McTAR: A Multi-Trigger Checkpointing Tactic for Fast Task Recovery in MapReduce

Jing Liu, *Member, IEEE*, Peng Wang, Jiantao Zhou, *Member, IEEE*, and Keqin Li, *Fellow, IEEE*

Abstract—Cloud computing and big data technologies have gained great popularity in recent years. MapReduce is still one of the most efficient and well-adopted computing paradigms for providing big data services. MapReduce applications need to be executed on cloud platform where failures are inevitable. Hadoop is the de facto implementation of MapReduce, but it deploys a coarse grained and unsatisfactory fault tolerant services. The failed tasks are rescheduled from scratch to re-execute from the very beginning, which apparently brings amount of overload for failure recovery, and the whole job would be heavily delayed as failures happen. In this paper, we propose a novel multi-trigger checkpointing approach for fast recovery of MapReduce tasks, named a Multi-trigger Checkpointing Tactic for fAst TAsk Recovery (McTAR). As a finer-grained and better fault tolerance tactic, our McTAR employs multi-trigger checkpoint generation, push-pull combined intermediate data distribution and optimized failure task prediction techniques together to make the recovery task attempt be able to start at a specific progress according to the valid checkpoint for intermediate data. In this way, McTAR could effectively speed up the recovery process of MapReduce jobs and highly reduce the task recovery delay.

Index Terms—Checkpoint, failure prediction, fault tolerance, Hadoop MapReduce, task recovery

1 Introduction

1.1 Motivation

With the rapid development of cloud computing and big data processing technologies, more and more services are provided over cloud computing platforms. In reality, MapReduce is still one of the most efficient and well-adopted computing paradigms to develop big data related services, which always utilizes many computing nodes over the clouds. Hadoop is the de facto implementation of MapReduce. However, they always cannot to fulfill their reliability requirements, because failures are no longer exceptions in cloud computing environments [1]. Besides those common hardware failures, software aging and design errors, such as data inconsistence or missing values, bring more vulnerabilities to the whole services quality [2]. Thus, it makes MapReduce suffer from poor performance and service incontinuity under failures. The essential reason is that in the Hadoop MapReduce, a basic and coarse grained fault tolerant service is deployed, that is, the failed tasks are rescheduled from scratch to re-execute from the very beginning. It apparently brings amount of overload for failure recovery, and the whole job was heavily delayed [3]. To better cope with such dilemma, optimizing current fault tolerant service in Hadoop to make it have stronger fault tolerant capability and improved service availability is indispensable and significant.

There are two major fault tolerance tactics in cloud services environment, namely replication and checkpoint [4]. As for the replication, no matter replicating hardware nodes or software components, the same back-ups are necessary. No matter whether the back-ups run in parallel or keep standby until the primary one fails over, the resource will be doubled to meet the requirement of fault tolerance. Thus, replication is not an ideal solution for the data-intensive application scenario. Another solution is to checkpoint the state of current running process onto a stable storage and resume the process based on the latest checkpoint in failure recovery. In the Hadoop MapReduce workflow, the important data worthy of checkpointing are intermediate results generated continuously during the *Map* phase. They help to re-execute the failed task from the last checkpoint, which skips all successfully finished range. However, some error-prone design aspects accompanying should be considered carefully. First, saving all these data significantly increases computing overloads. Then, storing intermediate results on stable storage also results in intensive network bandwidth usage, which interferes computing nodes to exchange data in time. At last, the checkpoint files to be fetched would also create heavy workload on both bandwidth and I/O. Thus, taking the snapshot of all the running processes as checkpoints directly is not a suitable choice either. That is, a better fault tolerance solution for Hadoop MapReduce based on checkpoint technology needs more intensive optimization.

1.2 Our Contributions

In this paper, we propose a novel multi-trigger checkpointing approach for fast recovery towards MapReduce tasks,

- *J. Liu, P. Wang, and J. Zhou are with the Inner Mongolia Engineering Laboratory for Cloud Computing and Service Software, College of Computer Science, Inner Mongolia University, Hohhot, Inner Mongolia 010021, China. E-mail: liujing@imu.edu.cn, wangpeng9302@gmail.com, cszhoujiantao@qq.com.*
- *K. Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA. E-mail: lik@newpaltz.edu.*

Manuscript received 5 Oct. 2018; revised 23 Jan. 2019; accepted 5 Mar. 2019. Date of publication 9 Mar. 2019; date of current version 9 Dec. 2021. (Corresponding author: Jing Liu.)
Digital Object Identifier no. 10.1109/TSC.2019.2904270

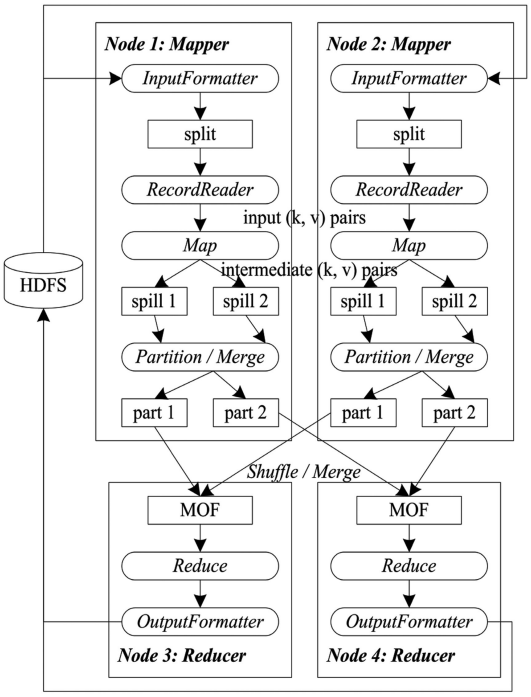


Fig. 1. The workflow of the original Hadoop MapReduce.

named a Multi-trigger Checkpointing Tactic for fAst TAsk Recovery (McTAR), an enhanced version of our previous work [5]. McTAR makes minor revisions to the original MapReduce framework, which achieves better fault tolerant service capabilities to handle task or node failures with optimized checkpoint tactics. Major contributions of this paper are listed as follows:

- *The McTAR framework:* McTAR composes of a bunch of specific tactics which help to speed up the recovery of MapReduce jobs, such as multi-trigger checkpoint generation, push-pull combined intermediate data distribution and optimized failure task prediction. The fundamental idea of our McTAR is to divide spills into groups instead of merging them into one single file. Such spills are generated as temporary output when buffer overflows in the Map phase. In this way, the intermediate data could be duplicated in time so that the recovery task attempt is able to start at a specific progress according to the valid checkpoint generated along with spills. It could reduce the task recovery delay caused by failures.
- *Multi-trigger checkpointing policy:* McTAR introduces two kinds of checkpoints of intermediate results in the Map phase, named as space-triggered checkpoint and time-triggered checkpoint. The former is generated along with the spilling process, which is more proactive, and the latter is created periodically. The costs of space-triggered checkpoint tend to be negligible compared with time-triggered checkpoint, but time-triggered checkpoint is more steady and capable of dealing node failures. Thus, this multi-trigger checkpointing tactic enables failed tasks to resume from certain progress instead of starting over.
- *Push-pull combined intermediate data distribution:* In the original Hadoop, intermediate data are pulled in a single file by the $Reduce$ phase when it receives the completion event of the Map phase. Beyond that, in our McTAR, the intermediate data could be pushed to the corresponding $Reduce$ nodes at a certain interval of time. Combining these two modes of data transmission, McTAR makes it possible to replicate intermediate data in time without too much cost, and the completeness of intermediate data is ensured.
- *Optimized failure task prediction:* For better fault tolerance performance, we propose a fault prediction method based on monitoring specific running indicators of cluster nodes. McTAR does not need to wait for timing out to detect a node failure, so that the recovery task can be scheduled in advance, which helps to accelerate the recovery process when node failures happen.
- *Comprehensive evaluation:* We use WordCount, the most typical application of MapReduce, to evaluate the cost and performance of McTAR under task failures and node failures in several scenarios. The execution time at different failure rate helps to verify the effectiveness of our McTAR approach, and realistic data are adopted in our evaluation to better understand both pros and cons of McTAR.

It should be noted that our previous work [5] just presents preliminary ideas and simple numeric analysis. However in this paper, we improve the details of checkpointing tactics, add failure prediction method, and what is more, we demonstrate sufficient practical experiments to evaluate the effectiveness of our McTAR. As McTAR introduces several fault tolerant capabilities into the original version of Hadoop MapReduce, we first address the execution flow and design flaws in failure recovery of current MapReduce and discuss default fault tolerance strategies in Section 2. Then, the design architecture and implementation details of McTAR are proposed in Section 3. The performance evaluation of McTAR is shown in Section 4 under different failure scenarios. At last, we discuss related work in Section 5 and conclude the paper in last section.

2 Fault Tolerance in MapReduce

2.1 Workflow of Hadoop MapReduce

MapReduce was introduced by Google in 2004 [6]. Currently, Hadoop is still the de facto implementation of MapReduce [7]. Our work is based on Hadoop 2.7.3 version, which has been adopted as a new generation of MapReduce with YARN framework [8], [9]. Intuitively speaking, MapReduce processing consists of three phases, Map phase, $Shuffle$ phase and $Reduce$ phase. Large-scale input data are split into independent chunks to make analysis process run in parallel. Two user-defined functions, $Map : (k1, v1) \rightarrow list(k2, v2)$ and $Reduce : (k2, list(v2)) \rightarrow (k3, v3)$, are key programming aspects to process data structured in $(key, value)$ pairs. At first, the original data have to be transformed into $(key, value)$ pairs so that the Map function could be applied. During the Map phase, a number of intermediate data will be generated. Then we have the Map Output Files (MOF), which will be shuffled and sorted, and finally sent to the $Reduce$ phase. The returns of the $Reduce$ function are collected as the desired results.

The workflow of MapReduce in Hadoop is presented in Fig. 1. For better understanding, we show the workflow with

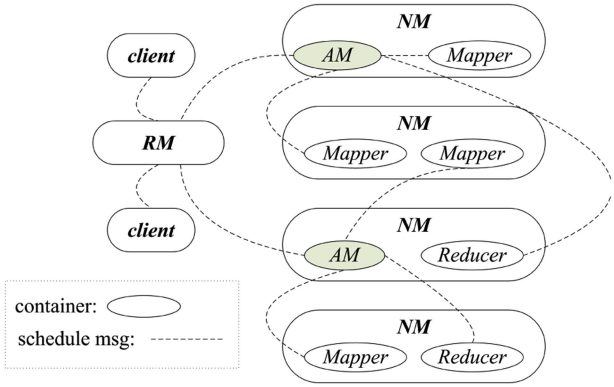


Fig. 2. Basic components in Hadoop MapReduce.

four computing nodes as a representative. Node 1 and node 2 run the Map tasks, named as *Mappers*, while node 3 and node 4 are responsible for the $Reduce$ tasks, named as *Reducers*. For performance reasons, the number of *Reducers* should be less than that of *Mappers*, and each of *Reducers* is responsible for specific keys assigned to it. The original data are stored on the HDFS. When a specific job is launched, the input file will be divided into several small logical units which are called split, each of which is assigned to a Map task. For different types of input data, implementation of *InputFormatter* function varies a lot. Given a specific split, *RecordReader* is performed to generate the initial (k, v) pairs which are the input of a Map function corresponding to the $(k1, v1)$ mentioned before. During the Map phase, the returns could be too large to maintain in memory. In that case, the buffer will be flushed and a spill is created. If the application is data-intensive, the spill process can be executed for a couple of times; otherwise, there might be no spill for a long period of time.

After the spill process, intermediate data are divided into several parts according to the hash value of keys. The number of parts is relevant to the number of *Reducers*, which can be set in the configuration file. Right before the intermediate data partition, there is an optional process called combiner. We could simply understand it as pre-reduce, which operates only on data generated by a single machine. After these operations, MOFs are ready to be fetched in a form like $list(k2, v2)$. The fetcher threads running on *Reducers* copy the MOFs as input to $Reduce$ function, which is called shuffle phase. Shuffle accounts for the most proportion of network traffic in MapReduce. An optimization for shuffle could bring acceleration of job execution. The retrieved MOFs should be merged if they are large enough. Shuffle produces data as $(k2, list(v2))$. Finally, the $Reduce$ function will work after the shuffle phase finishes, which means the sooner the shuffle ends, the faster a job completes. Then, the $OutputFormatter$ function takes the returns of $Reduce$ function as input, i.e., $(k3, v3)$, and write the results to HDFS.

2.2 Fault Tolerance in Hadoop MapReduce

In cloud computing systems, three types of faults are mainly considered, that is, byzantine faults, fail-stop faults and fail-stutter faults [7]. Hadoop MapReduce can tolerate the fail-stop faults and the fail-stutter faults. Treatments to byzantine faults are beyond the original design target of MapReduce. Fail-stop faults in MapReduce typically include node failures and task failures. Hadoop MapReduce often contains one node called the master and other nodes called workers. The master takes charge of scheduling tasks, and workers are responsible for executing Map function or $Reduce$ function. That is, a worker could be a *Mapper* or a *Reducer* or both of them. Deploying a back-up master is quite effective to tolerate the fail-stop faults. However, as for workers failures or task failures, Hadoop implements a simple fault tolerant technology, where the failed tasks are rescheduled from scratch to re-execute from the very beginning. Fail-stutter faults in Hadoop MapReduce are recognized as a kind of slow tasks. When they are detected, a speculative task attempt, processing the same input data as the slow task, is performed exactly in hope that this speculative attempt will finish sooner. In a word, fault tolerance tactics in the original MapReduce are elementary because of re-execution approach and may cause long delay when node or task failures happen.

Next, we give more detailed explanation to fault tolerance mechanism used in Hadoop MapReduce. As shown in Fig. 2, a job submitted by a client consists of multiple tasks including Map tasks and $Reduce$ tasks. When a job is submitted from a client, an Application Master (AM) is launched to schedule tasks. Task runs in the container which is collection of physical resource allocated by Resource Manager (RM). RM is the scheduler in YARN. Node Manager (NM) takes care of the individual compute node in Hadoop cluster. Since Hadoop 0.21, checkpoint is provided on the task level, that is, the basic unit during recovery is a container with a task running in. All tasks are separated into two parts: finished and unfinished. When the job crashed, the processes do not have to start all over, because the finished tasks will be skipped, which saves a lot of time when failure happens. As for the tasks running in progress when the last attempt crashed, Hadoop will restart the task just like the unfinished ones regardless of the progress. Apart from that, if the AM is running without failure, the failed tasks will be rescheduled and restarted from the beginning, which means the execution time will be doubled if the task fails to handle the last record assigned to it. Since tasks cannot continue after failures happen in Hadoop, we propose a fine-grained checkpointing tactic to make it possible for failed tasks to keep going after recovery process, which would be able to save a lot of time.

3 Design and Implementation of McTAR

We present a multi-trigger checkpointing strategy for MapReduce application, which brings negligible extra cost and helps a specific task that needs recovery from failures to start from a specific progress. In this way, the delay penalty in case of failures could be reduced.

3.1 Architecture of McTAR

Task is the minimal unit of scheduling in the original version of Hadoop, which makes it impossible for rescheduled task to resume from where the failure happened. Furthermore, the only provided fault tolerant approach is to execute the whole task all over again, so that the data generated by *Mappers* could remain completed. The main goal of McTAR is to make full use of such completed parts generated before the failure happens. By preserving the necessary progress

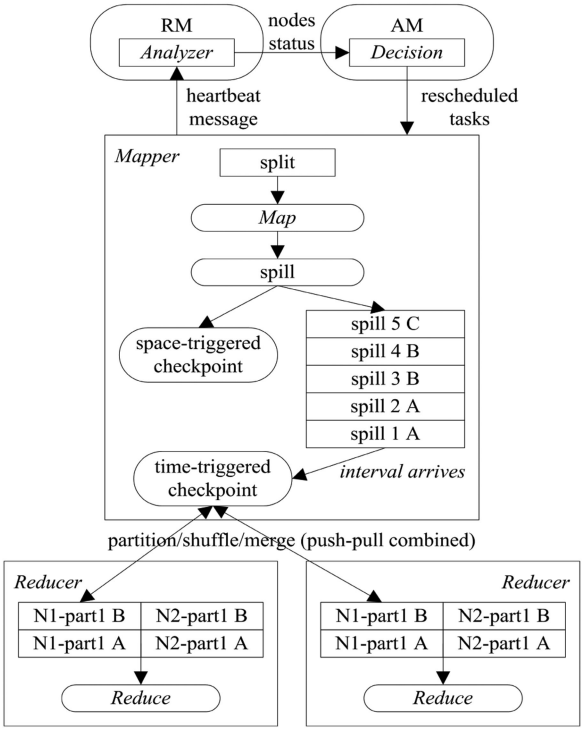


Fig. 3. The workflow of McTAR.

information and key location information, McTAR makes it possible for tasks to achieve faster recovery.

Fig. 3 shows the regular workflow of our approach. From the process perspective, the difference between McTAR and the original version of Hadoop MapReduce is mainly from two parts. First, in the original version of Hadoop, the detection of failure is based on the heartbeat between NMs and AM. Once a node failure happens, the AM has to wait until timeout. However, in our McTAR we present an proactive failure prediction where hardware running indicators data are transmitted via heartbeat messages and AM calculates the number of tasks to be rescheduled according to the resource usages. That is, tasks running on a node which is likely to be suffering from performance degradation will be rescheduled as speculative tasks. Second, the procedures after spilling process are different. Taking the efficiency of execution without failure into consideration, we choose to serialize the meta data as checkpoints only at the moment that spills are generated. Otherwise, the extra cost of keeping tracking of each single record may slow down the whole job. *Mappers* take splits as input just like the way that Hadoop originally does. With the generation of a spill, a space-triggered checkpoint is created.

Since the buffer will reach to a bounded limit, the spilling process is going to launch an asynchronous thread to persistence the intermediate data produced by *Map* function. We choose to create a space-triggered checkpoint at this very moment to decrease impacts of interfering the normal execution. The checkpoint file and the spilled files share the same directory, so that the cleaning process in Hadoop could delete them simultaneously. Thus, the spilled files and the checkpoint file are able to remain consistency in this way.

Instead of merging the spilled files until the *Map* task is finished, McTAR merges the files when the time-triggered checkpoint interval arrives. For a certain period of time, the time-triggered checkpoint thread would be launched. It performs the spill process by force, no matter the output buffer reaches to the limit or not. Along with the creation of time-triggered checkpoint, the spills that generated since the last time-triggered checkpoint was created would be merged into one single file and sent to the corresponding *Reducers*. As shown in Fig. 3, those intermediate data belonging to a sending process are marked with same letter, such as "spill 1 A" and "spill 2 A", and the spills marked with 'C' have not been sent yet.

The architecture of McTAR requires the *Reducers* being launched as soon as the job is submitted, so that the intermediate data sent by *Mappers* can be received in time. As the time-triggered checkpoints are being created one by one, the *Reducers* would get all the corresponding data in group. Since the output files from *Mapper* have to arrive to *Reducer* one way or the other, our approach does not bring too much cost in network aspect compared with the original version of Hadoop MapReduce.

3.2 Push-Pull Combined Intermediate Data Distribution

In McTAR, there are two ways for transmitting data from *Mappers* to *Reduces*. The first one is the sending process mentioned in Section 3.1, which is named as *proactively-push*, and the other one is called *pull-on-demand*. Combining these two ways of data transportation together, we could ensure the data integrity in McTAR.

For the most situation, we perform *proactively-push* to transmit the intermediate data more timely. *Mappers* invoke *proactively-push* along with time-triggered checkpoint. On one hand, these data are required by *Reducers*. On the other hand, it provides a replication for these data, which makes it possible to avoid recomputation after node failure happens. The last *proactively-push* would be performed when the progress of the *Map* task reaches to 100 percent. After that its output file will be completely transmitted to all corresponding *Reducers*. By the way, the combiner comes into effect if the number of spills about to be merged is greater than three by default. Comparing to sending replication whenever a spill is generated, it takes the advantage of combiner and decreases the extra cost in network furthermore, especially for the data-intensive applications.

When a *Reducer* is recovered from a node failure, it would have lost the corresponding map output files. In this case, *pull-on-demand* is more effective. It transmits data just like the way that the original version of Hadoop does, except for that the *Reducer* becomes the initiator. The *Reducer* performs the *pull-on-demand* based on the relevant events in message queue, and the fetch threads are going to assemble all the missing parts of map output files required by this *Reducer*. After executing *pull-on-demand*, the failed *Reduce* tasks will continue as normal.

In summary, *proactively-push* is used as the primary way of data transportation in McTAR for a better fault tolerant capability, and *pull-on-demand* is an alternative scheme for *Reducer* when node failure happens. With this push-pull combined intermediate data distribution, the output files of *Map* tasks are replicated in time without involving much cost and do not require for recomputation in most cases.

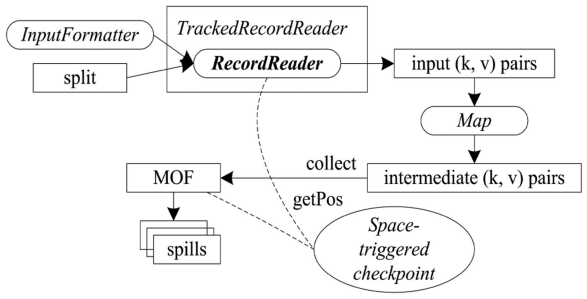


Fig. 4. Process for creating space-triggered checkpoint.

3.3 Multi-Trigger Checkpointing Policy

We present a multi-trigger checkpointing policy, including space-triggered checkpoint and time-triggered checkpoint. The former is designed to deal with task failures and the latter is saved on HDFS, which makes it possible to preserve meta data after node failure.

3.3.1 Space-Triggered Checkpoint

Space-Triggered checkpoint is stored on the local storage with the spills, and it saves the runtime information of a single task. Since no network cost is involved by space-triggered checkpoint, we choose to generate one for each spill, which provides a finer-grained progress. Thus, the priority of space-triggered checkpoint is higher compared with time-triggered checkpoint. The space-triggered checkpoint is saved as a triple, containing task ID, offset and spill ID. The task ID is the identifier of a single task, which remains immutable while failure happens. The spill ID represents the specific spill serialized along with current space-triggered checkpoint. The offset is the most important part, which saves the progress of a running task.

Fig. 4 shows the generation details of space-triggered checkpoint. Given the *InputFormatter* and split, we can get the binary file belongs to this *Map* task. The *RecordReader* wrapped by *TrackedRecordReader* is used to generate the input key value pairs, the only acceptable input for *Map* function, according to the given binary file. Multiple implementations of *RecordReader* are provided by Hadoop. Taking *LineRecordReader* as an example, the input keys are the line number of the original file and the input values are the content of each lines. The core function of a *RecordReader* is *nextKeyValue*, and the implementations of this function should move the cursor to the position where the next key value pair locates on the binary file. A variable named *pos* is used to represent this cursor, which can be saved as offset for checkpointing. With the *pos*, we can directly locate the unfinished part of the split. Thus, we provide an interface for the *Map* task to get the current *pos* in *RecordReader*.

The output buffer in a *Mapper* keeps collecting the intermediate data, which are the output of *Map* function, until the buffer verges to overflow. At that moment, a spill is created along with corresponding space-triggered checkpoint, which saves the spill ID, current *pos* as offset and the ID of this task. Those spills share the same directory with the checkpoint file, so that the space-triggered checkpoint is disabled when relevant spills is no longer accessible. Or else the recovery task would skip the finished ranges at last time but lose corresponding intermediated data.

3.3.2 Time-Triggered Checkpoint

Along with the intermediate data being sent, relevant meta data will be sent to stable storage, i.e., HDFS in our case, as time-triggered checkpoint. Based on time-triggered checkpoint, our McTAR makes the *Mapper* resume from the last checkpoint, and the *Reducer* fetch the relevant map output files when node failure happens. According to meta data, the execution of recomputation process can be accelerated.

The time-triggered checkpoint backups the checkpoint file to a remote node through network, and the intermediate data created before this time-triggered checkpoint have to be sent to *Reducers* to maintain the completeness of data after recovery. Thus, the increase of network traffic must be acceptable. To minimize the overload, we choose to send the replication of the spills to the *Reducers* that will consume them. This process is completed by *proactively-push* we mention in last section.

The time-triggered checkpoint is designed to be created at every interval for two reasons. On one hand, the spills in the compute-intensive applications could hardly be created because the computation lasts for most of the time, and the output may not be so large enough to trigger the spill process. In that case, making a checkpoint at the cost of interruption a task could save a great deal of time for computing if failure happens. On the other hand, when it comes to data-intensive applications, repeated data are very common. Taking WordCount as an example, the frequency of some words could be very high. To take the advantage of the combiner, we make time-triggered checkpoint for a certain period of time, so that the amount of data through the network can be smaller.

Different from the triple definition of space-triggered checkpoint, we use a variable called *merge-times* in the time-triggered checkpoint. It is a number increased by 1 each time a time-triggered checkpoint is made. For the nth time-triggered checkpoint, the spills created between the $(n-1)$ th time-triggered checkpoint and this one would be merged into a single map output file named after *merge-times* n. After that, a data-sending event is pushed to the message handling queue, then the *Reducers* launch fetching process to copy data from *Mapper* according to the given file name as receiving the data-sending event.

Considering the space-triggered checkpoint is controlled by spill process and the time-triggered checkpoint is controlled by the time cycle called *merge-period*, there is a chance that time-triggered checkpoint would start right after another one. It causes a spill process being launched with no data in the buffer. To avoid this, we propose another parameter named *merge-threshold*. We keep track of the time stamp of last spill thread, and make sure that the time-triggered checkpoint starts the spill thread only if the subtraction result of current time stamp and the time stamp of last spill is greater than *merge-threshold*.

Another meta data sent to HDFS along with time-triggered checkpoint is a list shown in Fig. 5, we call it Map Info List (MIL). The size of MIL is the number of *Reducers*. Each element in that list is a set contains the offsets corresponding to a certain *Reducer*. As we known, the split is turned into key-value pairs before it applies to the *Map* function, then it takes each of those key-value pairs as input and returns a group of intermediate data, which will be sent to the *Reducer*. When there are multiple *Reducers*, a strategy, typically a hash

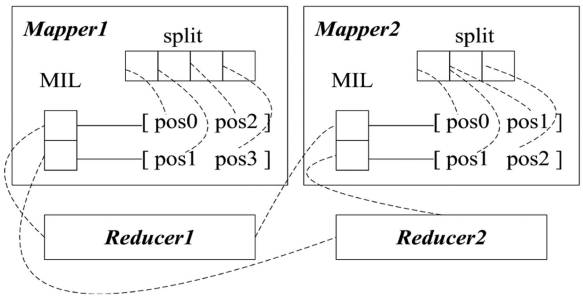


Fig. 5. Structure of map info list.

function on the key, is needed to decide which *Reducer* the intermediate data belong to. Saving the offsets of these keys as metadata is designed for recomputing the corresponding intermediate data for a particular *Reducer*. As it is shown in Fig. 5, if *Mapper2* is required for recomputing the data sent to *Reducer2* before, the second element in MIL indicates only data at *pos1* and *pos2* would generate intermediate data for *Reducer2*. Thus the range from *pos0* to *pos1* could be skipped without worrying about losing any data.

3.4 Dealing with Failures

By implementing the multi-triggered checkpointing policy, McTAR is able to preserve the progress as meta data along with the corresponding intermediate data. We talk about dealing with failures based on these checkpoints next. Algorithm 1 shows the pseudocode of initializing a Map task. $TaskAttemptID$ is greater than zero, which means that the current task is a recovery task, so we can skip some computed ranges if checkpoints could be found. Since the space-triggered checkpoint is finer, the time-triggered checkpoint is used only if no space-triggered checkpoint is available. If neither of them is found, it means no checkpoints have been created in the last task attempt and the new one is scheduled to created. In this case, a task with failures has to restart from very beginning like a new task.

Algorithm 1. Initiate Map task

Require: TaskAttemptID $attemptID$, JobConfigure $conf$
1: FileSystem fs = FileSystem.get($conf$);
2: scp = new Path($getOutputFile()$, "checkpoint");
3: tcp = new Path("HDFS://master:9000"+$getTaskID()$);
4: **if** $attemptID \geq 0$ **then**
5: **if** scp.exist **then**
6: (TrackedRecordReader)$input.setPos(scp.pos)$
7: **else if** tcp.exsit **then**
8: (TrackedRecordReader)$input.setPos(tcp.pos)$
9: **else**
10: (TrackedRecordReader)$input.setPos(0)$
11: **end if**
12: **else**
13: (TrackedRecordReader)$input.setPos(0)$
14: **end if**
15: $input.initialize()$

3.4.1 Task Failure

When a task fails, it sends a completion event with a failed status to AM. This AM will launch a retry attempt for the failed task. Space-triggered checkpoint is designed for task failure, because the task failure will not bring damage to the persistence files on the local storage. It helps the retry attempts to skip the finished part of split, which is not feasible in the original Hadoop MapReduce. The spills are stored in a directory name after taskID and removed after merging, so the recovery task attempt could just merge all the spills before the next time-triggered checkpoint without concerning about how many spills belong to the next *proactively-push*.

Another kind of task failure is caused by fail-stutter faults instead of fail-stop faults. In that case, McTAR launches a speculative task and kills the one suffering from stutter, instead of making the speculative task run in parallel. We make this choice for two reasons. First we have to maintain the consistency of two task attempts of a task due to time-triggered checkpoint, which means more interruption of the execution process. Second, the speculative task is also benefit from the checkpoint mechanism, so that the restarted task is unlikely to be slower than the original one. However, the space-triggered checkpoint is not able to cover all task failures. Time-triggered checkpoint could conduce to recovery when the recovery task is scheduled on another node, which is unlikely for fail-stop fault because of locality, but quite possible for fail-stutter faults.

3.4.2 Node Failure

If node failure happens only on *Mapper*, the recovery task could use time-triggered checkpoint to skip the finished part of split just like dealing with task failure. Because the intermediate data have already been sent to corresponding *Reducers*, and those data will only be used by them. A $Reduce$ task recovering from node failure will launch the *pull-on-demand* to get all the missing data it needs. If the *Mappers* did not suffer node failure before, no recomputation is needed either.

The only situation that requires for recomputation is the node failure happened on both *Mapper* and *Reducer* one after the other. In that case, the *mapper* fails after *proactively-push* and restarts without caring about the finished parts. It means the pushed data existing only on the corresponding *Reducers*. Once one of the *Reducers* runs into node failure, part of the intermediate data will be lost. Algorithm 2 shows the recomputation process with MIL.

Algorithm 2. Recomputation with MIL

Require: TaskStateInternal $taskState$, RecordReader $input$
1: **if** $taskState.getState()$ == RECOMPUTE **then**
2: tcp = new Path("HDFS://master:9000"+$getTaskID()$);
3: MapInfoList $mil = tcp.getMil()$;
4: $posSet = mil.getPos(taskState.getReducer())$
5: **for all** pos in $posSet$ **do**
6: $input.addPos(pos)$
7: **end for**
8: $Mapper.run()$
9: **end if**

Once a Map task is identified as a recomputation task by internal state, it would fetch the time-triggered checkpoint from HDFS. By searching the specified $Reduce$ task ID, a Map task would be able to get all offsets that generated intermediately for this *Reducer*. After iterating all the corresponding offsets and applying them to the Map function, the missing data caused by node failure will be recomputed

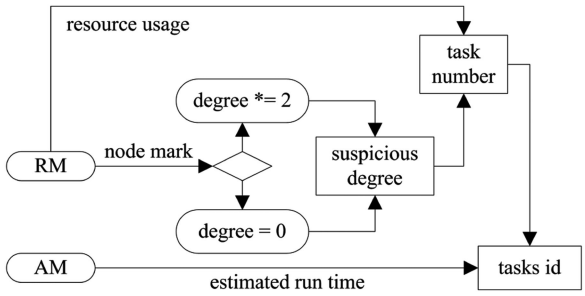


Fig. 6. Workflow of optimized failure task prediction.

on purpose instead of all over again. But the fact is that the MIL is only taking effects in extreme circumstances that requires for recomputing.

3.5 Optimized Failure Task Prediction

By now, we introduce the way of McTAR dealing with failures, and the recovery can be accelerated a step further if failures could be detected more timely. In the original version of Hadoop, the detection of failure is based on the heartbeat between node manager and AM. A task sends a completion event with status 'killed' or 'failed' when it terminates unexpectedly, and the AM is able to detect failure in time. But once a node failure happens, the AM has to wait until timeout, which delays the execution of recovered tasks.

A novel failure prediction mechanism based on software rejuvenation [10], [11] is proposed for our McTAR. We take the hardware running indicators into consideration. Tasks running on a node which is likely to be suffering from performance degradation are rescheduled as speculative tasks. Fig. 6 shows the workflow such failure task prediction.

We placed an analyzer as a plug-in in the RM. For each time the heartbeats from NMs are collected, the analyzer provides the node marks of each NM, which indicates the node is running normally or likely to fail in a short time, according to the metrics given by heartbeat. On receiving those node marks, the AM maintains a group of variables called suspicious degree of each node. If a node is marked as suspicious node continuously, the degree of it would be doubled and set to zero otherwise. We make the aging degree decision according to different levels of three metrics. C_i and M_i shows the CPU usage and available memory of certain node *i*. EAT indicates the expected time of next packet arrival at a node in regular case, and it is estimated as follows based on studies in [10]

$$EAT_{k+1} = \frac{1}{n}\left(\sum_{i=k-n-1}^{k}(T_i - \Delta T_i \times i)\right) + (k+1)\Delta t_i.$$

Without loss of generality, based on the Pareto principle (also known as the 80-20 rule), we give an exemplified aging degree decision method. Specifically speaking, if the packet for computing EAT delayed a lot, C_i is beyond 80 percent, and M_i is lower than 20 percent, we set the suspicious degree to 3. It indicates the most serious issues, such as node crash, are very likely to occur. If two of above three conditions are both satisfied, we set the suspicious degree to 2. It indicates the important issues, such as performance degradation, may occur. Otherwise, we set the suspicious degree to 1 to indicates that nodes are working well.

After time-triggered checkpoint is made, AM calculates the number of tasks to be rescheduled according to suspicious degree and resource usage by the following formula:

$$\min\left(T_n, \max\left(0, \left\lceil degree \times \left(1 - \frac{T_w}{T_r}\right) - \frac{1}{2}\right\rceil\right)\right)$$

T_r and T_w represent the number of tasks running and waiting in the Hadoop cluster separately. One minus the ratio of T_w to T_r shows the usage of resources in cluster. The rounding of the product of suspicious degree and the resource usage decides how many tasks to be rescheduled. T_n represents the tasks of this application on a specific node, thus all of the tasks on a node need to be rescheduled in the worst case. Once the number of tasks to be rescheduled is calculated, the AM needs to decide which tasks should be rescheduled first. McTAR selects tasks like the way that original Hadoop does, that is, it is based on the estimated runtime of each task by its progress and time escaped. Tasks that need more time to finish have the higher priority of being rescheduled.

It is worth mentioning that most of the stutter problems are caused by node failure. Using our optimized failure task prediction, the potential stutter tasks on suspicious node could be rescheduled before being detected, which accelerates the failure awareness and recovery process.

3.6 Modifications Towards Original Hadoop MapReduce

By far, the tasks can resume working after failures using McTAR. Even if the recomputation is inevitable in the extreme case, parts of the data processing could be skipped. In summary, we make following revisions based on original Hadoop MapReduce to implement our McTAR.

- The meta data required by task recovery are preserved through multi-trigger checkpointing mechanism. The combination of space-triggered checkpoint and time-triggered checkpoint provides both flexibility and stability.
- The intermediate data generated by *Mapper* are stored temporarily until being copied by fetching process when time interval arrives. The output files are distributed before finishing processing all data.
- When failure happens on *Reducer*, the intermediate data distribution remains the way that Hadoop does. The push-pull combined intermediate data distribution is constituted then.
- McTAR could perform recomputation for a particular *Reducer* according to MIL, which helps accelerate the recovering process in worst case.
- An optimized failure task prediction based on software rejuvenations theory is proposed to enhance failure awareness and recovery.

4 Experiments and Evaluation

First, experiments are designed to measure the performance of McTAR when no failure happens. That is, we need to confirm that our McTAR does not bring in too much execution cost. Then under different failure scenarios composed of task failures or node failures, we do sufficient experiments to evaluate the advantages brought by our McTAR.

TABLE 1
Node Configurations in Experiments

CPU	1 core 2.9 GHz
Memory	2 GB
Disk	200 GB
Network	Gigabyte Ethernet NIC
OS	Ubuntu 14.04

That is, we like to verify that McTAR should spend less duration time when task recovery is involved.

Our McTAR is implemented based on Hadoop 2.7.3. We construct a Hadoop cluster with four nodes. Each node is a virtual machine with configurations as shown in Table 1. WordCount is a classic application of MapReduce framework, so we used it as a representative to evaluate the performance of McTAR. The input data of WordCount are generated by the random text writer application.

Table 2 gives the appropriate parameter assignments for our experiment. First, Parameter *mapred.min.split.size* decides the size of input data for each *Map* task. The smaller this parameter is, the more tasks needs to start for a job, which requires more resource for scheduling. With the growth of input data for a task, Hadoop is influenced by failures more seriously. In our experiments with WordCount application, we set it to an appropriate value, i.e., 256 MB. Second, the values of next three parameters are decided according to the intrinsic design of McTAR. Specifically, parameter *reduce.slowstart.completedmaps* represents how much *Map* tasks need to be finished when a *Reduce* task starts to launch, parameter *reduce.rampup.limit* represents how much *Reduce* tasks could start when *Map* tasks complete, and parameter *reduce.preemption.limit* represents how much *Map* tasks could take over the running resources of *Reduce* tasks when resource deficiency occurs. Therefore, in order to make *Reduce* tasks be ready to perform at the beginning of the job and under the situation of execution resource deficiency in McTAR, these three parameters should be assigned to zero, one hundred percent and zero. Third, parameters *merge-period* and *merge-threshold* are relevant to time-triggered checkpoint, which is explained in Section 3.3.2. It should be noted that these two parameters could be optimized according to the application and input data scale, so in our experiments with WordCount application, without loss of generality, we set them as 90 and 10 seconds for better performance.

4.1 Analysis of Execution Cost Without Failures

We run WordCount application on Hadoop and McTAR separately with no failure to analyze the workload brought by McTAR. Fig. 7 shows the execution duration with different data size that varies from 1 to 50 GB. As we can see, they stick to each other very closely. The workload of McTAR is mainly caused by the creation of time-triggered checkpoints. With more time-triggered checkpoints are generated, the costs tends to more expensive. However, for long time running application with larger data scale, if we prolong the creation period of time-triggered checkpointing appropriately, the increased execution costs in McTAR could be slight consequently. Another thing should be noted that the time duration of whole job execution will actually be less affected by the input data scale if enough *Map* tasks could be executed in parallel in cloud datacenter, however, limited to our experiment conditions, the parallel degree of *Map* tasks are not big enough, so in Fig. 7, the time durations are increased with the increasing input data scale.

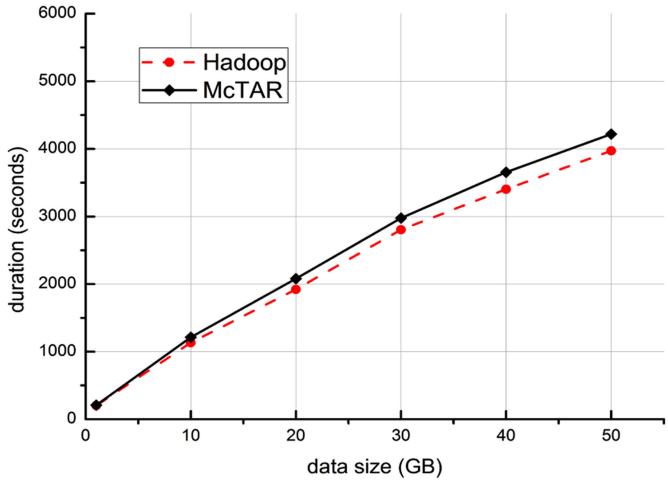


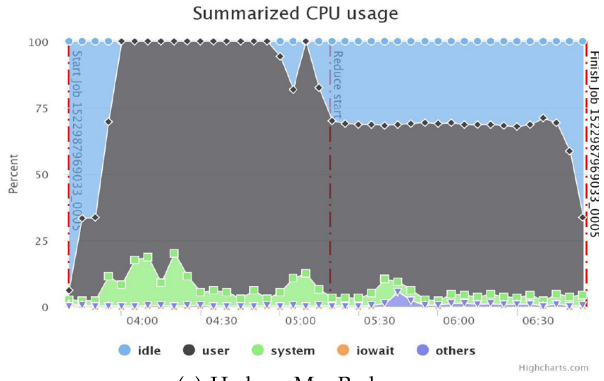
Fig. 7. Execution duration with no failure in both Hadoop MapReduce and McTAR.

Besides, we use HiBench, a bundle of benchmark applications for Hadoop, for a clear view of hardware running indicators of Hadoop MapReduce and our McTAR. There are no significant differences in CPU usage and storage usage as shown in Figs. 8 and 9. The reasons are discussed as follows. First, the checkpointing mechanism is implemented in asynchronous threading and will not interrupt the normal execution process. Thus, McTAR will not bring distinct and considerable execution cost. Second, an extra spill is made for each time-triggered checkpoint, and space-triggered checkpoint also acquires for I/O operation to serialize the meta data. They all need extra storages. However, the whole size of required storage is pretty small and acceptable, compared to dealing with input data.

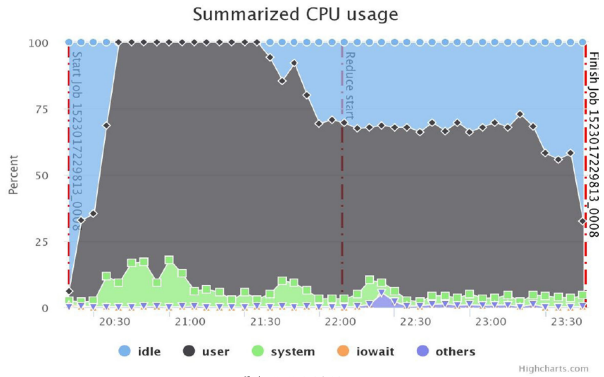
The network throughputs show the main differences between McTAR and Hadoop MapReduce. As we can see in Fig. 10, the use of network is brought forward in McTAR, which means *proactively-push* transmits the intermediate data before a *Map* task is completed. Since *proactively-push* transmits parts of the intermediate data once at a time, the peak value of network throughputs will be decreased significantly. The amount of data go through network of McTAR and Hadoop MapReduce stays closely to each other when no combiner is deployed. But McTAR needs to transmit more data when the combiner is used. Nevertheless, the workload is totally acceptable for most cases.

TABLE 2
The Basic Parameter Assignments in McTAR Experiments

Name	Value
mapred.min.split.size	268435456 (256 MB)
reduce.slowstart.completedmaps	0
reduce.rampup.limit	1 (100%)
reduce.preemption.limit	0
merge-period	90 (seconds)
merge-threshold	10 (seconds)

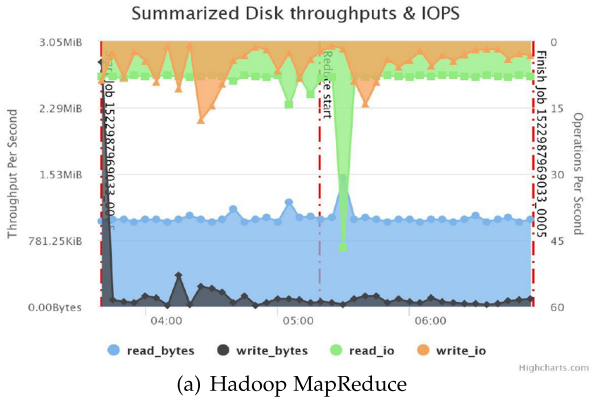


(a) Hadoop MapReduce

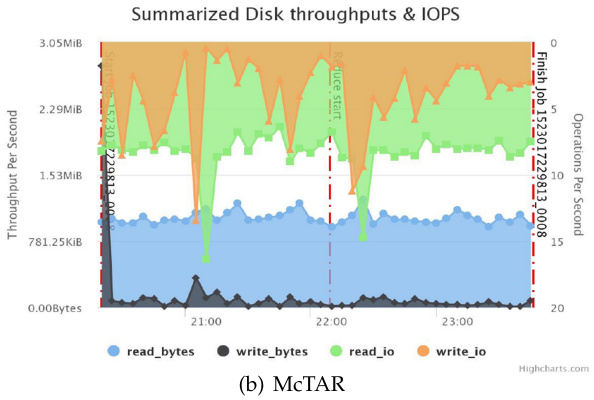


(b) McTAR

Fig. 8. Comparison of CPU usage in Hadoop MapReduce and McTAR.



(a) Hadoop MapReduce



(b) McTAR

Fig. 9. Comparison of storage and I/O usages in Hadoop MapReduce and McTAR.

4.2 Analysis and Comparison of Fault Tolerance Capability

We take different failure scenarios, i.e., no failure, task failure and node failure, into consideration, and each of them could happen during executions of *Map* function or *Reduce* function. Except for no failures occur on both the *Mapper* and the *Reducer*, we get eight different combined failure scenarios. According to the way we deal with failures in McTAR, we summarize them as three representative scenarios, which are able to cover significant failure situations in a MapReduce job.

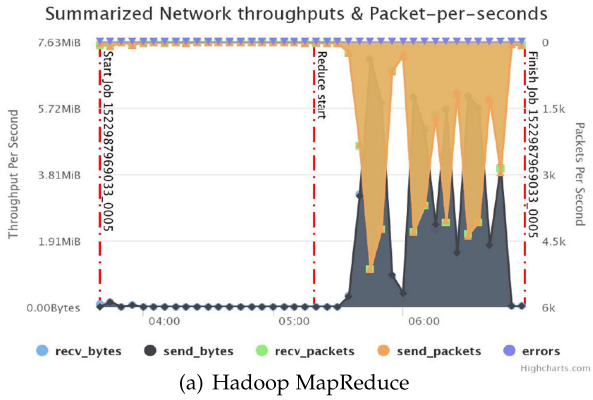
1) *Scenario 1:* Task failure or node failure occur on the *Mapper*, and no failure occur on the *Reducer*. As long as there is no node failure happens on *Reducer*, the intermediate data that have sent to *Reducers* by *proactively-push* obviously remain available, so the *Mapper* could resume working after failure without concerning about the intermediate data generated before the last space-triggered (for task failure) checkpoint or time-triggered (for node failure) checkpoint.
2) *Scenario 2:* Task failure or no failure occur on the *Mapper*, and node failure occur on the *Reducer*. In this case, the *Reducer* has to launch *pull-on-demand* to fetch intermediate data from all relevant *Mappers*, since there will be no local data after node failure on *Reducer*. If there is no node failure happens for all *Mappers*, the *Reducer* gets the intermediate directly.
3) *Scenario 3:* Node failure occur on the *Mapper*, and node failure also occur on the *Reducer*. It is the only scenario that requires for recomputing. The intermediate data on *Mapper* have missed after a node failure, and the node failure makes the specific part of the replication for missing data on the *Reducer* not be accessed when the *Reduce* function begins. The *Mapper* crashed before it has to recompute the lost data when the *Reducer* tries to fetch intermediate data from it.

We inject runtime exception by a given failure rate to trigger task failures and simulate node failures by shut down the VMs forcedly. We choose to run WordCount application in McTAR and Hadoop MapReduce three times with failures injected randomly and take the average value of respective experiments as final results. Two groups of experiments in the three scenarios are included, where jobs execute under different scale of input at 2 percent failure rate and under 20 GB input data at different failure rate.

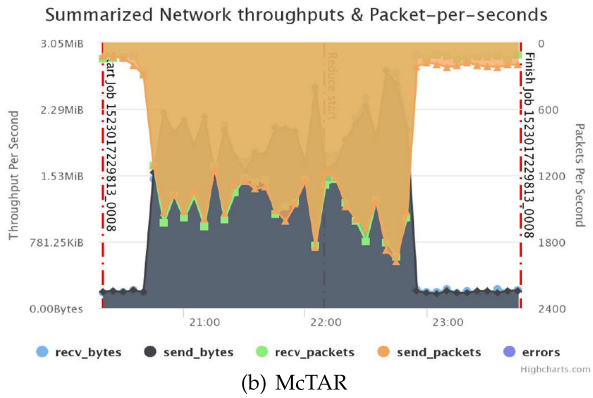
4.2.1 Scenario 1

Fig. 11 shows that the original version of Hadoop MapReduce is seriously influenced by failures. But our McTAR could effectively handle failures even if it takes slightly more time than the original version of Hadoop while no failure happens. With the growth of failure rate to 5 percent, the time duration of job execution in McTAR is nearly half of that in the original Hadoop MapReduce.

Fig. 12 is the job execution duration under different input data scale when we inject failure at 2 percent failure rate. We could see that the job execution durations in both McTAR and original Hadoop are increasing with the data scale increasing. However, our McTAR performs better under all data scale settings, that is, the fault tolerant capability of McTAR is independent of input data size.



(a) Hadoop MapReduce



(b) McTAR

Fig. 10. Comparison of network usage in Hadoop MapReduce and McTAR.

4.2.2 Scenario 2

Compared with above failure scenario, the node failure is involved on *Reducer* in this scenario. Fig. 13 shows that our McTAR could also effectively handle failures under various failure rates, that is, with the growth of failure rate to 5 percent, the time duration of job execution in McTAR is also nearly half of that in the original Hadoop MapReduce. It should be mentioned that our McTAR is mainly focus on the failure management towards the *Mapper*, so when node failure happens on *Reducer*, compared with experiments results in Scenario 1, the extra improvements in reducing the task recovery time are not so obvious.

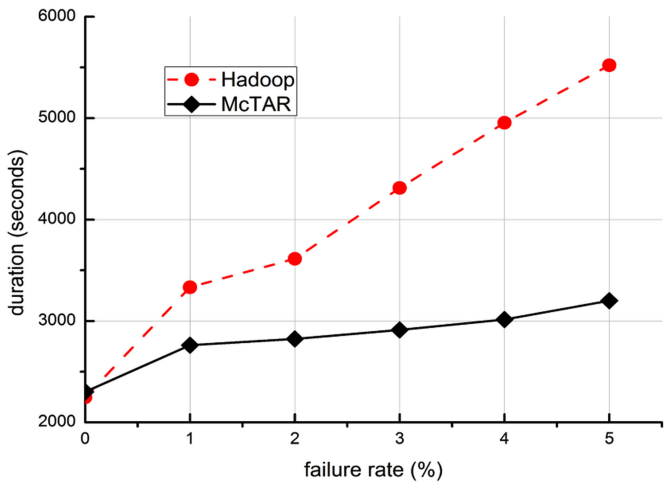


Fig. 11. Job execution duration under various failure rate in Scenario 1.

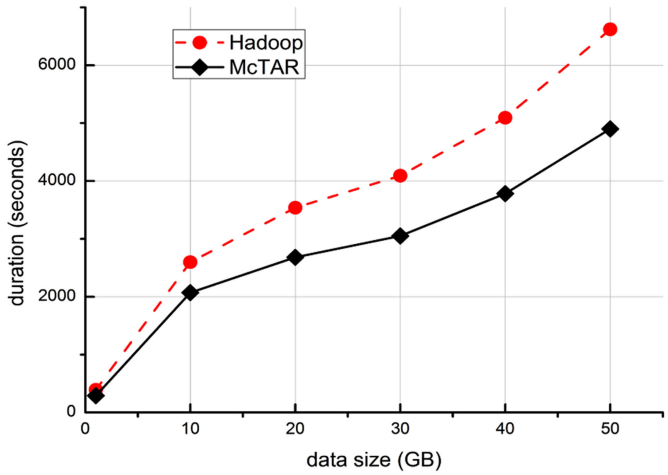


Fig. 12. Job execution duration under various data scale in Scenario 1.

Fig. 14 is also the job execution duration under different input data scale when we inject failure at 2 percent failure rate. We could also find that our McTAR performs better under all data scale settings, that is, the fault tolerant capability of McTAR is independent of input data size when failures occur on both *Mappers* and *Reducers*.

4.2.3 Scenario 3

In this section, we need to analyze in detail the situation where a node failure happens on *Mapper* first and then a node failure happens on *Reducer* later, because the job executions are seriously affected, and the recomputation of data is unavoidable. Fig. 15 shows the compared results in job execution duration under this representative situation. The duration time of job execution obviously increases compared with above two scenarios. However, we still find that our McTAR could also effectively handle failures under various failure rates, that is, with the growth of failure rate to 5 percent, the time duration of job execution in McTAR is nearly 65 percent of that in the original Hadoop MapReduce. Besides, with the help of MIL, McTAR just needs to recompute towards the failed *Reducer* nodes, which also saves certain execution time compared with the original version of Hadoop MapReduce.

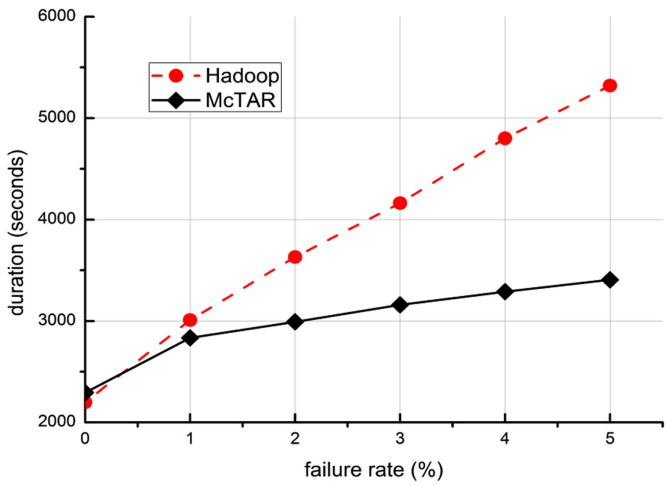


Fig. 13. Job execution duration under various failure rate in Scenario 2.

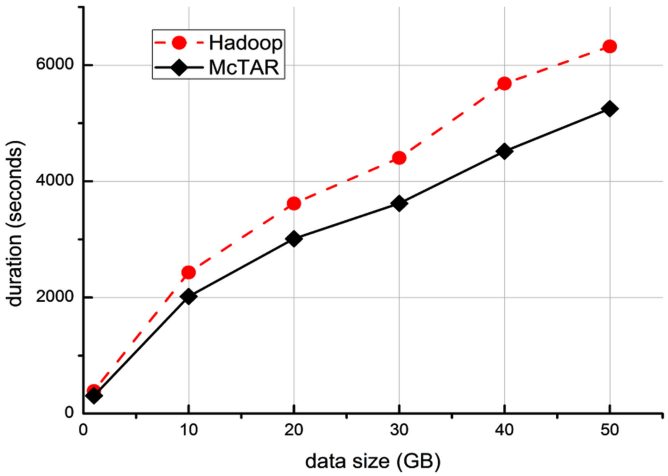


Fig. 14. Job execution duration under various data scale in Scenario 2.

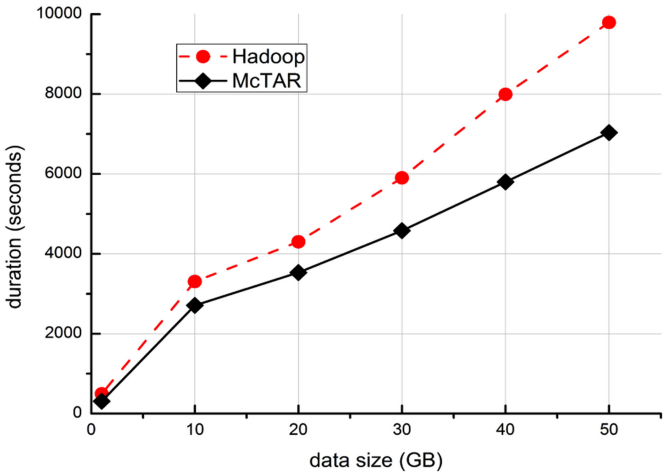


Fig. 16. Job execution duration under various data scale in Scenario 3.

Fig. 16 shows obvious advantages of McTAR as data scale increases in this scenario. Different from the results in Scenario 1 and 2, when the data size becomes bigger and bigger, our McTAR will perform better and better. The reason is that the increase of data scale leads to an increase in *Reduce* tasks, while MIL could play a more effective role for multiple *Reduce* tasks. When there is only one *Reduce* task, even if MIL is enabled, it needs to perform recalculation on all data. Therefore, as the data scale grows, the advantages of McTAR tend to more remarkable.

5 Related Work

Although Hadoop has provided a bunch of strategies to handle failures, a single failure could still cause serious delay of the job according to intensive related study [12].

A great progress was made in RAFT [13], which proposed a family of checkpointing techniques including RAFT-LC, RAFT-RC and RAFTQMC to handle different failures. Our approach is greatly inspired by their work. In their implementation, *Mappers* proactively pushed data to *Reducers* and made the intermediate data replicated as soon as a spill was generated. Since the operation was based on a spill, it could hardly take the advantage of the combiner, which highly increased the network traffic. Apart from that, the checkpoint cannot be made if no spill was generated. To remedy defects mentioned above, we choose to make checkpoint periodically, which is more suitable for various kinds of applications.

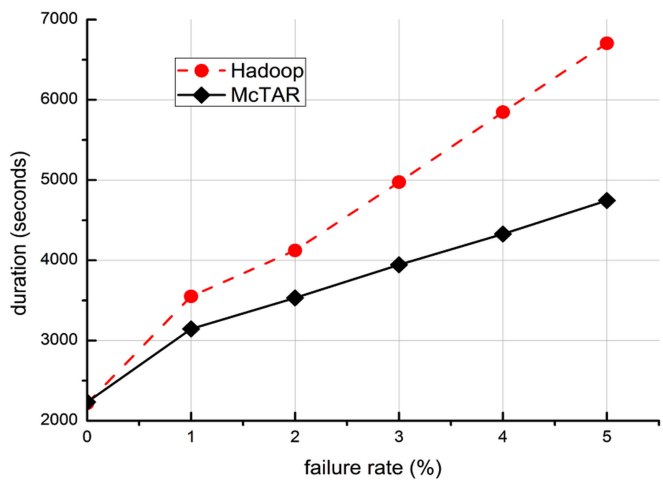


Fig. 15. Job execution duration under various failure rate in Scenario 3.

In [14], an effective MapReduce checkpoint approach named BeTL was presented, which also gives our work much inspiration. It was proposed to send the meta data to a master as checkpoints and come up with a solution to deal with the instability of the generation spills. However, the node failure could not be well handled since the intermediate data were not replicated in their implementation.

A recent study focused on the recovery of *Reduce* tasks using YARN [15]. They analyzed the drawback of several recovery methods and proposed a framework to crack down the failure amplification through analytics logging and migration. In [16], a *Map* task was separated into two parts when the preemption was received. The finished part was submitted, and the rest would be rescheduled as a new task. For the *Reduce* task, the memory was flushed to make a checkpoint when the preemption was received. In their approach, the interruption of tasks may slow down the job. Making checkpoint at a certain progress was proposed in [17], they took advantage of the concept of the LATE algorithm [18] to identify slow task. Another study concerning selecting appropriate checkpointing interval was presented in [19], which took the failure probability and task workload into account to make optimal checkpointing intervals.

Besides, there were also other fault tolerant solutions for Hadoop MapReduce. For example, in [20], [21], replication schemes were well used in job level or task level of MapReduce to tolerate arbitrary faults. But as MapReduce applications generated a great deal of intermediate data, the efforts of replication back-up are needed to be considered seriously. In [22], a distributed and synergetic architecture was developed to separate complex works from original master node to specific modules running on slave nodes to get better performance and fault tolerance capability for Hadoop MapReduce. In [23], an elastic solution using consistent hash was proposed to deal with random fail-stop failures at the node or the network level, which were caused by streamed data to minimize the workload about copying data between *Reducers*. In [24], BFT MapReduce was presented to mask byzantine faults by executing each task more than once to compare whether different outputs appeared or not. It surely has high

cost from re-execution of MapReduce tasks. Docker-Hadoop was proposed in [25], which is a container based Hadoop platform. It could simulate several failure scenarios and used to validate fault tolerant capability of various revised Hadoop MapReduce. Finally, an interesting topic about fixing crash recovery bugs was discussed in [26]. That is, faults in crash recovery approaches in Hadoop tend to cause severe consequences, thus, detailed analysis of their root causes, triggering conditions, and bug impacts would reveal several interesting findings for construct more comprehensive fault tolerance mechanism.

There were also studies that address failure detection problems, such as FARMS [27]. The authors advocated evaluating the stability of a node by recording multiple failures at each node. For nodes with poor stability, speculative execution should start earlier. However, it did not mention the handling of fail-stop faults, and the prediction based on random fault information cannot accurately express the status of nodes.

6 Conclusion

The Hadoop MapReduce paradigm is quite effective and well-adopted for developing big data applications yet. In Hadoop environments, though the coarse grained re-execution fault tolerant strategy works, the MapReduce performance are still suffering from node crash or failed tasks. In this paper, we propose a novel fault tolerant approach, McTAR, to better resolve such dilemma. Our McTAR makes minor revisions to the original MapReduce framework to deals with task or node failures with optimized checkpoint tactics and could achieve better fault tolerant capabilities.

The McTAR consists of several related specific tactics to work together to speed up the recovery of failed MapReduce jobs, including multi-trigger checkpoint generation, push-pull combined intermediate data distribution and optimized failure task prediction. In McTAR, spills are split into small piece groups instead of merging them into one single file. In this way, the intermediate data could be checkpointed and duplicated in time so that the recovery task attempt would start at a specific progress according to the valid checkpoint generated along with spills. It reduces the task recovery delay and improves the performance under failures.

We implement McTAR on the base of Hadoop 2.7.3 and comprehensively evaluate our McTAR with the current Hadoop MapReduce implementation. The experiment results show that McTAR could effectively optimize the recovery process of failed MapReduce jobs and highly reduce the task recovery delay.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (No. 61662051, No. 61662054). A preliminary version of the paper was presented on the 2017 IEEE 1st International Conference on Edge Computing.

References

[1] R. Jhawar and V. Piuri, "Fault tolerance and resilience in cloud computing environments," in *Computer and Information Security Handbook*, 3rd ed. Amsterdam, The Netherlands: Elsevier, 2017, pp. 165–181.

[2] Y. Sharma, B. Javadi, W. Si, and D. Sun, "Reliability and energy efficiency in cloud computing systems: Survey and taxonomy," *J. Netw. Comput. Appl.*, vol. 74, pp. 66–85, 2016.

[3] A. Sangroya, S. Bouchenak, and D. Serrano, "Experience with benchmarking dependability and performance of MapReduce systems," *Perform. Eval.*, vol. 101, pp. 1–19, 2016.

[4] C. Colman-Meixner, C. Develder, M. Tornatore, and B. Mukherjee, "A survey on resiliency techniques in cloud computing infrastructures and applications," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 3, pp. 2244–2281, Jul.–Sep. 2016.

[5] P. Wang, J. Liu, and K. Ding, "TRCID: Optimized task recovery in MapReduce based on checkpointing intermediate data," in *Proc. 1st IEEE Int. Conf. Edge Comput.*, 2017, pp. 112–119.

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Conf. Symp. Opearting Syst. Des. Implementation*, 2004, pp. 10–10.

[7] R. Li, H. Hu, H. Li, Y. Wu, and J. Yang, "Mapreduce parallel programming model: A state-of-the-art survey," *Int. J. Parallel Program.*, vol. 44, no. 4, pp. 832–866, 2016.

[8] Apache, "Apache hadoop 2.7.3," 2016. [Online]. Available: https://hadoop.apache.org/docs/r2.7.3/

[9] S. Shaikh and D. Vora, "YARN versus MapReduce - A comparative study," in *Proc. Int. Conf. Comput. Sustainable Global Develop.*, 2016, pp. 1294–1297.

[10] J. Liu, J. Zhou, and R. Buyya, "Software rejuvenation based fault tolerance scheme for cloud applications," in *Proc. 8th IEEE Int. Conf. Cloud Comput.*, 2015, pp. 1115–1118.

[11] N. Xiong, A. V. Vasilakos, J. Wu, Y. R. Yang, A. Rindos, Y. Zhou, W.-Z. Song, and Y. Pan, "A self-tuning failure detection scheme for cloud computing service," in *Proc. 26th IEEE Int. Parallel Distrib. Process. Symp.*, 2012, pp. 668–679.

[12] Q. Zheng, "Improving MapReduce fault tolerance in the cloud," in *Proc. 24th IEEE Int. Symp. Parallel Distrib. Process. Workshops PhD Forum*, 2010, pp. 1–6.

[13] J.-A. Quiane-Ruiz, C. Pinkel, J. Schad, and J. Dittrich, "RAFTing MapReduce: Fast recovery on the RAFT," in *Proc. 27th IEEE Int. Conf. Data Eng.*, 2011, pp. 589–600.

[14] H. Wang, H. Chen, Z. Du, and F. Hu, "BeTL: MapReduce checkpoint tactics beneath the task level," *IEEE Trans. Services Comput.*, vol. 9, no. 1, pp. 84–95, Jan./Feb. 2016.

[15] Y. Wang, H. Fu, and W. Yu, "Cracking down MapReduce failure amplification through analytics logging and migration," in *Proc. 29th IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 261–270.

[16] O. Yildiz, S. Ibrahim, T. A. Phuong, and G. Antoniu, "Chronos: Failure-aware scheduling in shared hadoop clusters," in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 313–318.

[17] C.-Y. Lin, T.-H. Chen, and Y.-N. Cheng, "On improving fault tolerance for heterogeneous hadoop MapReduce clusters," in *Proc. Int. Conf. Cloud Comput. Big Data*, 2013, pp. 38–43.

[18] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 29–42.

[19] S. Yassir, Z. Mostapha, and K. Najlae, "The impact of checkpointing interval selection on the scheduling performance of hadoop framework," in *Proc. 6th Int. Conf. Multimedia Comput. Syst.*, 2018, pp. 1–6.

[20] P. A. R. S. Costa, X. Bai, F. M. V. Ramos, and M. Correia, "Medusa: An efficient cloud fault-tolerant MapReduce," in *Proc. 16th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2016, pp. 443–452.

[21] P. A. R. S. Costa, F. M. V. Ramos, and M. Correia, "Chrysaor: Fine-grained, fault-tolerant cloud-of-clouds MapReduce," in *Proc. 17th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2017, pp. 421–430.

[22] T.-C. Huang, K.-C. Chu, G.-H. Huang, Y.-C. Shen, and C.-K. Shieh, "Distributed control framework for MapReduce cloud on cloud computing," in *Proc. IEEE/IFIP Netw. Operations Manage. Symp.*, 2018, pp. 1–4.

[23] A. Kumbhare, M. Frincu, Y. L. Simmhan, and V. K. Prasanna, "Fault-tolerant and elastic streaming MapReduce with decentralized coordination," in *Proc. 35th IEEE Int. Conf. Distrib. Comput. Syst.*, 2015, pp. 328–338.

[24] P. Costa, M. Pasin, A. N. Bessani, and M. P. Correia, "On the performance of Byzantine fault-tolerant MapReduce," *IEEE Trans. Depend. Secure Comput.*, vol. 10, no. 5, pp. 301–313, Sep./Oct. 2013.

[25] J. Rey, M. Cogorno, S. Nesmachnow, and L. A. Steffenel, "Efficient prototyping of fault tolerant Map-Reduce applications with Docker-Hadoop," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2015, pp. 369–376.

[26] Y. Gao, W. Dou, F. Qin, C. Gao, D. Wang, J. Wei, R. Huang, L. Zhou, and Y. Wu, "An empirical study on crash recovery bugs in large-scale distributed systems," in *Proc. 26th ACM Joint Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 539–550.
[27] H. Fu, H. Chen, Y. Zhu, and W. Yu, "FARMS: Efficient MapReduce speculation for failure recovery in short jobs," *Parallel Comput.*, vol. 61, pp. 68–82, 2017.

Jing Liu received the PhD degree in computer architecture from the Institute of Computing Technology, Chinese Academy of Sciences, China, in 2011. He was a visiting scholar with the University of Melbourne, from 2014 to 2015. He is currently an associate professor of computer science and technology with Inner Mongolia University. His research interests include software fault tolerance, cloud computing, and formal method. He has published more than 20 papers in international conferences and journals. He is a member of the IEEE.

Peng Wang received the master's degree in computer architecture from the Inner Mongolia University, China, in 2018. He is currently working as a software engineer with Alibaba Group. His research interests include software fault tolerance and cloud computing.

Jiantao Zhou received the PhD degree in computer science from the Tsinghua University, China, in 2005. She was a visiting scholar with the De Montfort University, from 2009 to 2010. She is currently a full professor of computer science and technology with Inner Mongolia University. Her research interests include cloud computing, software engineering, and formal method. She has published more than 90 papers in international conferences and journals, such as the *IEEE Transactions on Computers*, the *Journal of Supercomputing*, ICWS, SCC, and COMPSAC. She is a member of the IEEE.

Keqin Li is a SUNY distinguished professor of computer science with the State University of New York. He is also a distinguished professor with Hunan University, China. His current research interests include cloud computing, fog computing and mobile edge computing, energy efficient computing and communication, embedded systems and cyber-physical systems, heterogeneous computing systems, big data computing, high-performance computing, CPU-GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, intelligent, and soft computing. He has published more than 630 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He currently serves or has served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *IEEE Transactions on Cloud Computing*, the *IEEE Transactions on Services Computing*, and the *IEEE Transactions on Sustainable Computing*. He is a fellow of the IEEE.