eSplash: Efficient Speculation in Large Scale Heterogeneous Computing Systems

Jiayin Wang* jane@cs.umb.edu
Teng Wang* Teng.Wang002@umb.edu
Zhengyu Yang† yangzy1988@coe.neu.edu
Ningfang Mi† ningfang@ece.neu.edu
Bo Sheng* shengbo@cs.umb.edu

*Department of Computer Science, University of Massachusetts Boston, 100 Morrissey Boulevard, Boston, MA 02125
†Department of Electrical and Computer Engineering, Northeastern University, 360 Huntington Ave., Boston, MA 02115

***Abstract*—In this paper, we aim to develop an efficient speculation framework for a heterogeneous cluster. Speculation is a common mechanism that identifies 'slow' node in a cluster and starts redundant tasks on other nodes to guarantee the reliability. We consider MapReduce/Hadoop as a representative computing platform, and our general goal is to accurately and quickly identify the straggler nodes during the job execution. On the one hand, our approach significantly reduces unnecessary speculative executions that occupy system resources, but do not get finished. On the other hand, when a node is prone to failure, our solution is able to detect it at an early stage and effectively launch a speculative task to avoid the delay in the job execution. We implement our solution in Hadoop platform and evaluate it with extensive experiments. The results show that our solution is efficient and effective when handling the speculative execution. The job execution time in our system is superior to that in the current Hadoop distribution.**

I. Introduction

In the past few years, we have all witnessed the rise of Big Data. Processing platforms such as Hadoop [1] and Spark [2] have been widely adopted for different applications. When users deploy the platform in a large scale cluster, the data processing performance is always crucial to the applications. This paper aims to improve the data processing performance by developing an efficient speculation scheme in a heterogeneous system.

In any large-scale computing cluster, node failures are normality in practice. A usual omen is the straggling computing performance on the node. Speculative execution is a common and effective solution for mitigating the impact of node failures, e.g., speculation is a built-in component in Hadoop. Basically, once detecting a straggler node, the cluster will launch a redundant copy of the task running on the problematic node. Once either of them is finished, the other one will be killed. The intuition is to trade the resource efficiency with the reliability, especially if the delay of one task may further postpone the whole data processing. However, the traditional speculation does not work well in a heterogeneous cluster, which consists of nodes with different hardware profiles. The heterogeneous setting has become a common environment in practice due to various reasons such as incremental hardware upgrade and diverse demands from different applications. Designing a speculation scheme in such a heterogeneous cluster, however, is challenging because it is very difficult to distinguish straggling nodes from naturally slow nodes.

978-1-5090-5252-3/16/$31.00 ©2016 IEEE

In this paper, we present eSplash, a Hadoop system with an efficient speculation scheme specifically designed for heterogeneous clusters. We identify the problems in the existing Hadoop system and develop the following major components: (1) Cluster all the nodes into different levels according to their computing performance; (2) Identify straggler nodes by monitoring the task's estimated finish time and progress rate; (3) Submit speculative request with parameters that guide the future execution. All the techniques presented in this paper are implemented in Hadoop YARN system. We conduct extensive experiments for evaluation, and the results show that eSplash significantly improve the system performance.

II. Related Work

MapReduce [3] is a programming model and an associated implementation for processing and generating large data sets [4] [5]. Speculative Execution [6] predicts the 'slow' tasks and launch redundant copies to re-execute.

However, due to system perturbations and equipment partial upgrade, most clusters in industry are not homogeneous anymore. Regarding to this truth, a large volume of work aiming at improving performance of Hadoop in a heterogeneous cluster has been done recently. B.T Rao, etc [7] provide guidelines on how to overcome bottlenecks of heterogeneous clusters. Based on previous suggestions, a hybrid solution [8] of FIFO, FairSharing [9] and COSHH [10] is introduced based on job classification. What's more, J.X etc. [11] place data on different nodes to assure a balanced load aiming at improving performance by optimizing data locality. S.G etc. [12] propose a ThroughputScheduler which dynamically selects nodes by optimally matching job requirements to node capabilities. Targeting on the problem that speculation mechanism degrades predictability of a cluster, Hopper [13] finds a balance between scheduling decision and speculation, also retrieves outstanding result after testing on Hadoop, Spark [2] and Sparrow [14], both centralized and decentralized schedulers.

Not only about scheduler, some other work also seeks the opportunity to increase the usage of storages in a heterogeneous cluster. For instance, N.S.I and X.L [15] propose new hybrid design and data placement policies to accelerate HDFS. Similarly, Cura [16] optimizes global resource utilization by configuring MapReduce jobs from the view of a service

provider. In addition, to improve the performance, recent work FRESH [17] and OMO [18] have developed dynamically resource allocation in Hadoop.

Nevertheless, previous work neglects that many unnecessary speculative tasks generated by slow nodes is one of the most important reasons for traditional Speculative Execution strategy incapable of adapting heterogeneous environment. LATE Scheduler [19] is created to solve the previous problem by only speculatively execute a copy of task that will finish farthest among all currently running tasks. Unfortunately the estimation of tasks' remaining execution time in LATE [19] is not accurate enough, especially it is unable to make self-adjustment to adapt the system. Inspired by preceding work, we create eSplash, which labels nodes into levels for system to accurately and quickly identify straggling nodes. Our eSplash not only shows high performance on YARN [20], Next Generation MapReduce of Hadoop, but also can be integrated into other cloud computing systems.

III. Background and Motivation

A. Speculative Execution in Hadoop

Speculative execution is an important feature in a Hadoop system. It aims to identify the unstable slave nodes in the cluster and avoid the delay of the job execution caused by these nodes. In a large scale Hadoop system, each node's status and performance may not be consistent for a long-term process depending on a lot of hardware and software factors. It is possible that some nodes are prone to a failure, and their performance is degraded at the runtime. This laggard performance could be temporary, or eventually require restarting the Hadoop service or even a reboot. In the Hadoop system, the centralized ApplicationMaster monitors the execution of every task, and if it detects that a task is running slowly (more slowly than the other same type of tasks), it will start a redundant task, called speculative task, as an alternative. When one of them is completed (either the original task or speculative task), the other task will be killed.

Particularly, Hadoop runs a background speculator service that maintains a statistics table to record all the execution times of the identical tasks, i.e., all the map or reduce tasks in a job. In other words, each MapReduce job has two entries in this statistics table, one for its map tasks and the other for its reduce task. The data in this table is updated upon the completion of each task. The speculator service will periodically check this table and the running tasks to find the candidate tasks for speculative execution. Specifically, it enumerates all the running tasks and estimate the finish time of each task t_i based on the elapsed time and the current progress as shown in Eq(1), where T_{now} is the current timestamp, $T_{start}(i)$ is the starting time of task t_i, and $PG(i)$ indicates t_i's current progress. In addition, the speculator service estimates the finish time of the alternative speculative execution in Eq(2. The execution time of the speculative task is estimated as the mean value of the historic execution times of the same type of tasks maintained in the statistic table. Fig. 1 shows an example of estimating Eq(1) and Eq(2) in the current Hadoop system. Apparently, if $EstEnd$ is greater than $EstRepEnd$, the task is expected to benefit from a speculative execution. When there are multiple candidate tasks for speculative execution, the speculator service will pick the one with the maximum value of $EstEnd - EstRepEnd$. Finally, the speculator service will create a new task attempt of the selected running task, and submit it to the pending task queue as a regular task.

$$EstEnd = \frac{T_{now} - T_{start}(i)}{PG(i)} + T_{start}(i) \quad (1)$$

$$EstRepEnd = mean(\ getTaskType(t_i)\ ) + T_{now} \quad (2)$$

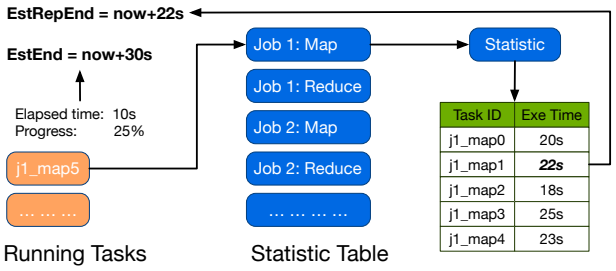


Fig. 1: Hadoop records the historic execution times of each type of tasks in each job for determining the candidate tasks for speculative execution

B. Problems in a Heterogeneous System

The current speculative execution, however, is not effective in a heterogeneous Hadoop system, and could even lead to severe performance degradations. The main issue is that a heterogeneous cluster consists of 'slow' nodes and 'fast' nodes. The mean value of the execution times is no longer a good guideline to judge if a node is abnormally slow. In addition, it is difficult to estimate the execution time of each speculative task as it depends on what type of nodes the task will be running on. Specifically, because of the diverse processing performance across the cluster, there are the following two major problems for the speculative execution.

First, the decision of starting or not starting a speculative task for each running task may be wrong. The intuition of the current design is to detect the running tasks that are far behind the expected progress compared to other finished tasks of the same type. This intuition, however, does not hold in a heterogeneous system as a 'slow' node does need more time to finish a task than a 'fast' node. If the current statistical data are mainly from the same type of tasks finished on 'fast' nodes, the speculator service may consider the task running on a 'slow' node behind the schedule and start a redundant speculative task for it. However, this 'slow' node is behaving normally, and the scheduled speculative execution is unnecessary. On the other hand, if a 'fast' node gets some problems and halts after executing a task for a while, the speculator service may consider its progress still in the normal range compared to the tasks finished on other nodes (especially on those 'slow' nodes). No speculative tasks will be created until this faulty 'fast' node has been hanging for a long time.

Second, the scheduled speculative tasks, when executed in the system, may not be as effective as we expect. The benefit

of speculative execution is to mitigate the negative effects of problematic nodes in the system and avoid the delay caused by them. In a heterogeneous system, however, the execution time of the speculative task depends on the node that hosts its execution. If the task is assigned to a slower node, the execution time would be longer than the original task. What is even worse is that the speculative task might be assigned to the same node that hosts the original task because of the diverse resource capacities.

We conduct an experiment on a cluster of 4 nodes with identical hardware settings. However, when configuring the Hadoop service, we set the each node's capacity of vcores with different values as follows, slave1(32 vcores), slave2(16 vcores), slave3(8 vcores), slave4(4 vcores). As the number of physical cores in each node is fixed, the node set more vcores has worse performance for each vcore. In another words, for the tasks in the same type and from the same job, the node with more vcores configured takes more time to execute a task. In this case, slave1 can be consider as the slowest node.

In the experiment, we execute 5 MapReduce jobs each consisting of 38 map tasks and 10 reduce tasks. The statistics of the speculative execution is listed in the following Table I. Out of the total of 240 original tasks, the speculator service has generated the redundant execution for 62 of them. All these 62 original tasks are initially assigned to the slowest node slave1. Among all the 62 speculative tasks, 25 of them are assigned back to slave1 and all of them get killed in the end.

	Task Type	slave1	slave2	slave3	slave4	Total
Finished	Map	0	19	5	5	28
	Reduce	0	0	1	0	1
Killed	Map	25	7	1	0	33
	Reduce	0	0	0	0	0

TABLE I: Speculative executions in an experiment

Above all, we aim to develop an efficient speculation scheme ESPLASH for a heterogeneous cluster which can accurately and quickly detect straggler nodes, effectively avoid unnecessary speculative execution, submit speculative tasks to the most appropriate nodes.

IV. DESIGN OF ESPLASH

In this section, we present the details of the design of ESPLASH that aims to efficiently manage the speculative execution in a large scale heterogeneous computing system. It mainly includes the three components:

- **Classify cluster nodes**: To accommodate the heterogeneous environment, our solution classifies the cluster nodes into different groups depending on their computing capabilities. A centralized manager maintains the run-time performance statistics for each individual group. These per-group data will serve other components such as detecting straggler nodes and submitting speculative tasks. The classification of the nodes can be pre-configured by the administrator, or dynamically determined based on run-time performance.
- **Detect straggler nodes**: The straggler nodes are the 'slow' nodes compared to other nodes in the same group, and could be prone to a failure. The tasks running on a straggler node are candidates for speculative execution. In ESPLASH, we develop a scheme that accurately and quickly detects straggler nodes in a large cluster.
- **Submit speculative tasks**: This is the most important component in ESPLASH. Basically, we need to determine whether a speculative task is worthwhile. The decision is based on the comparison of the estimated complete time of the current task (running on a straggler node), and the estimated execution time of a new speculative task. However, it is difficult to achieve an accurate estimation in practice, and it is more challenging in a heterogeneous cluster because the execution time of the speculative task depends on the computing capability of the hosting node. Our design in this component considers the practical factors, derives accurate estimation, and provides associated parameters for each speculative task for its future execution.

A. Classify cluster nodes

In order to effectively identify the straggler node and launch speculative tasks, the cluster manager has to compare a node's run-time performance to other nodes with similar hardware that are executing the same task.

In our design, each node in the cluster is associated with a *level* indicating its computation performance. Specifically, we classify all the nodes into multiple groups according to their performance, and the level value is the index number the group the node belongs to. We define that a higher level value represents a stronger computation ability. In other words, a level i node will finish a task faster than a level j node for $i > j$.

The node classification can be pre-configured by the cluster manager based on each node's hardware profile, or dynamically adjusted based the nodes' run-time performance. In this subsection, we focus on the dynamic classification algorithm at the run-time.

Performance vector: In our design, the cluster master maintains a *performance vector* PV_i for each node i,

$$PV_i = \{e_1, e_2, \ldots, e_D\},$$

where D is the number of distinct types of tasks node i has finished, and each value in the vector is the execution time of each type of tasks. For example, if there are five concurrent MapReduce jobs running in the cluster with the Fair scheduler, after finishing at least one map tasks from each job, every node will have a performance vector of five values. If more than one tasks have been finished for a particular type of tasks, then the average execution time will be filled in the PV. Then we will cluster all the nodes based on their PVs.

Clustering algorithm: We consider each node's performance represented by PV_i is a data point in a D-dimensional space, and our problem becomes similar to the traditional clustering problem such as k-mean. However, in our setting, the resulting

clusters(levels) indicate the performance and require a lexicographical order of the performance vectors. A node in a higher level is supposed to dominate any other nodes in lower levels for any type of tasks. Therefore, we present a new clustering algorithm based on the traditional k-mean algorithm.

Our solution consists of a grouping algorithm and a group-based k-mean clustering algorithm. The goal of the grouping algorithm is to merge individual PV data points into a set of groups that satisfy the lexicographic order. Then in our group-based k-mean algorithm, each group is the smallest unit to be assigned to a cluster, i.e., all the data points in a group will always stay in the same cluster.

The details are presented in Algorithm 1 and Algorithm 2. In Algorithm 1, we start with each data point as a group (line 1). Then the algorithm tries to merge the groups to enforce the lexicographic order. For each group g_i, we keep track of the minimum and maximum values of each dimension, recorded in min_i and max_i (lines 3–6). Then the algorithm compares every pair of groups, g_i and g_j, and merge them if they do not dominate each other. In line 8, we present the condition for the merging operation. If there exist two dimensions, where each of the two groups performs better in one of them, then we have to merge these two groups. After forming a new group, we need to merge other groups that overlap with the new group ($[min, max]$ overlapping in any dimension), and update the min_i and max_i (lines 10–11). The resulting groups are non-overlapping, and keep the lexicographic order between any two of them.

Algorithm 1 Grouping Algorithm

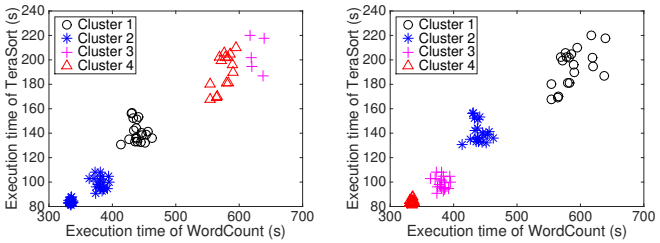
1: **Initial grouping:** $\forall i, g_i = \{PV_i\}$
2: **Merging groups:** form final groups based the lexicographical order
3: **for** g_i **do**
4: $\quad min_i(k) = min\{e_k \in PV_l | PV_l \in g_i\}, \forall k \in [1, D]$
5: $\quad max_i(k) = max\{e_k \in PV_l | PV_l \in g_i\}, \forall k \in [1, D]$
6: **end for**
7: **for** any g_i and g_j **do**
8: $\quad$ **if** $\exists a, b$, $min_i(a) > max_j(a)$ and $min_j(b) > max_i(b)$
9: $\quad$ **then** merge g_i and g_j, $g_i \rightarrow g_i \cup g_j$
10: $\quad$ Merge all other overlapping groups into g_i
11: $\quad$ Update min_i and max_i
12: **end for**

Based on the result of the grouping algorithm, we develop the following clustering algorithm. The basic steps are similar to the traditional k-mean algorithm. However, after each data point calculates the distance to each cluster center (line 2), it does not select the closest cluster center to join. In our algorithm, the decision has to be made by the whole group, not each individual data point. In particular, we adopt a voting scheme in line 3, by counting the preferred cluster center of every group member. The most popular cluster center will become the group preferred cluster center. Then all the data points in the group will be assigned to that cluster center. The algorithm repeats this iterative process until there is no reassignment.

Algorithm 2 Group-based k-mean Clustering Algorithm

1: Randomly select k cluster centers
2: For each PV_i, calculate the distance to each center, and pick the closest one as the preferred center
3: For each group g_i, check the preferred center selected by each member PV, and pick the center with the most votes as the group preferred center.
4: Assign all the PV points in a group to the group preferred center
5: Recalculate the cluster centers and repeat the process until no PV/group is reassigned.

The following Fig. 2 shows a comparison of traditional clustering algorithm and our group-based clustering algorithm. The ground truth is that we configured 4 types of nodes, each of 20 nodes. We conduct experiments with two jobs, WordCount and TeraSort, and measure the execution time of each job's map tasks on every node. Apparently, our algorithm accurately captures the pre-defined levels while the clustering result from the traditional k-mean is not feasible in our problem setting.



(a) Traditional k-mean (b) Group-based k-mean

Fig. 2: An example of clustering 80 nodes: we collect the execution time of two types of tasks (the map tasks in WordCount and TeraSort), thus each PV is a two dimensional data. The results of the clustering algorithms are plotted here.

B. Detect straggler nodes

The traditional speculation scheme makes the decision based on per-task performance and is not suitable for a heterogeneous system because of the naturally varying performance across the cluster. In our design, detecting straggler nodes is the first step for speculative execution. Only the tasks on a straggler node are candidates for speculation.

Performance Statistics Table: With every node associated with a level value, ESPLASH maintains a per-level performance statistic table (ST), and uses the data in this table to detect the straggler nodes in the system. This table consists of $L \times D$ cells, where L is the number of levels in the system and D is the number of active task types. Each cell $ST(i, j)$ represents the performance statistics of task type j at a level i node,

$$ST(i, j) =< \mu \text{ (mean)}, \delta \text{ (variance)}, PR \text{ (progress rate)} >,$$

where μ and δ are regular statistics for the task execution time, and PR records the average progress increase of this type of tasks in the past epoch. The table data is updated once the master node receives the heartbeat messages from the slave nodes.

Straggler Value: Based on the information in table ST, ESPLASH detects the abnormally slow node by comparing the task performance on the node with the statistic data for the level it belongs to. In particular, we assign each node a *straggler value* (SV) to indicate how likely the node is a straggler. Once the value exceeds a threshold τ, the node is marked as a straggler. The straggler value of a node is updated with the task performance on the node, and the following two aspects are included:

- **Estimated execution time:** For each running task on the node, we estimate its execution time (denoted bt $EstT$) by dividing the elapsed time by the task progress. Then, we compare it with the mean and variance values stored in table ST. Assume the node is in level i, the following formula is applied to update SV,

$$SV \leftarrow SV + \sum_{\text{running task } j} \frac{EstT - \mu(i, type(j))}{\delta(i, type(j))},$$

  where $type(j)$ returns the task type index of j and we exclude the running tasks whose $EstT$ values are smaller than the recorded mean values.
- **Progress rate:** While the estimated execution time is a good indicator of the performance, sometimes it takes a relatively long time for the system to detect a straggler node. For example, if a node normally executes a task and gets stuck when the task is almost finished, its estimated execution time will stay in the normal range for quite a long time. Therefore, we include the second metric, progress rate, to help quickly identify a straggler node. Let PR_j represents the progress rate of task j running on the node, we use the following formula to update SV,

$$SV \leftarrow SV + \sum_{\text{running task } j} \frac{PR(i, type(j))}{PR_j},$$

Note that there are other approaches to aggregate these two metrics to calculate SV, and their weights can be adjusted with coefficient parameters. The current design in ESPLASH is an empirical setting and out intuition is to pay more attention on the deficit of the progress rate.

C. Submit speculative tasks

Once straggler nodes are identified, all the active tasks running on those nodes are candidates for speculative execution. The goal of this module is to select one candidate task and submit a speculative task for it. In the traditional speculation scheme in Hadoop, we need to estimate the finish time of the speculative task and compare to the currently running task to determine if it is worthwhile. In a heterogeneous system, however, the finish time of the speculative task depends on which (level of) node will host the execution. For example, given a candidate task, it is possible that running a speculative task on a high level node will be faster, but running it on a low level node will be even slower than the current task. Therefore, when making the decision for speculative execution, we have to consider the level of the prospective hosting node of the speculative task. In addition, when submitting the speculation request, the level constraint should be specified.

In ESPLASH, we require every speculation request to be associated with a value of the minimum level ($minL$) to execute the speculative task. Only the nodes in the minimum level or higher level are eligible to host the speculative task. The following Algorithm 3 is developed in ESPLASH to determine the value of $minL$. When examining an active task running on a node at level i, the possible values for $minL$ range from i to HL which represents the highest level in the system. In the algorithm, we enumerate all the possible values, and then decide the best choice. Assume the $minL$ is set to be l, the algorithm calculates an expected execution time of the speculative task in line 3. The speculative task could be executed on a node at level l or above. We use $EstT(m)$ to indicate the execution time if the speculative task is hosted on a level m node, and $Pr(m)$ is the probability of this case. The value of $EstT(m)$ can be set as the mean value in the performance statistic table, and $Pr(m)$ is derived as follows:

Algorithm 3 Determine $minL$ for a speculation request

1: Given a task running on a straggler node at level i
2: **for** $l = i$ to HL **do**
3: $\quad ExpT_l \leftarrow \sum_{m \in [l, HL]} Pr(m) \cdot EstT(m)$
4: **end for**
5: $minL = min\{ExpT_l, \forall l \in [i, HL]\}$

$$Pr(m) = \frac{C(m)}{\sum_{j \in [l, HL]} C(j)},$$

where $C(j)$ is the number of containers on all the nodes at level j that can serve this type of task. The value of $C(j)$ can be pre-computed according to the resource capacity on all the level j nodes and the resource demands of the task. Eventually, in line 5, $minL$ is set to the value that yields the minimum expected execution time.

Finally, after every candidate task derives a $minL$ value with its speculative request, we need to pick one candidate and submit its request. Following the Hadoop workflow, this process will be repeated periodically, but every time only one speculative request can be submitted. We inherit the Hadoop's design, and use a *speculative value* to indicate the priority of each candidate task. However, this speculative value is re-defined as follows:

$$\frac{\sum_{m \in [minL, HL]} C(m) \cdot (EstEnd - T_{now} - EstT(m))}{\sum_{m \in [1, HL]} C(m)},$$

where $EstEnd$ is the estimated finish time of the original task, and T_{now} is the current timestamp. Essentially, this speculative value is the expected benefit (execution time reduction) the

task can obtain. Therefore, the task with the highest speculative value will be selected for speculative execution.

D. Other enhancements when executing speculative tasks

In ESPLASH, we developed a couple of other enhancements to improve the performance. First, the marked straggler nodes are excluded from hosting any speculative task. Second, if a speculative task waits for a certain amount of time in the queue of the pending task, we re-evaluate its estimated finish time and compare to the original task to see if it is still worth a speculative execution. Due to the page limit, the details are omitted in this paper, but these enhancements are also evaluated in our experiments.

V. System Implementation

To support our solution, we implemented our new scheduler ESPLASH on Hadoop YARN version 2.7.1 by creating a new **Speculator** component and modifying the **RMContainerAllocator** component (Container Allocator) in *MRAppMaster* (MapReduce Application Master). Fig. 3 shows the details of the system implementation.

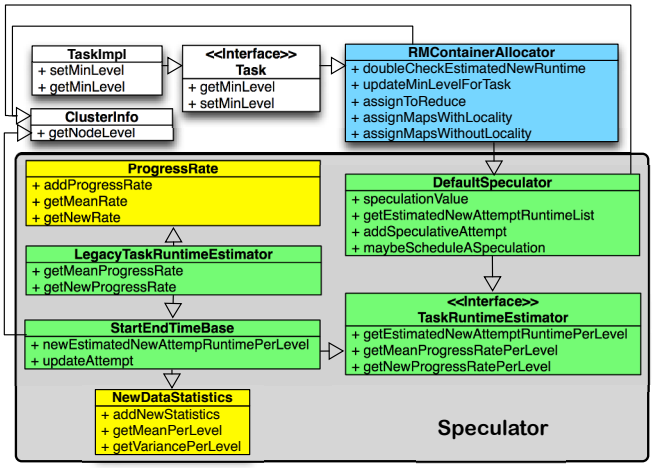


Fig. 3: System Implementation

First, to determine the speculative tasks, we create the new *Speculator* component. Its architecture follows *Speculate* component in native Hadoop. In *Speculator*, all modules in green exist in native Hadoop and we create new methods in them. And the modules in yellow are newly created by us. *NewDataStatistics* statistics the average execution time of each completed map/reduce task in each node level. According to such statistics, *StartEndTimeBase* estimates the execution time of every running task on the slave nodes in different levels. In addition, the new component *ProgressRate* monitors the progress variety of every running task in real time. *LegacyTaskRuntimeEstimator* is the subclass of *StartEndTimeBase* and it collects the information of the progress rate per running task from *ProgressRate*. *TaskRuntimeEstimator* is the interface for *DefaultSpeculator* to get all statistics above. Based on the estimated execution time and the progress rate of every running task, *DefaultSpeculator* firstly determines the candidate tasks for the speculative execution. Then it quantifies the speculative value of each candidate task and marks the minimum node level where the speculation can be executed. Finally, it selects the candidate task with the highest speculative value and creates a speculative task for it.

Second, to allocate the appropriate containers for the speculative tasks, we modify the *RMContainerAllocator* component. Firstly, it re-calculates the estimated execution time of the original tasks with pending speculative tasks to check whether it is still worthy to execute these speculative tasks. Secondly, it removes the unnecessary pending tasks and updates their minimum node levels. In the end, it checks the node levels that all available containers belong to and assigns the most appropriate container to each pending speculative task.

In addition, we modified *ClusterInfo* to set the node level for each slave node and the *TaskImpl*/*Task* in *job* to mark the minimum node level that every task can be executed in.

VI. Performance Evaluation

In this section, we evaluate the performance of ESPLASH and compare it with other alternative schemes.

A. Testbed Setup and Workloads

All the experiments are conducted on NSF CloudLab platform at the University of Utah [21]. In each server, there are 8 ARMv8 cores at 2.4GHz, 64 GB memory and 120 GB storage. We launched a cluster with 9 servers: 1 master node and 8 slave nodes. We create 4 node levels and assign two slave nodes in each level. To create the heterogeneous environment, we classify node levels by specifying different capacities of servers in different levels. Specifically, we configure 32 vcores in each slave node of level 1, 16 vcores of level 2, 8 vcores of level 3 and 4 vcores of level 4. As the number of physical cores is fixed in each server, the one configured by more vcores has worse performance for each vcore. So the slave nodes in level 4 achieve the best performance in executing a map/reduce task and the nodes in level 1 present the worst performance.

Our workloads for evaluation consider general Hadoop benchmarks with large datasets as the input. In particular, we use two datasets in our experiments including 20 GB wiki category links data and 20 GB synthetic data. The wiki data includes wiki page categories information, and the synthetic data is generated by the tool TeraGen in Hadoop. We choose the following four Hadoop benchmarks from Hadoop examples library to evaluate the performance: (1) *Terasort*: Sort (key,value) tuples on the key with the synthetic data as input. (2) *Word Count*: Count the occurrences of each word with a list of Wikipedia documents as input. (3) *Grep*: Take a list of Wikipedia documents as input and search for a pattern in the files. (4) *Wordmean*: Count the average length of the words with a list of Wikipedia documents as input.

B. Performance Evaluation

Given a batch of MapReduce jobs, our performance metrics are the increased makespan with stragglers and the accumulated wasted time of killed speculative tasks. We mainly

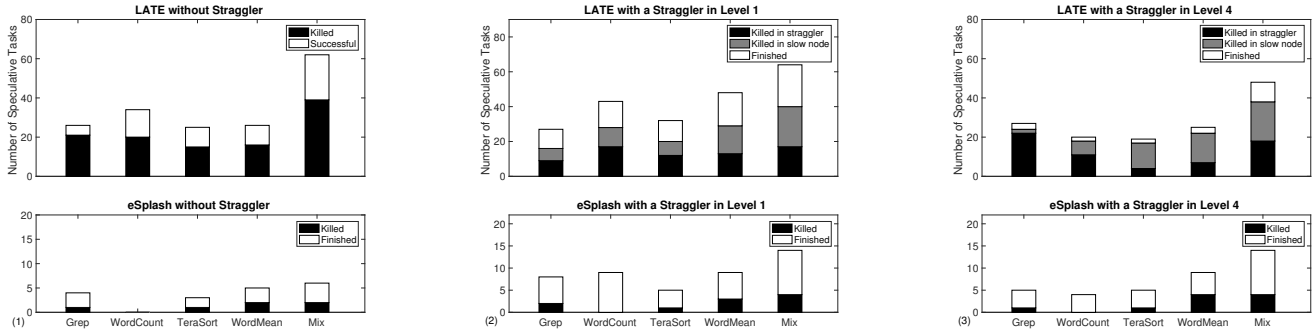


Fig. 4: The Speculative Tasks created under both *LATE* and *eSplash*: (1) without Straggler, (2) with one straggler on Node Level 1, (3) with one straggler on Node Level 4

compare ESPLASH to the native speculation scheduler in YARN (*LATE* [19]) and the one with speculation disabled (*Non-specu*). We have conducted two categories of tests with different workloads: *simple workloads* consist of the same type of jobs and *mixed workloads* represent a set of hybrid jobs. For each test of simple workloads, we generate 8 jobs of the same benchmarks. For testing mixed workloads, we mix all four benchmarks above and generate 2 jobs for each benchmark. For each job of both simple and mixed workloads, the input data is 20 GB. There are 80 map tasks and 10 reduce tasks created by each job and each task requires 1 vcore and 2 GB memory. In the rest of this subsection, we separately present the evaluation results in the heterogeneous environment: (1) without stragglers, (2) with stragglers which can be recovered, and (3) with stragglers which cannot be recovered.

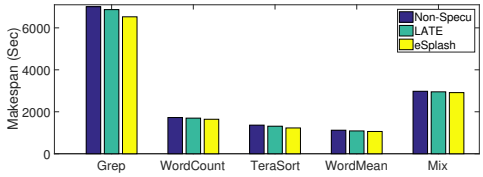


Fig. 5: Makespan without Stragglers

1) Performance without stragglers: For our first experiment, we test both single and mixed workloads in the heterogeneous cluster without any stragglers. The first graph of Fig. 4 shows the number of speculative tasks created during the experiments. The black parts represent all killed speculative tasks and the white parts show all successful ones. According to the principle of speculation, speculative tasks are killed because of their original tasks are finished earlier than the speculative ones. Ideally, there should be no speculative tasks created during the experiments. However, under *LATE*, there are 28 speculative tasks averagely created in the experiments of simple workloads and 62 ones in mixed workloads. The original tasks of such speculative ones are all from the slave nodes on node level 1. As there is no mechanism in assigning speculative tasks to appropriate slave nodes in *LATE*, on average, 64.8% of the speculative tasks are assigned back to the 'slow' nodes on node level 1 and killed when their original tasks are finished. We notice that there are still about 5 speculative tasks created in each experiment under ESPLASH. After tracing the logs of such tasks, we found the reason. In ESPLASH, we revoke the function in native YARN to report the estimated execution time of each running task. With a very low probability, it may report one extreme high value and ESPLASH mistakenly creates a speculative task based on such value. We will try to fix this issue in the future work. But still, ESPLASH reduces 80% - 91.9% unnecessary speculative tasks over *LATE*. In addition, the average accumulated execution time of killed speculative tasks in each experiment is 3536 seconds under *LATE* and 343 seconds under ESPLASH. And ESPLASH decreases 90.3% wasted time cost in killed speculative tasks over *LATE*.

Fig. 5 shows the makespan performance of ESPLASH, *LATE* and *Non-specu*. Although 64.8% speculative tasks are killed in *LATE*, the remaining successful speculative tasks help to speed up the finish of all jobs. There is no significant difference in makespan between *LATE* and *Non-specu*. On average, ESPLASH improves 5.89% and 4.58% of the performance on makespan compared to *Non-specu* and *LATE*.

2) Performance with stragglers which can be recovered: To evaluate the speculative execution with stragglers in the cluster, we manually slow down a slave node by running four CPU-intensive processes (the factorial of the integer 10,000) and four disk-intensive processes (*dd* tasks writing large files in a loop). Such processes last 1000 seconds during jobs execution and then the straggler will be recovered to normal performance. We run experiments separately with the straggler in 'slow' node (Level 1) and in 'fast' node (Level 4). The second and third graph of Fig. 4 shows the statistics of speculative tasks created during the experiments with a straggler in node level 1 and in node level 4. *Killed in Straggler* represents the killed speculative tasks which are assigned to the straggler node, *Killed in slow node* represents the killed speculative tasks which are assigned to the slave nodes in the same or lower node level compared to their original tasks. From the test results, all killed speculative tasks in *LATE* are either assigned to the straggler itself or a slower slave node. While triggering a straggler on the slow node (in node level 1), 61.9% averagely of speculative tasks are killed in *LATE*. Among all these killed speculative tasks, 50.4% of them are assigned back to the straggler. When the straggler is a fast node (in node level 4), on average 87.1% of speculative tasks are killed under *LATE* and 47.7% of the killed ones are assigned to

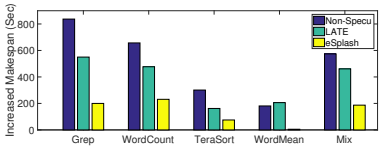


Fig. 6: Increased Makespan with a Straggler on Node Level 1

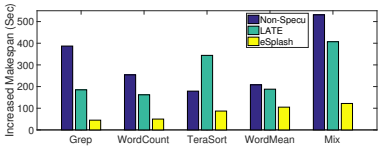


Fig. 7: Increased Makespan with a Straggler on Node Level 4

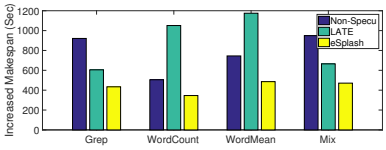


Fig. 8: Increased Makespan with a Straggler which will be Restarted

the straggler. Meantime, there is no speculative task assigned to the straggler or slower slave nodes under ESPLASH. Fig. 6 and Fig. 7 shows the increased makespan in both experiments. Generally, since *Non-specu* cannot address the situation with stragglers in the cluster, it represents the worst performance in the increased makespan. However, in the test with the straggler in a 'fast' node, as *LATE* assigns 13 out of 19 speculative tasks to the slave nodes in node level 1, the increased makespan of *LATE* is even 47.9% more than the one under *Non-specu*. With the straggler in a 'slow' node, averagely, ESPLASH decreases the increased makespan by 76.7% and 65.7% over *Non-specu* and *LATE*. With the straggler in a 'fast' node, on average, the increased makespan under ESPLASH is 69.4% and 66.7% less than the ones under *Non-specu* and *LATE*. From the test results, *LATE* cannot efficiently deal with the stragglers in a heterogeneous cluster.

3) Performance with stragglers which cannot be recovered: In practice, abnormally slow execution of a server is a sign of the system failure. Rebooting the whole system is a common operation to handle such failure. So we design an experiment to check whether speculative executions can deal with this issue. In the experiment, we set a slave node in the node level 4 as a straggler and slow down it by the same CPU-intensive and disk-intensive processes above. After running these processes for 1000 seconds, we manually shut down the processes of NodeManager and DataNode of the slave node and restart them after 600 seconds (to simulate the rebooting of the straggler node). We run the same experiment under *Non-specu*, *LATE* and ESPLASH. Fig. 8 illustrates the increased makespan under each mechanism. As the straggler is considered as a normal node and speculative tasks from other 'slow' nodes are assigned to the straggler, for the benchmarks Wordcount and Wordmean, the increased makespan under *LATE* is even more than the one under *Non-specu*. Under ESPLASH, as all the tasks running on the straggler have created speculative tasks on other 'faster' nodes, nearly no time is wasted to recover failed tasks on the straggler when it's shut down. ESPLASH shows the best performance and the increased makespan is averagely 42.4% less than *Non-specu* and 45.8% less than *LATE*.

VII. Conclusion

This paper studies the speculative execution in a large-scale heterogeneous computing cluster. Our goal is to mitigate the impact of node failures in the cluster. We develop a new speculation scheme ESPLASH which can efficiently and quickly identify the stragglers, submit the speculative tasks to the most appropriate nodes and avoid resource waste on the unnecessary speculative execution. We have implemented our solution on Hadoop YARN platform and conducted extensive experiments with various workloads. The results show a significant improvement on distinguishing the stragglers, assigning speculative tasks, and reducing the impact of stragglers on the makespan compared to a conventional YARN system.

Acknowledgement: This work was partially supported by UMB Healey Grant, NSF Grant CNS-1552525, NSF Career Award CNS-1452751, and AFOSR Grant FA9550-14-1-0160.

References

[1] Apache Hadoop. http://hadoop.apache.org.
[2] Spark. http://spark.apache.org.
[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
[4] Yucheng Low, Danny Bickson, Joseph Gonzalez, et al. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.
[5] Tong Wang, Vish Viswanath, and Ping Chen. Extended topic model for word dependency. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, July 2015.
[6] Tom White. Speculative execution. In *Hadoop: The Definitive Guide*, chapter 6. O'Reilly Media, Inc., 2012.
[7] B.Thirumala Rao, N.V.Sridevi, V.Krishna Reddy, and L.S.S.Reddy. Performance issues of heterogeneous hadoop clusters in cloud computing. 07 2012.
[8] Aysan Rasooli and Douglas G. Down. A hybrid scheduling approach for scalable heterogeneous hadoop systems. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 1284–1291, Washington, DC, USA, 2012. IEEE Computer Society.
[9] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, et al. Job scheduling for multi-user mapreduce clusters. Technical report, EECS Department, University of California, Berkeley, Apr 2009.
[10] Aysan Rasooli Oskooei and Douglas G. Down. Coshh: A classification and optimization based scheduler for heterogeneous hadoop systems. *Future Generation Comp. Syst.*, 36:1–15, 2014.
[11] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, J. Majors, A. Manzanares, and Xiao Qin. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010.
[12] Shekhar Gupta, Christian Fritz, Bob Price, Roger Hoover, Johan Dekleer, and Cees Witteveen. Throughputscheduler: Learning to schedule on heterogeneous hadoop clusters. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 159–165, San Jose, CA, 2013. USENIX.
[13] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 379–392, New York, NY, USA, 2015. ACM.
[14] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, pages 69–84, 2013.
[15] N. S. Islam, X. Lu, M. Wasi ur Rahman, D. Shankar, and D. K. Panda. Triple-h: A hybrid approach to accelerate hdfs on hpc clusters with heterogeneous storage architecture. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 101–110, May 2015.
[16] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bryan Langston. Cura: A cost-optimized model for mapreduce in a cloud. In *IPDPS*, 2013.
[17] Jiayin Wang, Yi Yao, Ying Mao, Bo Sheng, and Ningfang Mi. Fresh: Fair and efficient slot configuration and scheduling for hadoop clusters. In *CLOUD*, 2014.
[18] Jiayin Wang, Yao Yi, Ying Mao, Ningfang Mi, and Bo Sheng. Optimize mapreduce overlap with a good start(reduce) and a good finish(map). In *IPCCC*, Dec 2015.
[19] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, et al. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 29–42, 2008.
[20] Apache hadoop nextgen mapreduce (yarn). http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.
[21] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX*, 39(6), December 2014.