Performance Implications of Failures on MapReduce Applications

Mohammad Tanvir Rahman, Edgar Gabriel and Jaspal Subhlok

Department of Computer Science,
University of Houston, Houston, TX, USA,
email: {mtrahman3,egabriel, jaspal}@uh.edu

***Abstract*— Due to the growing size of compute clusters, large scale parallel applications increasingly have to deal with hardware malfunctions and other failure scenarios during execution. The overall goal of this research is to get good performance of MapReduce applications despite failures. The paper focuses on evaluation of the performance of two representative Hadoop MapReduce applications, 'WordCount' and 'Stack Exchange', with different execution parameters and under different failure scenarios. The paper also presents different options to inject failures into MapReduce applications to simulate real world failures. Some of the preliminary observations are that slowdown due to failure is higher with relatively larger input split sizes, slowdown peaks near the optimal split size, and that the performance and slowdown are sensitive to key MapReduce execution parameters.**

I. Introduction

As the size of clusters employed for parallel and distributed computing increases, hardware malfunctions and other failures are increasingly common during execution of long running applications. For certain classes of applications (e.g. applications with safety implications), it is crucial to deal dynamically with failures, since a down-time of the application is often not an option. But even for mainstream applications, the ability to deal elegantly with software and hardware failures can improve performance significantly, e.g., by having to re-execute only a small subset of the application instead of re-running the entire application, in case of a node failure.

Analyzing the behavior of applications in failure scenarios is however a challenging topic. In real life, failures are unpredictable, and in most instances, not frequent enough to allow for a systematic evaluation of application behavior under worst case conditions. Hence, for failure analysis, one needs the ability to introduce failures at much higher rates than their occurrence in real life, and in a reproducible manner. Failure injection is a technique to trigger failures in applications or systems, and has been long used in computer design to test and evaluate error correction and failure management schemes [5]. Literature distinguishes between different types of failures, such as soft vs. hard failures or temporary vs. permanent failures.

Failure injection in parallel applications was the topic of multiple research projects [4]. Recent work introduced software and methodology that allows injection of various types of failures at runtime at specific locations in the source code [3], and in communication operations [2].

Many researchers have studied failure statistics in High performance computing. Schroeder et al. [**?**] studied 9 years of data including 23000 failures records on more than 20 different systems. The data identified the root cause of failures, the mean time to failures, and the mean time to repair, in order to understand failure statistics. Yuan et al. [**?**] studied job failures on High performance computing systems and showed that a failed job often consumes more computational resources than a successful job.

Parallel programming models and execution environments offer various degrees of support for surviving failures. MPI [6], the de-facto standard programming framework for High performance computing, does not have the ability to deal with process failures: if one of the compute nodes used by an MPI application crashes, the entire MPI application is terminated. Although there are efforts by the MPI Forum to integrate failure management based on ULFM [1], users mostly rely on checkpoint-restart [8] for failure mitigation, i.e., the applications has to regularly write out the content of its (most important) variables. In case of a node failure, the application is restarted from the most recent checkpoint. Other HPC programing models such as CHARM++ offer better support for handling failures, the ultimate challenge being on how to balance support for handling failures and scalability of applications on high-end systems [7].

In Big data analytics, the Hadoop environment incorporates resilience to failures on multiple levels. The Hadoop File System (HDFS) replicates file system blocks in order to survive node and/or hard drive failures. The execution model of MapReduce applications allow Hadoop to keep track of work (or part of the work) that has already finished successfully, and re-execute missing work (splits) if a compute node is marked as lost. However, to the best of our knowledge, no research project has systematically evaluated the impact of various types of process and node failures on the performance of MapReduce applications.

The goal of this paper is to evaluate and quantify the performance of Hadoop MapReduce applications in failure scenarios. Within the scope of this work we are interested in failures that will trigger the error handling protocols in Hadoop to deal with lost work. We are ignoring problems such as bit-flips in the user data, which would lead to erroneous

application results, but would not necessarily be recognized by Hadoop. We discuss various approaches to inject failures into a MapReduce applications, and evaluate the performance penalty depending on the type of failure injection, the function being affected by the failure, and various system and application parameters.

The remainder of the paper is organized as follows: section II describes our methodology to inject process and node failures into Hadoop. Section III provides a description of the application scenarios and the actual test cases performed as part of this work. Finally, section IV summarizes our findings and presents the future work.

II. Background

This section provides a brief overview of the structure and execution of Hadoop MapReduce applications, and explains the techniques used in this project for estimating the number of failures and injecting failures.

A. Brief Anatomy of MapReduce Applications

The scope of this work is to analyze the performance and behavior of Hadoop MapReduce applications in failure scenarios. We describe the most important aspects of the execution of a MapReduce application; for a more detailed description see [10]. Starting from Hadoop version 2, a MapReduce application first launches an *Application Master*, which is responsible for managing the execution of the application. The Application Master requests resources from the Resource Manager (e.g. YARN) and launches and monitors the map and reduce tasks. For a given input problem size, the data is split into equal chunks of, the so-called, *split size*. Depending on the number of compute nodes available for the execution of the map and reduce tasks, the Application Master assigns a certain number of splits to each map instance for execution. If a map task fails, the work is re-assigned to another map task. The reduce tasks are launched after the completion of all map tasks, although there can be overlap between these two phases since the shuffling of intermediate key-value pairs between map tasks and reduce tasks is started as soon as a designated amount of data is available.

From the programing perspective, the user typically has to implement, at the minimum, a mapper and a reducer class. The mapper class consists of three methods: (i) the *setup* method of the mapper class is called to execute operations that only have to be executed once, e.g., reading a file from the distributed cache of Hadoop to access a set of global constants; (ii) the *map* method is called once for every split assigned to that map instance; (iii) the *cleanup* method, which is called upon completion of all splits assigned to a map instance. The reducer class contains similar three methods that are implemented by the application developer.

B. Logistics of failure execution

We discuss various options for injecting failures in a MapReduce application.

1) Task Failures injected at Source Code Level: The first approach to introduce failures is based on terminating a map or reduce instance by introducing a `system.exit()` statement in the software. For simplicity, we are calling these types of failure as 'Task failure'. Task failure is considered as nondeterministic software failure in real life. Examples for these type of failures are a non-recoverable ECC errors in a main memory address of the application; or external (and erroneous) interference of a device driver. Re-executing the same application with same data set will typically finish without any failures. We focus on mappers and reducers, but ignore for the sake of simplicity, other components of execution, such as combiners or the shuffle operation occurring between the map and reduce tasks.

The goal within the scope of this work is to terminate a subset of the map or reduce tasks, without making the entire application fail. To achieve this goal we introduce a control mechanism such that a randomly selected subset of tasks is terminated. A random number is generated in the range of $< 1, upper\_bound >$ for each task, and if the value is below a given *threshold*, the task is terminated. Thus, the combination of the upper bound and threshold value control the granularity and the percentage of tasks to be terminated.

One challenge when using this approach is how to derive the upper bound and the threshold value used to determine whether to terminate a task. Assuming that failures are not correlated and follow a Poisson distribution, the parameters needed for this estimate for the Map phase are the time required to process a single split ($T_{TimePerMap}$), the number of nodes used for the Map phase, and the assumed Mean-Time-To-Failure (MTTF) of a single node. Using these parameters, the probability of a node facing a failure while executing the splits assigned to it can be estimated, and the values for the random number generated adjusted accordingly.
Example: Assume a problem that leads to 1024 splits, processing a split takes 2 minutes, and the cluster has 32 nodes. Each map instance will thus have to process 32 splits, which takes 64 minutes total. Assuming an (unrealistically low) MTTF value of 2 hours (or 120 minutes), the success probability of a map instance is

$$\begin{aligned} p &= e^{-\lambda t} \\ &= e^{-\frac{1}{MTTF} \cdot T_{TimePerMap} \cdot N_{SplitsPerMapper}} \\ &= e^{-\frac{1}{64} \cdot 2 \cdot 32} \\ &= 0.5866 \end{aligned}$$

The failure probability is then 1-0.5866 = 0.4134. One approach to map this failure probability to a range of values used in combination with a random number generator is to define the range of random number to be $< 1, 100 >$ and the threshold value to 41: any random number less or equal to 41 would lead to the task being terminated.

From the implementation perspective, the code terminating a map or reduce instance could be inserted in the *setup* method, the *map* or *reduce* method, or the *cleanup* method. If the termination code is inserted in the *setup* method, the task

would terminate before any split could be processed, and thus no work would be lost by this approach. Any overhead observed would be purely from restarting the JVM that runs the map or reduce instance. Inserting the termination code into the *map* or *reduce* method allows for the most fine grained control on a per-split basis for mappers, or per-intermediate key basis for reducers. Depending on the data volumes (i.e. split size as well as some other parameters of Hadoop), all intermediate results up to the point when the termination is triggered could be lost. If a certain Hadoop internal threshold is reached, intermediate key-value pairs start to spill to disk, such that some work could be retained and might not have to be re-executed. Based on the Hadoop protocols however, it is not entirely clear to us whether that actually does happen, since the Hadoop daemons and Application Master would have to be sure that all intermediate key-value pairs generated by processing a particular split have actually been spilled to disk, or not.

The final option is to insert the termination code into the *cleanup* method. This option ensures, that all splits have been processed before terminating the map task. Similar to the previous discussion, some data might have already been saved to disk by Hadoop, and the shuffle operation might have been started already. In case the application uses a *Combiner*, we are however confident that the shuffle operation should not have started yet, and thus we will maximize the damage caused by terminating the mappers in the *cleanup* method. Similarly for the reduce class, terminating a reduce instance in the cleanup method ensures maximum damage in our opinion, with some results potentially already written to disk, but probably not everything yet. Thus, Hadoop might have to re-execute the entire reduce instance.

2) Node Failure injected with Administrative action: An alternative approach to terminate map and/or reduce tasks is by shutting down selectively Hadoop *NodeManagers* that are executing the job. For our discussion, we label this type of failures as 'Node failure', since it emulates a hardware level failure which leads to all operations running on the node to be terminated. This approach does not require modifications to the source code, but it requires administrative privileges. Shutting down a NodeManager could impact more than one map or reduce instance, since a node with multiple cores and sufficient memory might very well host multiple Hadoop JVMs. Furthermore, the impact of node failure might be very different whether that nodes executes the Application Master or 'just' regular map and reduce instances. Note that in our approach the Hadoop File System (HDFS) is not affected by the NodeManager shut down.

III. Evaluation

A. Application scenarios

We used two different applications in our experiments, the well known 'WordCount' and 'Stack Exchange' codes.

WordCount is a simple application that counts the number of occurrences of each word in a given input file. Although it is known as a benchmark, WordCount is at the core of many real applications in Big data analytics and other domains. The data used in this analysis is a system generated log file of size 36 GB. A combiner was used in our WordCount source code. We applied different split sizes ranging from 16 MB to 1 GB. Different number of reducers (1,4,8,16,32) were also used in our experiments. The split/node ratio for WordCount application was in the range $< 0.84, 53.19 >$.

The Stack Exchange application is based on the following scenario: given a data set containing many questions asked on www.stackexchange.com, the sets of corresponding answers with a label of accepted or not-accepted by the questioner, and a set of user rating scores for each answer, the goal is to compute and compare the average score of accepted and not-accepted answers. The input files are stripped down XML files where XML headers have been removed. The application parses files as regular string/text. The code consists of two chained MapReduce jobs. In the first MapReduce job (stage 1) , the input XML file is used to calculate the average score for each answer and store the result in an intermediate file in HDFS. In the second MapReduce job (stage 2), the intermediate file is used as input and processed to calculate the average score for accepted and not accepted answers. The stage 2 MapReduce job starts after finishing the stage 1 MapReduce job. This application was chosen to understand the impact of failure in chained MapReduce jobs. The input file size was 28.5 GB. The Stack Exchange source code had 10 reducers in stage 1 and two reducers in stage 2. The split/node ratio for Stack Exchange application was in the range $< 0.67, 42.47 >$.

For both applications, we injected different types of failures and checked the overall completion time of the application. Failures were injected during different phases (Map vs. Reduce) for the WordCount application, and on both stages of the Stack Exchange jobs. Each experiment was executed 3 times and the average value was used for the evaluation. The relative standard deviation observed across the various test cases was on most instances very low (between 1% and 10%) since we used a dedicated cluster with no interference from other users or jobs. Nevertheless, some experiments lead to higher standard deviations of around 30%. We will comment on those experiments and the conclusions drawn when discussing the individual tests. The failure rates used for the task failure experiments were 11% and 25%. The entire MTTF range for those experiments is $< 0.0001, 0.5107 >$. As discussed earlier, while these MTTF values are unrealistically low, it is often necessary to use such values to analyze the error protocols in a distribute software environments.

B. Cluster description

Tests were executed on the 'Whale' cluster located at University of Houston. The cluster consists of 42 compute nodes in total. Each node has two 2.2 GHz quad-core AMD Opteron processor (8 cores total) and 16 GB main memory. The nodes are interconnected by Gigabit Ethernet. The operating system is Linux using kernel 3.12.62-55. Hadoop version 2.7.2 is used on the cluster. The cluster has a 7 TB HDFS file system

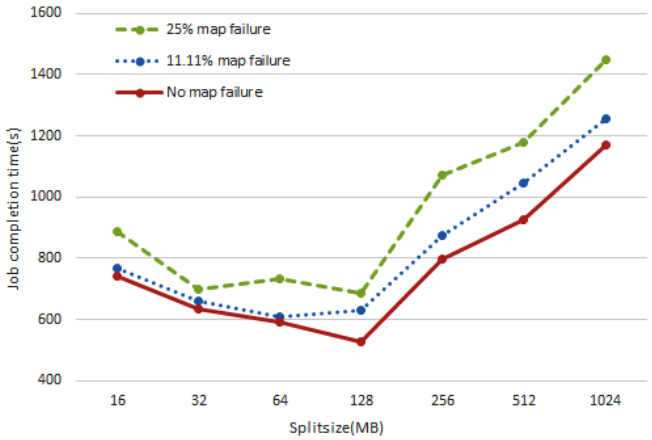

Fig. 1. Performance impact of map task failures on WordCount application with different split sizes and failure rates

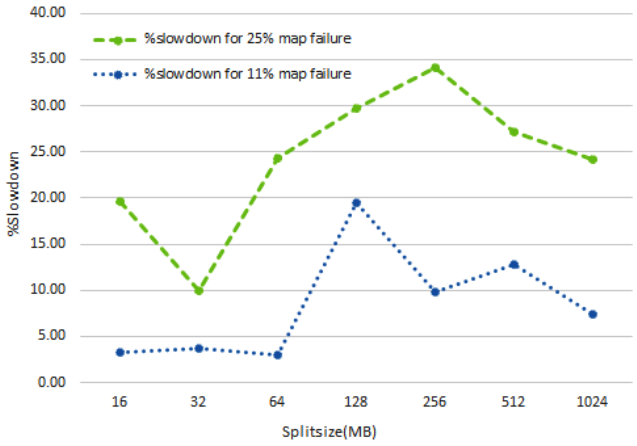

Fig. 2. Performance impact of map task failures on WordCount application as percentage slowdown

(using triple replication) with a default block size of 128 MB. Most YARN parameters used on the cluster were based on the default values used by the distribution.

C. Impact of Task Failures

As discussed in section II-B.1, task failures were injected by calling `system.exit()` in the `cleanup()` method of the application code. No hardware failure occurred during these failures. During a task failure, the corresponding node's JVM is shutdown; However the node is still available and can be assigned for performing other tasks.

1) Mapper failures: Fig 1 shows job completion time for different split sizes when injecting task failures in the mapper class of the WordCount application for the failure rates mentioned previously(11% and 25%). The X-axis shows different split sizes in MB and the Y-axis shows job completion time in seconds. The percentage increase in execution time with different failure rates and split sizes is plotted in Fig 2.

The results indicate that the job completion time is modestly higher with failures, with an expected higher increase for the higher failure rate. The lowest job completion time with and without failures is achieved for a split size of 128 MB, which is due to the fact that each split corresponds to exactly one HDFS block in this scenario. There is a sharp increase in completion time when going from 128 MB to 256 MB split size. This is due to the fact that split sizes larger than the HDFS block size necessitate a mapper to access data that is usually located on two or more physical nodes, thus reducing the locality of the operations and causing higher I/O cost and higher completion time.

Furthermore, we notice that the job completion time does not increase much for split sizes below 128 MB and completion time increases at a modestly higher rate for larger split sizes. This can be seen clearly from Fig 2. This can be explained by the fact that the same percentage of map failures causes a greater loss for larger split sizes, since more work has to be re-done in this scenario. In addition, there is a lower number of splits with larger split sizes and even fewer number of failed tasks to re-execute. As a result, a few nodes are re-executing failed tasks, while many nodes in the system remain idle.

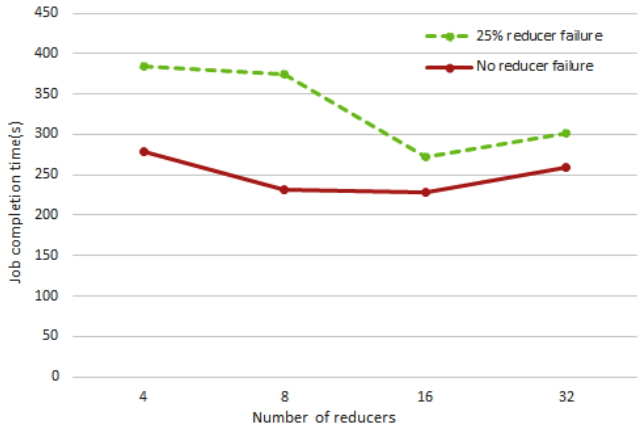

Fig. 3. Performance impact of reduce task failures on WordCount application with different failure rates and number of reducers

2) Reducer failures: Fig 3 shows job completion time for different number of reducers when injecting task failures in the reducer class of the WordCount application. The X axis shows the number of reducers and the Y axis shows the job completion time in seconds. The split size in these experiments is fixed at 128 MB, since the split size does not have an impact on reduce tasks. Tests have been executed with 4, 8, 16, and 32 reducers. Due to the relatively low number of reducers used, the application showed in many instances no failures at all for 11% failure rate. We focus therefore on a 25% failure rate in the subsequent analysis.

As observed from the graph, job completion time is lowest with 16 reducers with and without failures. We notice that with 4 and 8 reducers, job completion time is comparatively higher for 25% reducer failures. This can be verified from Fig

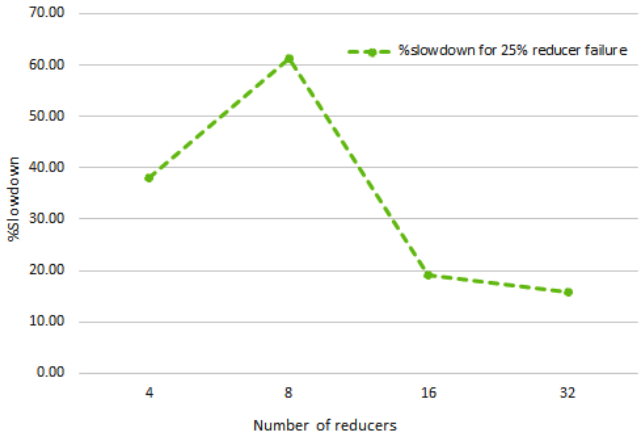

Fig. 4. Performance impact of reduce task failures on WordCount application as percentage slowdown

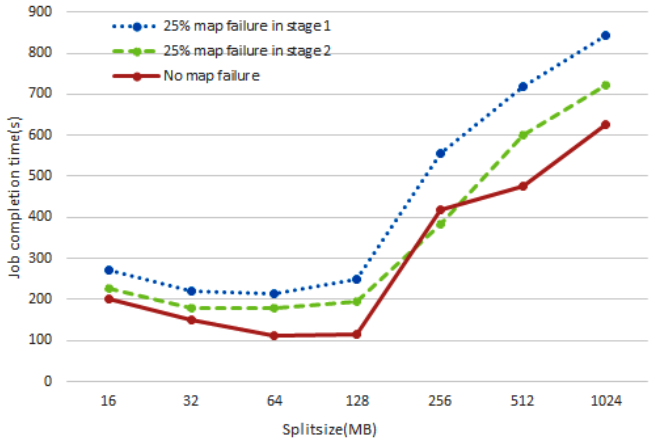

Fig. 5. Performance impact of map task failures on Stack Exchange application with different split sizes and for failures in mapper stage 1 and stage 2

4 which shows %slowdown for the same scenario as Fig 3. The graph clearly depicts that slowdown is larger for smaller number of reducers. The reason is that, with fewer reducers, the same percentage of reducer failure causes a greater loss and more work has to be re-done since the work per reducer is now larger. Please note that for this scenario, the work is not evenly distributed among all the nodes of the cluster. We had a maximum of 32 reduce tasks (no failure scenario) to finish with 42 nodes.

3) Chained MapReduce jobs: As explained previously, the Stack Exchange application consists of two separate MapReduce jobs executed in chained sequence. Fig 5 shows job completion time for different split sizes for task failures injected in the mapper class at both stages. Fig 6 shows the slowdown for the same scenario. The fundamental result of Fig 5 is very similar to the observations made in the previous subsection (see e.g. Fig 1), namely: i) the execution time is minimized for a split size equal to the HDFS block size, however, the job completion time does not increase significantly for split size below 128 MB; ii) completion time increases at a higher rate for larger split sizes; and iii) there is a sharp increase in completion time for split sizes from 128 MB to 256 MB.

Furthermore, we also note that a job with 25% map failure in stage 1 takes more time to finish than for a 25% map failure in stage 2. This is due to the fact that stage 2 of the application takes much less time to finish than stage 1. Furthermore, for larger split sizes, the map tasks operate on bigger splits and the re-execution takes more time. This is the reason for a relatively large gap for execution time between no failure, 25% stage 1 map failure and 25% stage 2 map failure lines when split size is bigger.

From Fig 6, we observe that slowdown is highest for optimal split size (128 MB). For the split size of 256 MB, the results seem to indicate that the failure-free execution is slower than injecting some failures. This data point is clearly obscured by the variability of the measurements across the three runs – also shown in the somewhat larger standard deviation of just

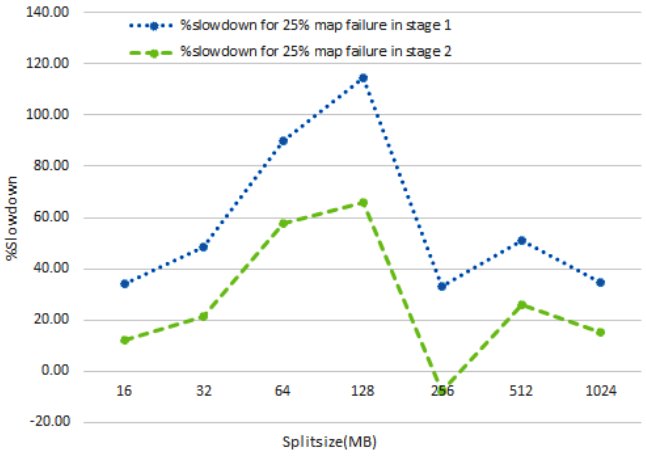

Fig. 6. Performance impact of map task failures on Stack Exchange application as percentage slowdown

over 10% – and might require further tests to achieve a lower standard deviation and higher reliability of the data points.

D. Impact of node failures

In the second set of experiments we injected node failures by shutting down the NodeManager service during our MapReduce job. As a result, the corresponding node becomes unavailable for computation and cannot be used for performing other tasks until the end of the current job. Note that the data node service in that node is not affected by this operation and hence data located on the node is still available. We used two types of node failures in our experiment: i) shutting down a non-application-master NodeManager aka *normal node failure* and ii) shutting down the NodeManager of an application master aka *application-master node failure*. After starting the application we checked which node is executing the *Application Master* in the cluster web interface and which other nodes are running non-application-master containers. Once

that information is available, we either kill another container running NodeManager (normal node failure) or shut down the application master NodeManager (application-master node failure). The terminated NodeManager is restarted again after the job has finished, ensuring that the same number of nodes is used through the entire analysis.

1) Normal Node failures: Fig 7 and 9 show job completion times for different split sizes (Fig 7) and number of reducers (Fig 9) when shutting down a non-master NodeManager during mapper and reducer phase of WordCount application respectively (*normal node failure*). For Fig 7, the number of reducers is 1 and for Fig 9, split size is set to 128 MB. Fig 8 and 10 shows %slowdown for the corresponding scenarios.

The main result of this analysis is that injecting a *normal node failure* leads to significant increases in the job completion times for the WordCount application. As shown in Fig. 7, the observed penalty exceeds 600 seconds, even though the re-execution of the failed task is relatively quick. The reason for this enormous penalty is the node failure detection timeout. Normally, the YARN Resource Manager monitors NodeManagers based on a heartbeat signal. If the heartbeat is missing for some time, the resource manager does not immediately declare the node to be 'dead'. Instead, it waits for a certain amount of time defined by *yarn.nm.liveness-monitor.expiry-interval-ms* (default value 600000ms) before removing the node from the pool of available resources.

In contrast to that, the Hadoop NodeManagers were not affected by the task failures in the experiments conducted in the previous subsection. Thus, they were able to immediately detect the exiting JVM, and report the work as 'lost'. On average, the observed slowdown increases by 80% for a normal node failure whereas using 25% task failure (in map phase) leads to a slowdown of 24% . For reducer phase, on average the slowdown increases by 150% for normal node failures whereas 25% task failures in reduce phase lead to a slowdown of 34%. The penalty is higher in reducer phase because of larger work loss.

We note using Fig 7 that the split size of 128 MB leads to the lowest execution time without failures. However, in case of node failure, the optimal split size moves reliably to 32 MB, both for the average execution time as well as all three individual runs. This result is especially relevant, since it shows that some of the parameters of Hadoop are in fact very sensitive to the type of failures experienced by an application, and very different values can lead to the optimal execution time under different failure conditions.

While a detailed explanation for the change in the optimal split size with and without failures is still work in progress, there is clearly a trade-off between having a more fine-grained execution using smaller splits (and therefore less work being lost in the case of a failure) and the overhead generated by having to manage more splits. The split size minimizing the application execution time will depend on the number of failures as well as the penalty incurred by a failure. One of our future goals is to derive analytic approximation of the execution time for MapReduce application that would allow us to determine the optimal split size for different failure scenarios.

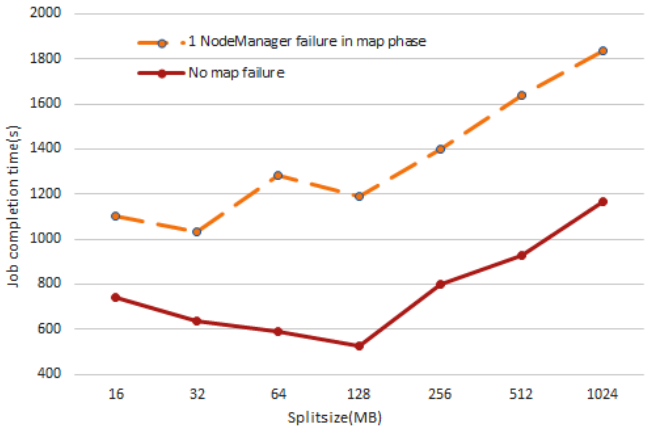

Fig. 7. Performance impact of node failure during mapper phase on WordCount application with different split sizes

In Fig 9 we notice that the lines are almost flat, which means the number of reducers has little impact on the completion time. Even with a low impact factor, Fig 10 tells us that slowdown is larger for optimal number of reducers (Reducer number = 8 and 16). In Fig 9, there is a significant delay between *no failure* and *node failure* plots. We believe, the delay is due to NodeManager failure detection and recovery of partial lost work. We observe that there is no 'actual' failed reducer task even after shutting down the NodeManager in reducer phase. The failure simulation procedure is explained as follows.

To simulate node failure in the reducer phase, a NodeManager is shutdown as soon as there are $r + 1$ nodes running the jobs, where r = number of reducers; 1 node is for the application master. According to our observations, each reducer is launched in a container in a separate node. We could not shutdown the NodeManager before this step to avoid shutting down a node with map container or a reducer container.

For example, consider a job running with 16 reducers ($r = 16$). Initially the job starts with just one node containing the application master. Soon it allocates more nodes and assigns map tasks in different containers on these nodes. As time passes and 100% map tasks are finished, all the nodes with mapper containers are released and we are left with 17 allocated nodes running 16 reducer containers and the application master. We observe the Hadoop web interface of the cluster and wait until we have 17 nodes left in that job. Then we kill one NodeManager to make sure we are killing a reducer container. Though shutting down the NodeManager at that stage does not cause any 'actual' reducer failure, it takes a significant amount of time to detect the node failure, overcome the loss and finish the job.

Fig 11 shows job completion time for different split sizes when we injected *normal node failures* by shutting down a

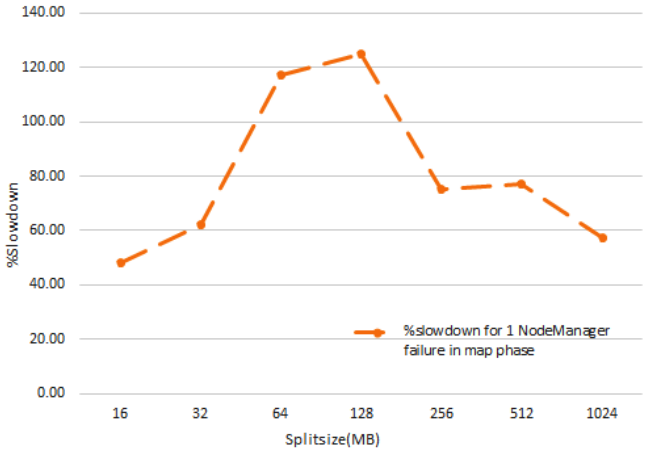

Fig. 8. Performance impact of node failure during mapper phase on WordCount application as percentage slowdown

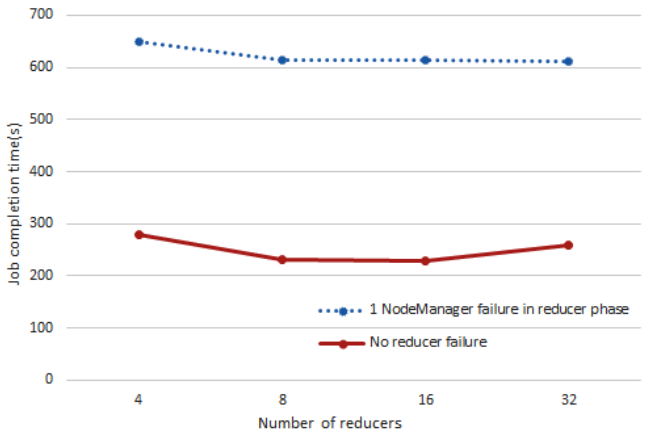

Fig. 9. Performance impact of node failure during reducer phase on WordCount application with different number of reducers

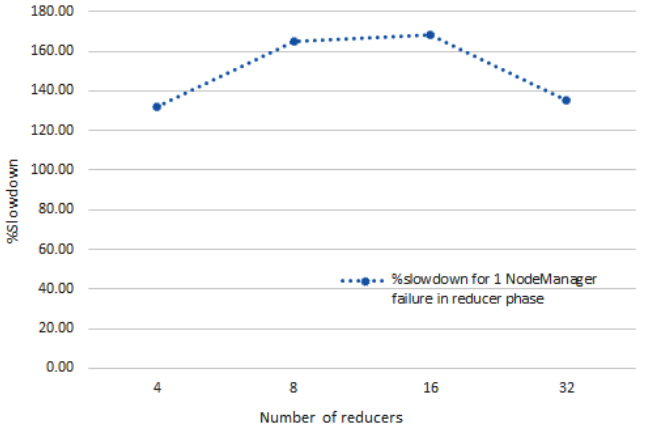

Fig. 10. Performance impact of node failure during reducer phase on WordCount application as percentage slowdown

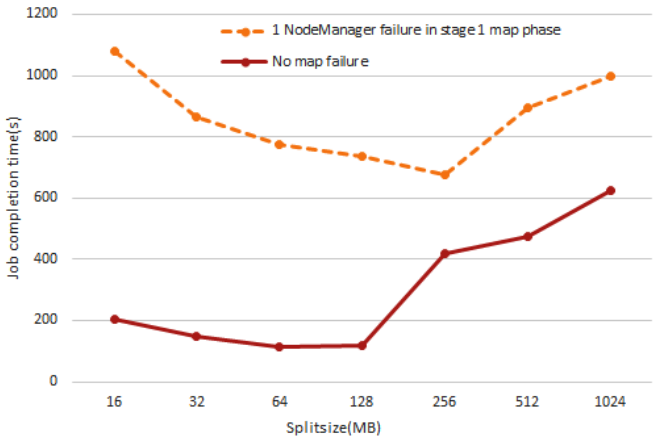

Fig. 11. Performance impact of node failure during stage 1 mapper phase on Stack Exchange application with different split sizes

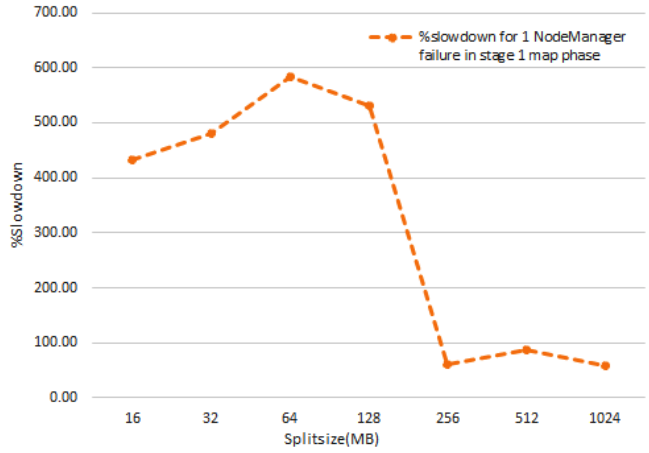

Fig. 12. Performance impact of node failure during stage 1 mapper phase on Stack Exchange application as percentage slowdown

non-master NodeManager during the mapper phase of Stack Exchange application. Here, we use 10 reducers in stage 1 and two reducers in stage 2. Similar to the WordCount application, injecting *normal node failures* increases job completion time significantly, and completion time increases at a higher rate for larger split sizes.

The optimal split size is 128 MB for the failure free execution, but 256 MB with a node failure. While this change in the optimal split size is observed for both, the average and minimum execution time, it would require some additional tests since the overall standard deviation for this test series is relatively high in the range of 30%.

Fig 12 shows the slowdown for the corresponding scenarios. Slowdown is much higher for split sizes less than 256 MB and slowdown is very high near optimal split size (with no failure).

2) Application-master node failures: Fig 13 shows job completion time while injecting *application-master node failure* by shutting down NodeManager of Application Master

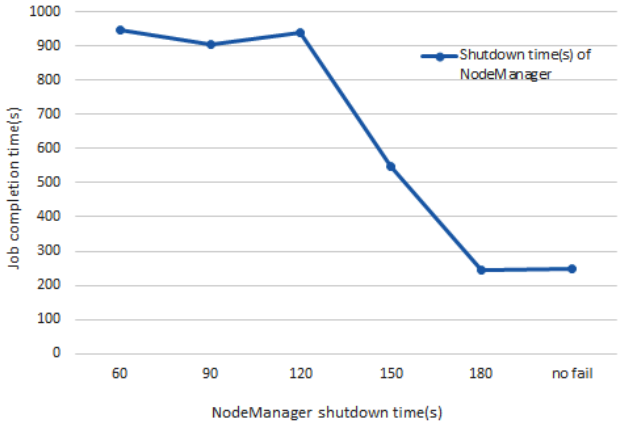

Fig. 13. Performance impact of Application-master node failure on WordCount application

in WordCount application. The 'Application Master' is terminated at different times and the time required to automatically restart the Application Master and finish the job is measured. Here, the split size is set to 128 MB, and the number of reducers is 16.

As seen from Fig 13, the completion time of WordCount application without failure is 240 seconds and killing the NodeManager of Application Master before 120 seconds is worse than killing after 150 seconds. In fact killing the NodeManager at 180 seconds has almost no impact on job completion time. The likely explanation is that after around 180 seconds most of the map tasks have finished executing with a few speculative map tasks running. At that point killing the NodeManager probably does not cause any map failure. The reduce job is also not hampered much because they were at the beginning stage and hence the overall job execution time is close to execution time without failure.

Finally, during both type of node failures, there was significant variation in job execution time while injecting node failures, especially for larger split sizes. The reason behind this variation is the speculative task execution adopted by Hadoop. Speculation is an option in Hadoop to launch backup tasks if slow tasks are detected on some cluster nodes. The backup tasks will be preferentially scheduled on the faster nodes. Whichever of the duplicate tasks finishes first becomes the one that is used in further operations. Hadoop starts speculative tasks towards the end of map phases. If the killed NodeManager was a slower node and Hadoop started backup speculative task in another faster node, then killing the slower NodeManager will not have a big impact on the overall job execution time. The speculative task running on the other node will finish within a short time and its output will be used instead of the dead node.

IV. Conclusions

The paper focuses on evaluation of the performance of two representative Hadoop MapReduce applications, with different execution parameters and under different failure scenarios. An important observation is that slowdowns are modest even with relatively high rate of task failures. This shows that the basic MapReduce model provides reasonable resistance to failure. The slowdown due to task failure is higher with relatively larger input split sizes, and peaks near the optimal split size. The latter is not surprising as efficient execution implies that there is little slack in execution that can be exploited for failure resistance. Another key observation is that complete node failure has a much more dramatic impact on performance than task failures; in case of node failure the default value of *yarn.nm.liveness-monitor.expiry-interval-ms* has a major impact on job completion time.

The paper also presents techniques to inject failures into MapReduce applications to simulate real world failures. We believe this is an important contribution towards future experimental research in resilient MapReduce computing. Finally, this paper can be considered early work towards accurate performance models for MapReduce applications considering node and process failures.

References

[1] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J Dongarra. An Evaluation of User-level Failure Mitigation Support in MPI. In *European MPI Users' Group Meeting*, pages 193–203. Springer, 2012.

[2] Douglas M Blough and Peng Liu. FIMD-MPI: a tool for injecting faults into MPI application. In *14th International Parallel and Distributed Processing Symposium. IPDPS*, pages 241–247. IEEE, 2000.

[3] Qiang Guan, Nathan BeBardeleben, Panruo Wu, Stephan Eidenbenz, Sean Blanchard, Laura Monroe, Elisabeth Baseman, and Li Tan. Design, Use and Evaluation of P-FSEFI: A Parallel Soft Error Fault Injection Framework for Emulating Soft Errors in Parallel Applications. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, pages 9–17. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016.

[4] Thomas Herault, Mathieu Jan, Thomas Largillier, Sylvain Peyronnet, Benjamin Quétier, and Franck Cappello. Emulation platform for high accuracy failure injection in grids. In *High Performance Computing Workshop*, pages 127–140, 2008.

[5] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[6] Message Passing Interface Forum. *MPI-2.2: Extensions to the Message Passing Interface*, September 2009. `http://www.mpi-forum.org`.

[7] Dmitry Mogilevsky, Gregory A Koenig, and William Yurcik. Byzantine anomaly testing for Charm++: Providing fault tolerance and survivability for Charm++ empowered clusters. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid Workshops (CCGRIDW'06)*, volume 2, pages 30–37. IEEE Computer Society, May 2006.

[8] M. T. Rahman, H. Nguyen, J. Subhlok, and G. Pandurangan. Checkpointing to minimize completion time for inter-dependent parallel processes on volunteer grids. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 331–335, May 2016.

[9] Bianca Schroeder and Garth Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, October 2010.

[10] Tom White. *Hadoop: The Definitive Guide* . O'ReillyMedia, Inc., second edition, October 2010.

[11] Yulai Yuan, Yongwei Wu, Qiuping Wang, Guangwen Yang, and Weimin Zheng. Job failures in high performance computing systems: A large-scale empirical study. *Computers & Mathematics with Applications*, 63(2):365–377, January 2012.