less than β of the average node performance as slow nodes. In the evaluation, we discuss the impact of this parameter on the speculative execution results.

Figure 3 shows the design of the secondary straggler detection layer. Specifically, it takes the stragglers detected by the underlying straggler detection layer as input. Then, it calculates the performance of each node and filters out the stragglers that are not hosted on slow nodes. We calculate the performance of a node using the following equation.

$$Perf_{host} = \frac{1}{n} * \alpha * \sum_{i=1}^{n} Perf_{task}^{i} \tag{3}$$

where α is the slowdown factor and

$$Perf^{i}_{task} = \frac{progress * input}{duration} \tag{4}$$

Equation 4 evaluates the performance for a specific task, where *progress* represents the ratio of finished work over the task's total work, *input* is the size of the task's input data in bytes and *duration* is the time from the starting moment of the task. This information of each task is extracted from the Master node's database. Equation 3 means that the performance of a host is defined as the sum of the performances of all tasks running on the host. After the filtering, the rest stragglers are sorted according to their own performance and the most critical straggler (with the worst performance) is placed in the beginning of the list. We filter and sort the stragglers according to Equations 3 and 4 with the consideration of optimizing the energy efficiency of speculative execution. Apparently, a slow straggler running on a poor performance node is expected to be more critical straggler. Such stragglers are the main reason of causing heavy-tails in job executions and as a result wasting a lot of energy. Thus, handling those critical stragglers first can potentially lead to better energy efficiency. It is important to note that our secondary straggler detection layer is independent from the underlying detection layer, and therefore it can be easily integrated with any existing straggler detection mechanisms.

6 Energy-Aware Speculative Copy Allocation

After having the list of stragglers detected by the hierarchical straggler detection mechanism, we propose an energy-aware speculative copy allocation method to further optimize the energy efficiency of speculative execution.

6.1 Problem Definition

Given a list of suspected stragglers, the copy allocation method maps each straggler to a node with idle slots (denoted as an idle node) and start a copy of the straggler on that node, in order to optimize the overall energy efficiency of speculative execution. Assume there are S copies (s_i, $i \in [1, S]$) to be launched and N idle nodes (n_j, $j \in [1, N]$) to host the copies. We can easily formulate the copy allocation problem as a variant of the classic bin packing problem, where the size of each bin (i.e., a node) equals to the number of idle slots in the bin. Thus, the copy allocation problem is a NP-hard problem. In the next subsection, we propose a heuristic to obtain a good solution to this problem.

When the value of N is small, there are not many choices to make copy allocation decisions and the optimized energy efficiency results may not be good. Thus, we adopt the same methodology as Delay scheduling [19]. That is, we first check the value of N when making the copy allocation decision. If N is small, we wait a few seconds to have more idle nodes for potentially better results. In our experiments, we wait three seconds when N equals to one.

6.2 Copy Allocation Heuristic

There are many existing heuristics such as first-fit and best-fit algorithms to solve bin-packing problems. In this paper, we propose a heuristic similar to best-fit for our copy allocation problem. Following the order of stragglers sorted by the hierarchical straggler detection, we search for the node that can best fit each copy sequentially. We define the fitness of mapping a copy to a node according to the energy efficiency of the map. As the energy efficiency is affected by both the energy consumption and the performance of job, given any map from copy of straggler i to node j, we first provide two models to estimate the job execution time change and the energy consumption change caused by launching a copy of straggler i on node j.

Execution time change estimation. As the list of stragglers returned by hierarchical straggler detection mechanism are sorted according to their performances, the head of the list is always the most critical straggler. Handling the critical straggler can directly contribute to the reduction of job execution time. Thus, we can estimate the job execution time change ΔT_{ij} caused by launching a copy of straggler i on node j using the difference between the task execution time of straggler i before and after launching the copy. Assume that straggler i is running on node k.

$$\Delta T_{ij} = \alpha_k * \frac{(1 - progress_i) * input_i}{Perf^k_{host}} - \alpha_j * \frac{input_i}{Perf^j_{host}} \tag{5}$$

where the first term stands for the left over time for the straggler to finish if no copy is launched and the second term stands for the execution time of the copy on node j.

Energy consumption change estimation. 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 energy consumption change caused by launching a copy of straggler i on node j as follows.

$$\begin{aligned} \Delta E_{ij} &= (\mathcal{P}_k + \mathcal{P}_j) \cdot T_s - (\mathcal{P}_k + \mathcal{P}'_j) \cdot T_c \\ &= \mathcal{P}_k \cdot \Delta T_{ij} + \mathcal{P}_j \cdot T_s - \mathcal{P}'_j \cdot T_c \end{aligned} \tag{6}$$

where T_s equals to the first term of Equation 5 and T_c equals to the second term of Equation 5. $\mathcal{P}_j$ and $\mathcal{P}'_j$ are the power consumption of node j before and after adding a copy of straggler i, which can be calculated using Equation 1.

Given the above two models and the definition of energy efficiency, we can choose the map which gives the best ΔE_{ij} result as the best fit solution (i.e., the highest improvement to energy efficiency). **Algorithm 1** presents the general flow of our copy allocation heuristic, where *stragglers_list* contains the list of ordered stragglers and *idle_nodes* contains the list of nodes with idle slots.

7 Evaluation

In this section, we evaluate our hierarchical straggler detection mechanism and copy allocation method in real Hadoop cluster and compare them with existing

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

Algorithm 1: Speculative copy allocation heuristic.

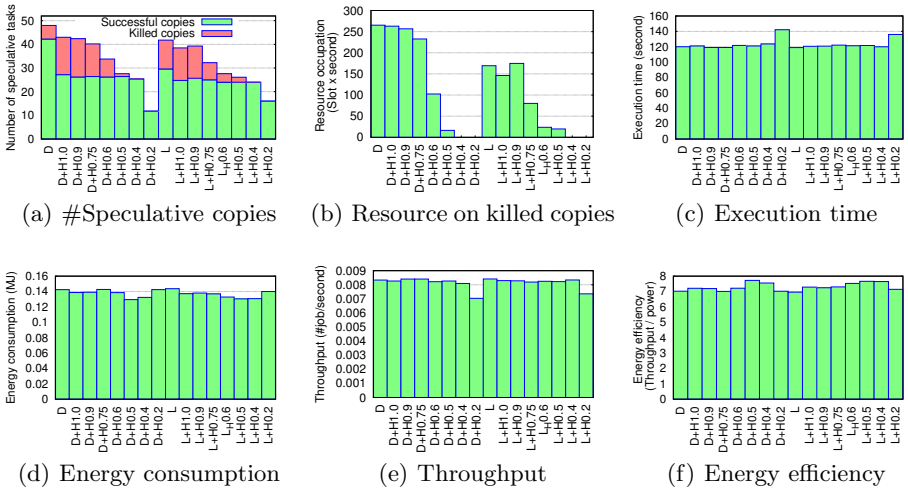
straggler detection mechanisms and copy allocation methods. We implemented our techniques in the Hadoop 1.2.1 stable version, with roughly 1500 lines of JAVA code. Both mechanisms are implemented as extra modules to the core of Hadoop to allow users to easily adopt our techniques using the Hadoop general configuration file.

7.1 Experimental Setup

Testbed. All of our experiments were conducted on a cluster of 21 nodes from the Nancy site of Grid'5000 testbed [8]. We configured the cluster with one master and 20 workers. Each node in the cluster is equipped with 4-core Intel 2.53 GHz CPU, 16 GB of RAM and 1 Gbps Ethernet network. The power consumption of the nodes are monitored by Power Distribution Units. Thus, we can acquire fine-grained and accurate power consumption values during the experiments. All experiments are run for 10 times and the average values are reported.

Applications. We adopt three widely-used MapReduce applications chosen from the well-known Puma MapReduce benchmark suite [16]. The three applications have different characteristics, where *Kmeans* is a compute-intensive application, *Sort* is an I/O-intensive application and *WordCount* has similar requirements on the computation and I/O resources. The input data size of the applications are all set to 10 GB. The number of Map and Reduce tasks are both set to 160 tasks.

Straggler injection. In order to inject stragglers, we use the Dynamic Voltage-Frequency Scaling technique (DVFS) to tune the CPU frequencies (hence the capabilities) of nodes. According to the CMU Hadoop production cluster traces [14], the ratio of stragglers varies from 0 to 40% of the total number of tasks. We choose the straggler ratio of 20% in our experiments. Thus, we set four nodes out of the 20 workers in our cluster to lower CPU frequencies, which are 1.20 Ghz, 1.33 Ghz, 1.46 Ghz and 1.60 Ghz.



(a) #Speculative copies (b) Resource on killed copies (c) Execution time

(d) Energy consumption (e) Throughput (f) Energy efficiency

Fig. 4. WordCount application with different straggler detection mechanisms.

Comparisons. We conduct two sets of comparisons. In the first set, we compare the hierarchical straggler detection mechanism with the Default detection mechanism [4] and LATE [20] detection mechanism. Second, we compare our proposed copy allocation heuristic (denoted as Smart) with the following two methods:

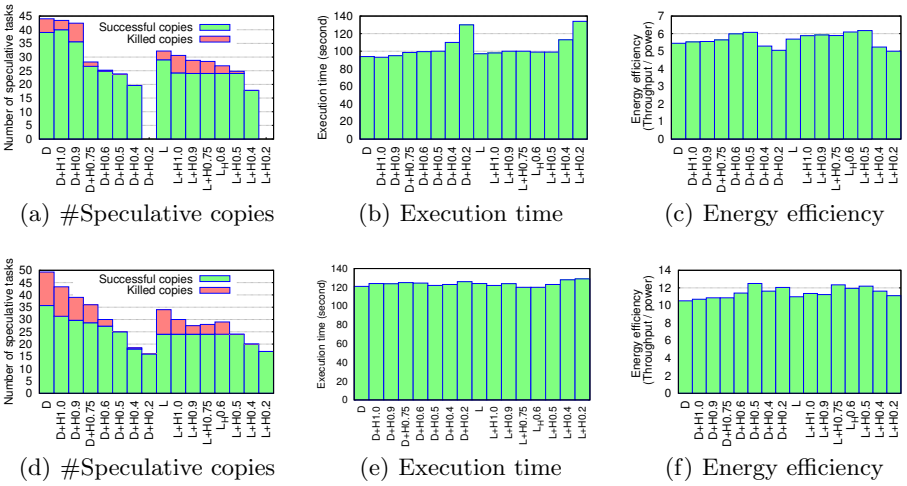
Performance-driven allocation: This method differs from Smart in that it launches speculative copies on nodes which give the best execution time reduction as calculated by Equation 5.

Power-driven allocation: This method differs from Smart in that it launches speculative copies on nodes which cause the lowest additional power consumption. The additional power consumption for a node j equals to $\mathcal{P}'_j - \mathcal{P}_j$ as in Equation 6.

7.2 Evaluation

Comparison results on straggler detection mechanisms. Figure 4 shows the performance and energy results of a single WordCount job running with different straggler detection mechanisms. We use the default copy allocation method in this experiment. In the x-axis, "D" stands for the Default straggler detection mechanism, "L" stands for the LATE detection mechanism, "D+Hx" and "L+Hx"stand for using the hierarchical straggler detection mechanism on top of Default and LATE, respectively, where "x" stands for the value of the β parameter used for node filtering in the hierarchical layer.

We have the following observations. First, from Figure 4(a), we find that the hierarchical straggler detection layer can greatly reduce the number of un-



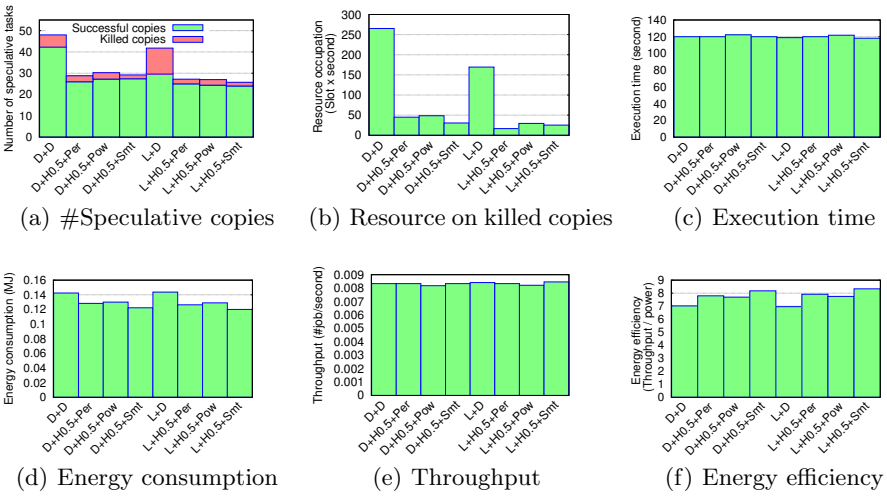
(a) #Speculative copies (b) Execution time (c) Energy efficiency

(d) #Speculative copies (e) Execution time (f) Energy efficiency

Fig. 5. Kmeans (a - c) and Sort (d - f) applications with different straggler detection mechanisms.

successful speculative copies, and the reduction increases with the increase of β. As a result, the amount of resources wasted on the killed copies is reduced (see Figure 4(b)) by up to 94% compared to Default and 88% compared to LATE. The total energy consumption is also reduced (see Figure 4(d)) by up to 9% compared to both Default and LATE. Second, adding the hierarchical layer does not sacrifice the performance too much (except when $\beta = 0.2$) as shown in Figure 4(c) and 4(e). When $\beta = 0.2$, there is an obvious degradation in the performance. This is mainly because that when β is too small, some of the real stragglers are missed and can still cause a heavy-tail to the job. Specifically, we can see that when $\beta = 0.4$, almost all the stragglers filtered by hierarchical are successful stragglers. Thus, when we reduce β to be smaller than 0.4, some of the real stragglers will be filtered out. Third, the hierarchical straggler detection mechanism can obtain better energy efficiency compared to Default and LATE (except when $\beta = 0.2$), as shown in Figure 4(f). When $\beta = 0.5$, we obtain the best energy efficiency, which is 10% higher than both Default and LATE. Thus, we set β to 0.5 by default to have the best energy efficiency result while maintaining similar performance compared to existing mechanisms

Similar observations have also been found with the other two applications. Figure 5 shows the results obtained for the Kmeans and Sort applications. We can observe that, for the compute-intensive Kmeans application, we can obtain even higher reduction in the energy consumption while maintaining similar performance. When $\beta = 0.5$, we improve the energy efficiency by 13% and 10% compared to Default and LATE, respectively. Thus, we can conclude that, the hierarchical straggler detection mechanism can greatly improve the energy efficiency of speculative executions with a comparable performance. In the following



(a) #Speculative copies (b) Resource on killed copies (c) Execution time

(d) Energy consumption (e) Throughput (f) Energy efficiency

Fig. 6. WordCount application with different copy allocation methods.

experiments, we focus on the WordCount application which shows the average improvement and use the default value of 0.5 for β.

Comparison results on copy allocation methods. Figure 6 shows the energy and performance results of running a single WordCount job with different speculative copy allocation methods. We evaluated in total eight combinations of the straggler detection mechanisms and copy allocation methods. Specifically, "D+D" and "L+D" are chosen as the baseline, which stand for using the Default detection mechanism with Hadoop's default copy allocation method and using the LATE straggler detection mechanism with default allocation, respectively. "D+H0.5+y" and "L+H0.5+y" refer to using hierarchical straggler detection mechanism on top of Default and LATE, respectively, where "y" stands for the copy allocation method used for allocating the copies.

We have the following observations. First, with our Smart copy allocation method, we can further reduce the energy consumption of speculative executions compared to existing mechanisms. For example, with the combination of Hierarchical and Smart, we can achieve 17% and 20% higher energy efficiency compared to Default and LATE using the default copy allocation method (see Figure 6(f)). Second, considering only the performance or power during the copy allocation is not good enough. For example, from Figure 6(a), we observe that Power-driven has the highest number of killed copies among the three compared allocation methods. This is because Power-driven tends to launch copies on nodes with low additional power consumption (i.e., highly utilized nodes) and thus can cause long execution time for the copies. As a result, some of the long running copies are killed and causing resource waste (see Figure 6(b)) and thus extra energy consumption (see Figure 6(d)). This suspicion can be verified with Figure 6(c) and 6(e), which shows that Power-driven has the longest execution time (and the

lowest throughput) compared to other allocation methods. Overall, Smart can improve the energy efficiency by 7% and 8% compared to Power-driven with the Default and LATE detection mechanisms, respectively. The improvement over Performance-driven are 5% and 6% using Default and LATE, respectively. The observations show that our Smart copy allocation method can further improve the energy efficiency of speculative execution.

8 Conclusion

Speculative execution is an important technique used for mitigating stragglers and improving performance of MapReduce jobs. However, few studies have looked at the energy efficiency of speculative executions. In this paper, we propose two techniques to trade-off the performance and energy efficiency for speculative executions. First, we propose a hierarchical straggler detection mechanism, which eliminates non-critical stragglers to reduce the energy waste on killed speculative copies. Second, we propose an energy-aware speculative copy allocation method which consults the performance and energy models to allocate speculative copies to the most energy efficient locations. Experimental results using real implementation demonstrate that our solution can reduce the energy waste on killed speculative copies by up to 100% and improve the energy efficiency by up to 20% compared to state-of-the-art methods. For future work, we plan to study the impact of using reservation-based scheduling on the energy efficiency of speculative executions.

Acknowledgment. This work is supported in part by the ANR KerStream project (ANR-16-CE25-0014-01). 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).

References

1. Ananthanarayanan, G., Ghodsi, A., Shenker, S., Stoica, I.: Effective Straggler Mitigation: Attack of the Clones. In: USENIX NSDI 2013. pp. 185–198 (2013)
2. Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., Harris, E.: Reining in the Outliers in MapReduce Clusters Using Mantri. In: USENIX OSDI 2010. pp. 1–16 (2010)
3. Chen, Q., Liu, C., Xiao, Z.: Improving MapReduce Performance Using Smart Speculative Execution Strategy. IEEE Transactions on Computers 63(4), 29–42 (2014)
4. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM 51(1), 107–113 (2008)
5. Ibrahim, S., Jin, H., Lu, L., He, B., Antoniu, G., Wu, S.: Maestro: Replica-Aware Map Scheduling for MapReduce. In: IEEE/ACM CCGrid 2012. pp. 435–442 (2012)
6. Ibrahim, S., Jin, H., Lu, L., Wu, S., He, B., Qi, L.: LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud. In: IEEE CloudCom 2010. pp. 17–24 (2010)

7. Ibrahim, S., Phan, T.D., Carpen-Amarie, A., Chihoub, H.E., Moise, D., Antoniu, G.: Governing Energy Consumption in Hadoop Through CPU Frequency Scaling. Future Generation Computer Systems 54(C), 219–232 (2016)
8. Jégou, Y., Lantéri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.G., Iréa, T.: Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. The International Journal of High Performance Computing Applications 20(4), 481–494 (2006)
9. Jin, H., Ibrahim, S., Qi, L., Cao, H., Wu, S., Shi, X.: The MapReduce Programming Model and Implementations. Cloud Computing: Principles and Paradigms. pp. 373–390 (2011)
10. Lee, G., Chun, B.G., Katz, H.: Heterogeneity-aware Resource Allocation and Scheduling in the Cloud. In: USENIX HotCloud 2011. pp. 4–4 (2011)
11. Leverich, J., Kozyrakis, C.: On the Energy (in)Efficiency of Hadoop Clusters. SIGOPS Operating Systems Review 44(1), 61–65 (2010)
12. Phan, T.D., Ibrahim, S., Antoniu, G., Bouge, L.: On Understanding the Energy Impact of Speculative Execution in Hadoop. In: IEEE DSDIS 2015. pp. 396–403 (2015)
13. Qureshi, A.: Power-Demand Routing in Massive Geo-Distributed Systems. In: Ph.D.dissertation, MIT (2010)
14. Ren, K., Kwon, Y., Balazinska, M., Howe, B.: Hadoop's Adolescence: An Analysis of Hadoop Usage in Scientific Workloads. The VLDB Endowment 6(10), 853–864 (2013)
15. Ren, X., Ananthanarayanan, G., Wierman, A., Yu, M.: Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale. In: ACM SIGCOMM 2015. pp. 379–392 (2015)
16. Thottethodi, M., Ahmad, F., Lee, S., Vijaykumar, T.: Puma: Purdue MapReduce Benchmarks Suite. Technical Report, Purdue University (2012)
17. Xu, H., Lau, W.C.: Resource Optimization for Speculative Execution in a MapReduce Cluster. In: IEEE ICNP 2013. pp. 1–3 (2013)
18. Xu, H., Lau, W.C.: Task-Cloning Algorithms in a MapReduce Cluster with Competitive Performance Bounds. In: IEEE ICDCS 2015. pp. 339–348 (2015)
19. Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., Stoica, I.: Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In: ACM EuroSys 2010. pp. 265–278 (2010)
20. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R., Stoica, I.: Improving MapReduce Performance in Heterogeneous Environments. In: USENIX OSDI 2008. pp. 29–42 (2008)
21. Zhou, A.C., He, B., Cheng, X., Lau, C.T.: A Declarative Optimization Engine for Resource Provisioning of Scientific Workflows in Geo-Distributed Clouds. In: ACM HPDC 2015. pp. 223–234 (2015)
22. Zhou, A.C., He, B., Liu, C.: Monetary Cost Optimizations for Hosting Workflow-as-a-Service in IaaS Clouds. IEEE Transactions on Cloud Computing 4(1), 34–48 (2016)