RCMP: A System Enabling Efficient Re-computation Based Failure Resilience for Big Data Analytics

Florin Dinu T. S. Eugene Ng
Rice University
Technical Report TR13-04

ABSTRACT

Multi-job I/O-intensive big-data computations can suffer a significant performance hit due to relying on data replication as the main failure resilience strategy. Data replication is inherently an expensive operation for I/O-intensive jobs because the datasets to be replicated are very large. Moreover, since the failure resilience guarantees provided by replication are fundamentally limited by the number of available replicas, jobs may fail when all replicas are lost.

In this paper we argue that job re-computation should also be a first-order failure resilience strategy for big data analytics. Re-computation support is especially important for multi-job computations because they can require cascading re-computations to deal with the data loss caused by failures. We propose RCMP, a system that performs efficient job re-computation. RCMP improves on state-of-the-art big data processing systems which rely on data replication and consequently lack any dedicated support for re-computation. RCMP can speed-up a job's re-computation by leveraging outputs that it stored during that job's successful run. During re-computation, RCMP can efficiently utilize the available compute node parallelism by switching to a finer-grained task scheduling granularity. Furthermore, RCMP can mitigate hot-spots specific to re-computation runs. Our experiments on a moderate-sized cluster show that compared to using replication, RCMP can provide significant benefits during failure-free periods while still finishing multi-job computations in comparable or better time when impacted by single and double data loss events.

1. INTRODUCTION

Computations executed as a series of I/O-intensive jobs are common in big data analytics. In these multi-job computations, soon after a job finishes, its output becomes the input of a subsequent job. Such multi-job computations are popular because the primitives provided by big data processing systems (e.g. Hadoop [2], MapReduce [16]) constrain the amount of work possible in a job. As a result, users need to divide their algorithms into multiple jobs [24, 34, 23] or rely on higher level languages (e.g. Hive [31, 32] or Pig [27]) which usually also get compiled into sequences of jobs. Moreover, the division in multiple jobs facilitates the use of simple, general, computation-agnostic check-pointing mechanisms based solely on handling the outputs of intermediate jobs [14]. We are aware of one computation requiring as much as 150 jobs to complete [1]. In the future, we expect the number of multi-job I/O-intensive computations to significantly increase as more and more applications ranging from scientific research to health care (e.g. DNA sequencing) or risk and resource management reap the benefits of processing increasingly large datasets.

Good performance, despite dealing with the occasional failures, is important for users running multi-job I/O-intensive computations. Unfortunately, this goal is tough to achieve today because data replication is the primary failure-resilience strategy employed by big data processing systems. Data replication is essentially an expensive operation in the context of big data analytics because the large data transfers that it requires put significant stress on the network and the storage. Today's clusters are especially inefficient at handling large transfers due to economical constraints and architectural bottlenecks (e.g. oversubscribed networks [10], poor disk throughput [29, 30]). Our evaluation shows that in the absence of failures, a data-intensive multi-job computation can almost double its running time when the replication factor is increased from 1 to 3. The same multi-job computation took 40% more time to complete when a replication factor of 2 was used instead of 1. Importantly, the large performance penalty induced by replication is paid on *every* use of replication, even during failure-free periods. To cut down the overhead, one may choose to avoid replication and single-replicate[1] the output of each job. However, in this case, when failures do occur, they are likely to have a significant detrimental impact on the performance of multi-job computations. In a single-replicated system, failures can easily cause data loss which can trigger cascading re-computations: several jobs will need to be re-computed for the lost data to be re-generated. In the worst case, the re-computation may have to revert all the way to the beginning. Unfortunately, today's big data processing systems have no built-in support to handle cascading re-computations efficiently. All affected jobs are re-submitted and the system treats the re-submissions identically to the initial runs: it computes the jobs entirely.

In this paper we argue that job re-computation should be a first order failure resilience strategy for multi-job I/O-intensive computations. If done right, re-computation can be efficient when failures do occur while bearing no cost during failure-free periods. Moreover, re-computation support is necessary because of the fundamental limitation of replication. Replication can only tolerate the

This research was sponsored by the NSF under CNS-1018807 and CNS-1162270, by an Alfred P. Sloan Research Fellowship, an IBM Faculty Award, and by Microsoft Corp. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the ofïňAcial policies, either expressed or implied, of NSF, the Alfred P. Sloan Foundation, IBM Corp., Microsoft Corp., or the U.S. government.

[1]Single-replicate means writing only one copy of the data. In contrast, a triple-replicated system always writes three copies of the data.

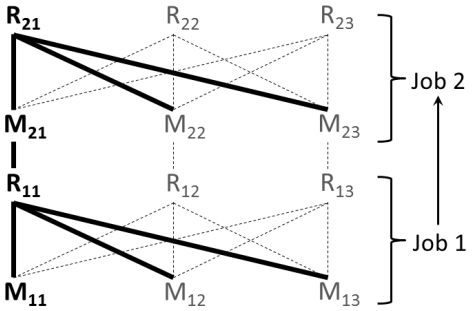

Figure 1: The set of tasks and data transfers (both in bold) that are part of the re-computation of a MapReduce job under RCMP. M = mapper task, R=reducer task. A failure occurs just before Job 2 completes. The outputs of tasks M_{11},R_{11},M_{21},R_{21} are lost due to the failure and need to be re-computed. In contrast, existing systems re-compute everything.

failures that do not affect all replicas in the system but the number of replicas is usually limited for performance and resource utilization reasons. In contrast, re-computation can recover a job from any number of failures as long as the input to the multi-job computation remains accessible. Nevertheless, we view re-computation not as a replacement for replication but rather as a complement. Our position is that enabling efficient re-computation will in turn enable judicious use of replication thus facilitating improvements in overall computation performance. For example, replication can be used to bound the cost of re-computation.

This paper makes three contributions towards making re-computation a first class failure resilience strategy for big data computations. The first contribution is the design of RCMP, a system that performs efficient job re-computations in the context of the popular MapReduce paradigm. RCMP re-computes only the minimum of tasks necessary for each re-computed job. For this, RCMP persists across jobs map outputs as well as reducer outputs that are part of successfully completed intermediate jobs. On failures that cause data loss, RCMP decides which jobs must be re-computed and based on the persisted data it also determines the minimum number of tasks that need to be re-computed for each re-computed job. As an example, consider Figure 1 which illustrates a re-computation performed by RCMP. The failure occurs just before Job 2 finishes. R_{21} is lost and needs to be re-computed. But R_{21} requires the output of M_{21} which is also lost. In turn, M_{21} is based on the output of R_{11} which was also on the failed node and was lost. Thus, Job 2 cannot continue before R_{11} is re-generated. Therefore, RCMP has to cascade back to Job 1 to re-generate the input of Job 2. In total, RCMP re-computes only the tasks that had outputs on the failed compute node (M_{11},R_{11},M_{21},R_{21}) as well as the data transfers that are required for these re-computed tasks (bold lines in Figure 1). Note that the re-computation work performed by RCMP is a fraction of the work performed by current systems which re-compute everything.

The second contribution is identifying and tackling two fundamental challenges that limit the efficiency of RCMP's re-computation runs: the difficulty in fully leveraging the available compute-node parallelism and the presence of hot-spots. These two challenges are not limited to RCMP. They will appear for any advanced MapReduce system that single replicates job outputs and is able to perform limited re-computations. The first challenge is that during job re-computation, the re-computed tasks are unlikely to be numerous enough to efficiently utilize the available compute node parallelism. In other words, the task scheduling granularity used during the initial run is insufficient for efficient job re-computation. This results in underutilized compute nodes and consequently inefficient re-computation. The second challenge is that hot-spots appear during the re-computation of a job's mappers. In the initial run of the job these mappers were running in multiple waves on the failed node. This essentially serialized their accesses to the input. During re-computation, these mappers run on multiple nodes in far fewer waves, oftentimes only one. Thus their data accesses are now parallelized, so they concurrently attempt to obtain map input data from the location holding the output of the previously re-computed job. The resulting contention significantly increases mapper running time and consequently the whole job re-computation time. RCMP's approach to these challenges is to switch to a more fine-grained task scheduling granularity during re-computation. This better utilizes the available compute nodes. This also mitigates hot-spots because the work of a re-computed reducer is distributed over many nodes and the mappers in the subsequent job will balance their accesses across all these nodes now holding the reducer output.

The third contribution is that with experiments on a moderate-sized research cluster, we quantitatively describe the magnitude of the overheads that data replication can introduce as well as the benefits of efficient re-computation. We implemented RCMP on top of Hadoop. When no failure impacts the system, compared to using replication, RCMP is able to reduce the running time of a multi-job computation by up to half. Furthermore, RCMP yields better or comparable total multi-job running time under single and double data loss events. RCMP can re-compute a single job up to 5x faster compared to a full re-computation of the same job.

Perhaps the closest work related to RCMP is the work on Resilient Distributed Datasets (RDDs) [36]. RDDs log the transformations used to build a dataset and use this lineage information to guide re-computation on failures. RDDs are concerned with determining *what* to re-compute while RCMP focuses both on *what* and *how* to re-compute. RDD is geared towards applications that can fit most of their data in memory. RCMP focuses on the general case where data may not fit in memory and thus needs to be written to stable storage. As a result, unlike RDD, we were able to use RCMP to quantitatively analyze the overhead of replication and compare this with the benefits of re-computation.

The paper is organized as follows. Section §2 makes a number of important clarifications and presents relevant MapReduce background. Section §3 discusses at length the problems and overheads of replication. Section §4 presents the capabilities and design details of RCMP while section §5 discussed additional considerations. Section §6 presents the evaluation. Section §7 discusses related work and section §8 concludes.

2. BACKGROUND AND NOTATION

A MapReduce job is a unit of work that the client wants to be performed. Commonly, multiple job are chained together to form more complex computations. The input and output of a job is composed of key-value pairs and is stored in a distributed file system. To enable efficient processing, jobs are further subdivided into tasks. A MapReduce task is a computation that is performed concurrently and identically by different compute nodes, usually on different pieces of data. There are two types of tasks in MapReduce: mappers and reducers. Mappers run first and process the job's input data. The output of the mappers becomes the input to the reducers and the output of the reducers is the output of the entire job. Mappers process key-value pairs independently while each reducer is responsible for processing a separate key range. The mappers and

reducers are linked in a computation DAG. The exchange of data between the two types of tasks is called the shuffle phase. During the shuffle phase, reducers copy from mappers the key-value pairs that correspond to the keys they need to process. In practice, each reducer usually ends up getting a piece of each of the mapper's output data. This results in an all-to-all traffic pattern between the nodes running the tasks.

We refer to the first execution of a job as the *initial run* of that job. During failure recovery, parts of the job may have to be re-executed. We call such a re-execution a *re-computation run* of that job. It is important to clarify how the notion of re-computation applies to MapReduce. This paper is about job-level re-computation. Such re-computation is necessary when the currently running job cannot continue to completion because part of its input data became unavailable. Its input needs to be re-generated and this is done by re-computing parts of a number of previous jobs. This should not be confused with the speculative execution mechanism in MapReduce. Speculative execution is a task-level mechanism. Its purpose is to detect tasks that are slow and duplicate or restart them on other nodes in the hope that in the new location they will progress faster. Speculative execution is useful only when the input to the job is available.

In this paper, when we use the term replication we refer to data replication, that is, writing multiple copies of the same data on multiple nodes. The name of our system, RCMP, is not an acronym but rather a name derived from the word re-computation.

3. WHY REPLICATION IS PROBLEMATIC

In this section we provide detailed arguments in support of our claim that replication is too costly to be the only failure resilience strategy used in big-data analytics. Our position is that despite replication's failure resilience guarantees and the performance benefits it offers in a few narrow cases, there are simply too many practically relevant situations in which the costs of replication significantly outweigh the benefits, thus suggesting the need for devising supplementary failure resilience strategies.

Part of the overhead of replication stems from inefficiencies in the design and implementation of current systems. For example, in their most common and affordable design, data center networks are often oversubscribed [10]. Moreover, the disk throughput obtained by applications frequently falls well short of the full capabilities that the disk hardware can deliver [30, 28]. A number of solutions have been proposed to improve I/O performance. These solutions could also decrease the cost of replication. Some proposals advocate batching optimizations designed to mitigate the detrimental effect of excessive seeks caused by concurrent disk accesses [30, 28]. Others note that a major culprit is the layering of the distributed file systems on top of general purpose file systems which are not optimized for big data workloads [30]. Solutions leveraging raw access to disk are needed to overcome this limitation [25]. While these solutions are able to incrementally improve the performance, the fundamental limitation remains. Replication adds extra I/O workload into the system. Therefore, the relative overhead of replication is expected to persist despite the proposed optimizations.

3.1 Overrated benefits of replication

Failures are not an ubiquitous threat Replication does provides useful failure resilience guarantees. Current replication strategies [18, 2] are able to protect against the simultaneous failure of two nodes or an entire rack of compute servers. This is particularly useful when a job has a high probability of encountering a failure. One example are large-scale, long-running jobs spanning thousands of nodes in a cluster.

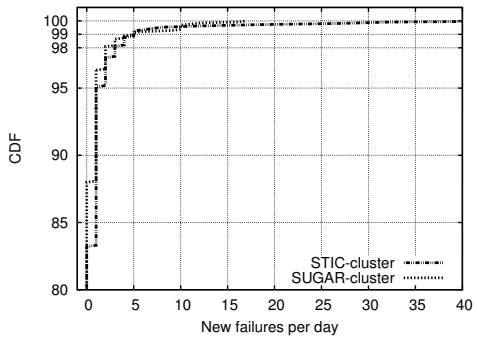

Figure 2: CDF of number of new failures reported per day for the STIC and SUG@R clusters at Rice University

However, most data analytics users do not routinely run such large scale jobs. Only a handful of Internet companies have extremely large clusters. Cloudera reports that the median size of a cluster used for data analytics is under 30 nodes [3], while the average is around 200. At this moderate scale node failures are expected only at an interval of days [28].

Figure 2 depicts the rate at which compute nodes become unavailable for the STIC [4] and SUG@R [5] clusters at Rice University. STIC has 218 nodes and SUG@R has 121. The traces capture the period Sep 2009 - Sep 2012 for STIC and Jan 2009 - Sep 2012 for SUG@R. The statistics are based on the output of an automated script which checks the status of all compute nodes and reports unavailable nodes once per day. Less than 5% of the days had no reports and therefore we discarded them. Note that only 12% of days for SUG@R and 17% of days for STIC show new failures. Discussions with the IT staff revealed that most failure events reported in Figure 2 are likely hardware issues that take at least a day to solve. The few days with many nodes becoming unavailable are unplanned situations (scheduler and file system outages or performance degradation). Our numbers corroborate with estimates from other studies [28] and suggest that for moderate-sized clusters occasional failures should be expected but are not an ubiquitous threat. Therefore, in these situations continuous use of replication for failure resilience is unwarranted.

Data locality is oftentimes inconsequential In clusters where computation and storage are collocated, replication can improve the chance of scheduling data local tasks. More replicas result in better chances that a node having a replica of a task's input data will be selected by the scheduler to run the task. The increase in disk locality can translate into improvements in job running time when it is significantly more efficient to process data locally. One example is a cluster with a highly oversubscribed network.

Note that when computation and storage are separated, data locality is not even applicable [25]. There are also many situations in which data locality is inconsequential. The best example are environments where the network is not the bottleneck. Such systems are proposed often today even for the large scale [20, 6, 26] and have long been economically viable at moderate and small scale. Future trends point to advances in networking technologies that will outpace corresponding advancements in disk drive technologies thus eventually making disk locality completely irrelevant [7].

Note also that in many situations, data locality is easily achievable without replication. The best example are data analytics computations performed on moderate-sized clusters that use the collocated design. In this case, data locality is trivially obtained by

distributing data evenly across the same set of nodes that perform computations. Thus, each node will have plenty of local data to compute on and little or no remote access is required. Moreover, improving data locality can also be done with smart scheduling decisions [35]. Even if all of the above examples do not apply, the benefits of data locality may not necessarily offset the overhead of replication.

Speculative execution Replication may also improve the chance of performing successful speculative execution of mappers. The idea is that if a slow mapper needs to be duplicated (or restarted), the duplicate can read the input from another location thus potentially bypassing the problem that caused the initial task to be slow.

Note that this improvement applies only to the case when the slowness is caused by inefficiencies in reading input data (bad drives, slow network transfers, overloaded compute nodes). If the cause of the slowness strictly affects the computation, then speculative execution will succeed even in a single-replicated system. Moreover, the overall benefits of speculative execution should not be overestimated. Studies show that as many as 90% of speculatively executed tasks do not help improve computation time [11]. The reason is that most speculative execution algorithms are unable to collect enough information to understand the causes of stragglers and end up making suboptimal decisions as a result [8].

3.2 Indirect costs of replication

Increasing job running time is an obvious danger of using replication. However, replication also has several less obvious disadvantages. First, replication in one job indirectly affects other concurrently running jobs by increasing the contention on disk and network resources. Second, replication increases the costs necessary for provisioning a cluster that can sustain a given job execution rate because extra compute nodes or disks are necessary to compensate for the overhead of replication. Third, replication makes a system that collocates storage and computation harder to scale. Future projections show that the number of cores in a commodity compute node will increase significantly but this trend will not be matched by a similar increase in the throughput of commodity drives [7]. The only way to increase local I/O throughput will continue to be increasing the number of disk drives. Today system designers are already pushing the limits of cooling solutions and chassis design as they are forced to fit as many as 16 or 24 disks per compute node [25]. The extra overhead introduced by replication further aggravates the trend.

4. IMPROVING RE-COMPUTATIONS WITH RCMP

In this section we detail the design and capabilities of RCMP. To provide a theoretical quantification of the magnitude of the challenges and of RCMP's benefits this section uses a simple model of the environment and of the MapReduce paradigm. We make the following notations and assumptions. There are N compute nodes and each node writes to D separate storage locations (local disks or remote storage nodes). Each node is provisioned with S map slots and S reduce slots. Each node runs W_M waves of mappers and W_R waves of reducers. For simplicity consider that each compute node runs the same number of tasks and each task performs the same amount of work. We make these assumptions for illustration purposes only. RCMP does not need them.

4.1 System design

We now present the design of RCMP using Figure 3 as the illustration. RCMP builds on Hadoop's design. Where applicable, we

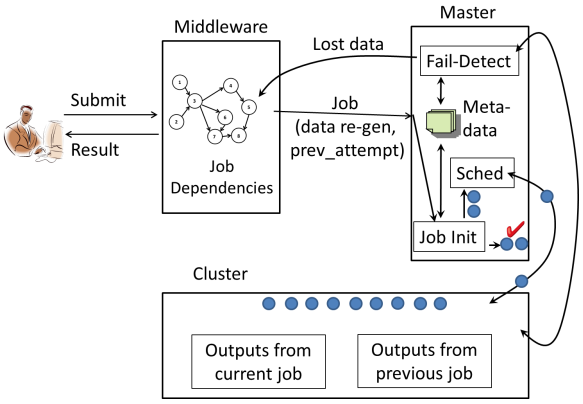

Figure 3: RCMP system overview

emphasize the additional features that RCMP introduces compared to Hadoop.

The user submits the multi-job computation and describes the job dependencies. A middleware program uses the dependencies to decide the order of job submission. For the middleware, we built upon the primitive implementation offered by Hadoop. A job is submitted only after the other jobs that it depends upon have been successfully computed. The jobs are submitted to the Master one by one. The Master possesses no knowledge of job dependencies and knows only how to run individual jobs to completion.

Upon receiving a regular job (not re-computation), a job initialization component (JobInit) in the Master creates the tasks (circles in Figure 3) that need to be executed and the scheduler assigns them to cluster nodes. RCMP recognizes that during the computation of a job a significant amount of data (intermediate map outputs as well as reducer outputs) needs to be materialized anyway for the job to complete. RCMP persists this data across jobs to benefit potential future re-computation, effectively trading-off storage space for re-computation speed-up. In the common case, component failures are likely to lead to the loss of only a small portion of a job's persisted data. Therefore, most of the data persisted on an initial run can be re-used to minimize the work that would need to be performed in the event of re-computation.

The Master periodically checks the cluster's health and detects failures. In RCMP, if those failures cause data loss (all replicas of some partitions are lost), then the Master informs the middleware which files (job outputs) were affected and which partitions were lost. The middleware then immediately cancels the currently running job since it cannot complete without the lost data. The middleware uses the job dependency information and the affected files to infer which jobs need to be re-computed and in which order so that the lost data is re-generated. When submitting a re-computation job the middleware tags it with the data partitions that need to be re-generated and the job IDs of any previous successful attempts to compute this job.

Upon receiving a re-computation job, JobInit uses the tagged information to decide which of the persisted data to consider. JobInit checks the metadata on the list of already persisted map outputs and readies for execution only minimum necessary number of mappers: the ones for which the outputs are missing. The rest of the mappers are treated as if they had already finished. Concerning reducers, JobInit readies for execution only the reducers for which partitions were lost. Note that on re-computation, RCMP departs from current MapReduce systems. These systems do not decide which lost data actually needs to be re-generated because they do not under-

stand the notion of a job that needs to be re-computed. They treat each job submitted to the system as a brand new job and re-execute it entirely.

It is possible that a second failure occurs while RCMP is still re-generating data lost because of the first failure. RCMP treats this second nested failure like any other failure. It interrupts the currently running job and starts re-computation. This is because RCMP does not need to recover from each failure separately. A re-computation job can be started to service any number of data loss events. RCMP only needs to be careful and tag the submitted re-computation job with the partitions lost on all failures.

To give a sense of the benefits that RCMP can provide during re-computation when re-using persisted data, consider a system where storage and computation are collocated. After a single node failure, RCMP only needs to compute 1/N of the mappers and 1/N of the reducers. This also translates into a 1/N decrease in the shuffle traffic compared to the initial run of the job. If these 1/N mappers took W_M waves in the initial run, they can now potentially be re-computed in $ceil((W_M * S)/((N-1)*S)) = ceil(W_M/(N-1))$ waves if they can be distributed over all compute nodes. The same arguments applies for the W_R waves of reducers in the initial run.

Ideally, re-computed mappers would only do a small portion of the initial work (1/N for a balanced computation), strictly the amount necessary for the 1/N re-computed reducers. However, it is difficult to make mappers skip specific input records because the reducer destination of each map output record is only decided after the map function is applied to the record. RCMP can do the next best thing and disregard all re-computed records that do not match re-computed reducers. This reduces the I/O necessary to materialize the map outputs to 1/N.

4.2 RCMP during re-computation

A large part of the re-computation speed-up provided by RCMP comes not from re-using persisted data but rather from the efficient manner in which it executes re-computation jobs.

4.2.1 Maximizing resource use for re-computation

Ideally, all available computing resources will be leveraged for re-computations. Ensuring this however, is not a concern for current systems. Today, since job re-computations are simply entire re-runs of a job, if the initial job was provisioned correctly to maximize resource use, the same will likely hold for its re-computation. RCMP needs to do better. Because RCMP may end up re-computing a fraction of a job's tasks there is a real danger that these tasks may be too few to fully utilize all available computing resources. The root of the problem lies in the granularity at which tasks are scheduled. A coarser granularity may be sufficient for the initial run when many tasks need to be executed, but can severely under-utilize computing resources under re-computation. This has profound implications. The job re-computation time, instead of being bounded by the number of available computing resources ends up being bounded by the impact of the failure, the scheduling algorithm, and the job configuration.

Take for example the case when $W_R * S << N$ (i.e the total number of reducers ran by a node for a job is smaller that the total number of nodes used). This is common for many jobs today because simply using very fine-grained tasks for the initial job is often inefficient. Fine-grained scheduling may lead to increased task start-up and shut-down costs and does not overlap computation and I/O well. For example, the suggested best practice is to set the number of reducers so that W_R is 1 because this allows the shuffle phase to fully overlap with the map computation [34]. Unfortunately, this approach will severely under-utilize reducer slots during re-computations. Most of the N nodes will re-compute no reducers.

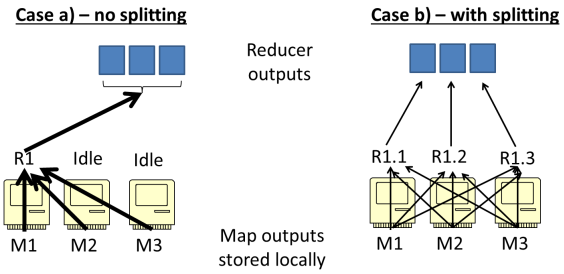

Figure 4: Maximizing resource use for re-computation using task splitting

RCMP can split the tasks that belong to re-computation jobs in order to enable finer-grained scheduling. We implemented this functionality for reducers because mappers are less likely to under-utilize resources since they are usually more numerous and there is no negative side-effect of having $W_M >> 1$. Nevertheless, mappers can be trivially split since oftentimes each record is processed individually. Users should configure RCMP to split reducers only if the application logic allows it. For example, a reducer performing a top-k computation may not be split. Fortunately, for many jobs, reducers can be split. Each reducers is responsible for a number of keys and these keys can simply divided among the multiple splits of the reducer. For simplicity, in our current implementation the split ratio for re-computation is set to use all available node as much as possible.

Figure 4 illustrates reducer splitting in the context of a cluster where computation and storage are not collocated. A storage-side failure caused the loss of reducer R1's output but all map outputs can be re-used because they are hosted on compute nodes. In case a), splitting is not used and 2 compute nodes have idle reducers slots. One node has to re-compute R1 entirely. With splitting (case b), the reducer work is divided among all available nodes and each reducer contributes a portion of R1's output data.

4.2.2 Avoiding hot-spots

Under re-computation there is also the danger of encountering hot-spots when many mappers concurrently converge on one storage location to obtain their input data. This is mostly applicable to the case of a cluster where computation and storage are collocated. For illustration, consider the case in Figure 5. The left side of the figure illustrates the initial run in which a failed node computes 3 mappers in 3 different waves, because it has one mapper slot. These mappers are based on R1's output. During re-computation (right side of the figure), reducer splitting is not used and R1 writes one copy of the data locally on node Y. The 3 mappers based on R1's output are re-computed in 1 single wave because they are distributed over all the nodes. All mappers will attempt to simultaneously access node Y to get their input, thus severely increasing contention.

To further quantify the contention, consider that node Y has D local disk drives. During the map phase of the initial run, the average number of mapper accesses on the D drives is on the order of S, which is the number of mapper slots on a node. Under re-computation, the contention can be as high as $S * N$ which is the number of mapper slots over all available nodes. Furthermore, a network bottleneck may also appear because of the large number of simultaneous transfers.

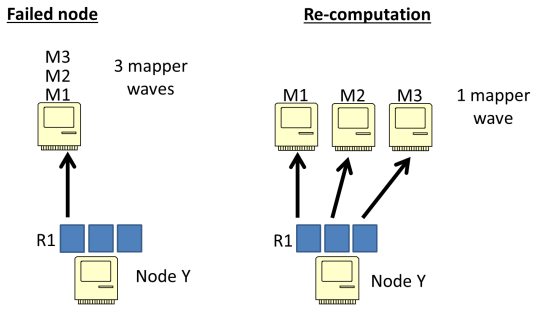

Figure 5: Increase contention on storage during re-computation

RCMP can also use reducer splitting to mitigate the hot-spots. This works because reducer splitting distributes the reducer computation over many available nodes. Thus, this also implicitly distributes reducer output data over the nodes, mitigating the contention in the subsequent map phase. In effect, reducer splitting helps speed-up both the current re-computation job as well as the subsequent one.

We have implemented and tested another solution for mitigating contention. RCMP can instruct the reducers belonging to re-computed jobs to write each output block to a different storage location in order to distribute the output data and mitigate contention in the next job. Compared to reducer splitting this solution does not have the benefit of splitting shuffle work among several nodes. As a result, its capability to lower job running time is more limited. If the shuffle phase is significantly more expensive than the map output computation, then lowering mapper running time does not improve overall job running time because the shuffle will still be the bottleneck. This can occur when a small fraction of the mappers need to be re-computed. In this case map re-computation is fast but the re-computed reducers still need to shuffle data from all mappers, including those for which the map output has been persisted. This solution does have a number of advantages. It can be used even when the application logic does not allow reducer splitting and can always improve cluster utilization by reducing mapper running time.

5. DISCUSSION

Bounding re-computation time with replication Using both replication and re-computation as failure resilience strategies can help ensure that under common failure scenarios, cascading re-computations revert only until the last replication point and not all the way to the beginning of the computation. Replication thus can serve as a way to bound the overhead of re-computation based on the user's risk tolerance. In its current implementation, RCMP does not decide the outputs of which jobs should be replicated. It assumes that the user has the corresponding knowledge and configures jobs accordingly. Alternatively, a simple approach that replicates the output once every constant number of jobs can be used.

A number of additional capabilities need to be built on top of RCMP so that it can efficiently choose replication points. First, RCMP needs to estimate the time required to replicate the output of some job. This can potentially be done using historical information, starting with a rough estimate and improving the accuracy of the estimation with each passing replication. This historical information can also be based on other jobs for which replication is mandatory (e.g. single job computations). Second, RCMP would benefit from historical, cluster-wide failure information in order to extrapolate the most probable failure types. Third, using the failure information, the insight presented in our paper and already available cluster information (e.g. number of nodes), RCMP can estimate the cost of re-computation for each completed job. Fourth, to further optimize the replication decisions, RCMP can benefit from future knowledge about the yet un-computed jobs in the multi-job computation. For example, the expected input to shuffle to output ratio of future jobs would be of great benefit. One way to estimate such information is via static job analysis [19, 22].

Re-claiming the extra storage space used by RCMP With regard to available storage space RCMP behaves similar to Hadoop. If storage space runs out then jobs fail. In environments that are storage space constrained, RCMP needs additional capabilities to manage the storage and re-claim the extra space used. A simple modification is to make RCMP delete the stored data every time a full replication is performed. If this is insufficient, RCMP would have to implement an eviction policy to delete stored data and make room for data required for the job to finish. This eviction policy would be based on the speed-up that stored data would bring to re-computation. For example, it is more efficient to delete stored data at the granularity of waves, since partial waves take roughly the same time to complete as a complete wave. For instance, assume 4 nodes each saving 3 map outputs and having 1 map slot. It makes sense to delete 4, 8 or 12 map outputs at a time. If 6 map outputs are deleted, then the 6 mappers that need to be re-computed would finish in 2 waves, exactly the same as if 8 mappers are re-computed.

6. EVALUATION

6.1 Methodology and environment

RCMP is built on top of Hadoop version 1.0.3, the latest release at the time this project was started. We use the term *Hadoop* to refer to the default implementation. Hadoop uses in all our experiments a replication factor of either 2 or 3. RCMP always uses a replication factor of 1 (only writes a local copy of the job output).

The computing environment To evaluate RCMP we used STIC[4], one of Rice University's compute clusters. STIC is real research cluster, used around the clock by several departments at Rice. The compute nodes that we used are equipped with dual quad-core Intel Xeon CPUs at 2.67Ghz and are interconnected by a 10GbE network. Each compute node has one local S-ATA drive. We only used this local storage for the experiments, thus our experiments cover the case where computation and storage are collocated. Because the nodes have one local drive and a fast network we expect our experiments to be mostly disk-bottlenecked. With the exception of subsection §6.3 all other experiments are run using 10 compute nodes. In all cases, a separate node runs the JobTracker and the NameNode. All compute nodes are non-virtualized and we have exclusive access to them during each experiment.

The multi-job computation used We have built a custom 7-job chain computation. A job starts only after the previous job in the chain successfully finishes. Mappers randomize the key of each record that they receive. Reducers simply output every record received. For the purpose of this paper we believe that the exact computation performed by mappers are reducers is inconsequential. What is important is that the I/O intensive nature of the computation is preserved.

Our job has a ratio of input size to shuffle size to output size of 1 to 1 to 1. This ratio is not uncommon in big data analytics jobs [9, 12, 13] and it is the same ratio used in sorting which is the type of

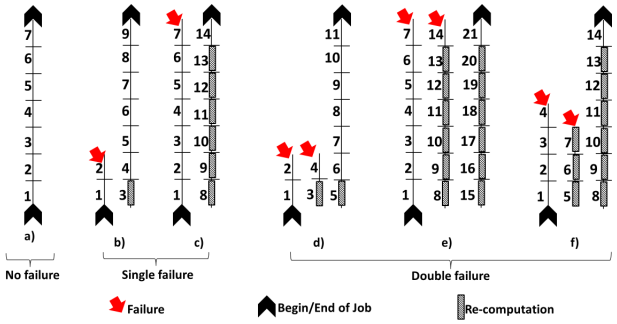

Figure 6: The jobs computed by RCMP for the different moment at which failures are injected in the experiments

job often used as a barometer of cluster performance. The relative benefits of RCMP vs Hadoop are expected to increase when the job output is relatively larger compared to the input and shuffle (i.e. ratios of the form $x : y : z$ where $z > y$ and $z > x$).

The 7-job computation uses as input 40GB of randomly generated input data. This input data is triple replicated. Each of the 10 computes nodes processes roughly 16 mappers, each mapper using as input a block of 256MB. The results that we present are averages based on 5 separate runs of the computation.

How the jobs are numbered Each job that starts running, regardless of whether it is a re-computation or not, receives as an unique ID the next available integer number starting with 1. Re-computations increase the total number of jobs ran. For an illustration consider Figure 6 which depicts the different moments at which we injected failures during the experiments. Consider case c). A failure occurred during the 7th job. As a result, RCMP recomputes the first 6 jobs and then restart the 7th. In this case RCMP started a total of 14 jobs; each of the different 7 jobs was started twice. On the other hand, since Hadoop uses replication to tolerate failures it never has to re-start jobs and always starts a total of 7 jobs.

How failures are injected We inject failures by killing both the TaskTracker and DataNode processes on a randomly chosen compute node. Single failures are injected 15s after the start of the job. If we wish to inject a second failure during the same job, after a further 15s we randomly select an additional node. Both Hadoop and RCMP are configured to trigger failures after a timeout of 30s. Thus, a first failure is detected roughly 45s after the job start. For RCMP, we chose the moments to inject failures as follows. We do not inject failures during the first job since its input is replicated. Case b) in Figure 6 represents a single failure impacting the computation early. Therefore RCMP needs to re-compute just 1 job. In case c), the failure impacts the computation when it is close to completion. RCMP has to re-compute 6 jobs. Case d) shows double failures injected early while in case e) the failures are injected when the multi-job approaches completion. Case f) is an example of a nested failure: the second failure occurs while re-computation is still being performed to address the first failure. For Hadoop we inject failures at jobs 2 or 7.

On the efficiency of our implementation Currently, RCMP has a number of implementation inefficiencies that put it at a slight disadvantage. For instance, for the job during which the failure occurs, RCMP discards the partial results computed before the failure. A more efficient implementation of RCMP would freeze the affected job, perform re-computation as needed to re-generate the lost data and then re-use the partial results after restarting the frozen job. Thus, currently, the roughly 45s necessary for RCMP to react to one failure are pure overhead. In contrast, Hadoop uses replication and can successfully finish the affected job after re-starting the affected tasks. The execution of these re-started tasks may overlap with the computation of other tasks in the job. In the best case, Hadoop may not be at all impacted by the 45s penalty. If we had set the failure detection timeout to more than 30s, or if the failure were injected late in a job (instead of after 15s) then RCMP would be at an even greater disadvantage.

Evaluation road-map We first present overall system comparisons between Hadoop and RCMP in the absence of failures and under single and double compute node failures. We then assess the capability of RCMP to maximize resource use and mitigate hotspots by using reducer splitting. Finally, we isolate the contribution of the reduce phase and mapper phase in the re-computation speed-up provided by RCMP.

6.2 Overall system comparisons

Single failure Figure 7 presents the benefits of RCMP against Hadoop under no failures and single node failures. In the experiment in Figure 7-Left, Hadoop and RCMP use 1 map and 1 reduce slots per node. Each node runs a total of 1 reducer per job in 1 wave. The experiment in Figure 7-Right shows the results when contention is increased by using 2 map and 2 reduce slots per node. In this case each node runs a total of 2 reducers per job, also in 1 wave. For the runs leveraging reducer splitting a ratio of 8 is used.

Increasing the slot count improves overall performance for both RCMP and Hadoop but the relative benefits of RCMP persist. Under no failures, RCMP reduces total job running time by roughly one third compared to Hadooop with a replication factor of 2. Compared to Hadoop with a replication factor of 3, RCMP slashes total running time in half. Even when a failure occurs RCMP remains efficient, slightly edging Hadoop with replication 2. Reducer splitting improves performance every time it is used but more so when the failure is injected during the 7th job because more jobs are recomputed and can benefit from splitting.

Double failures Figure 8 discusses double failures. For this experiment we only consider Hadoop with replication 3 because it can survive any sequence of double failures. In contrast, with a replication factor of 2, Hadoop can fail computing a job when the failures affect both replicas of a block of input data. As it is intuitively expected, Hadoop performs better when the failures are injected late since only a small portion of the computation needs to be executed with 8 nodes instead of 10. On the other hand, it is challenging to assess under which sequence of double failures RCMP is most efficient. If the failures occur late, many jobs need to be re-computed but after the re-computation is finished few job will have to be fully completed with 8 nodes. If the failures occur early, there is little re-computation but many jobs will have to be completed with 8 nodes. The final answer depends on the efficiency of re-computation compared to the overhead of using fewer compute nodes.

Nevertheless, in all the runs RCMP performs well and consistently beats Hadoop when reducer splitting is used. Splitting lowers total running time by 1000s when failures are injected at jobs 7 and 14. Note that RCMP successfully and efficiently handled a nested double failure (the case when failure are injected at jobs 4 and 7) where the second failure occurs while the RCMP is still recovering from the first failure.

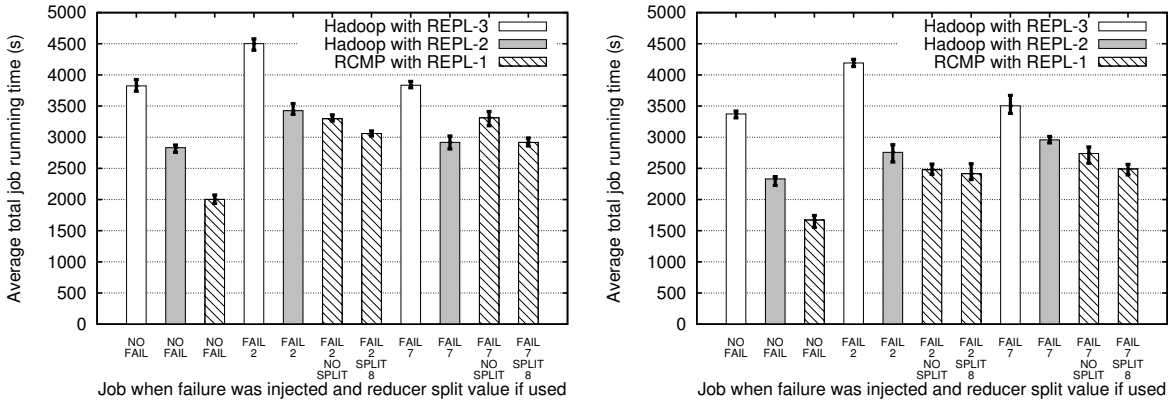

Figure 7: Left: 1 map slot and 1 reduce slot per node. Right: 2 map slots and 2 reduce slots per node. The error-bars show min and max. "NO FAIL" means no failure was injected. "FAIL 2 SPLIT 8" means failure injected at job 2 and RCMP used a reducer split ratio of 8 for re-computation. Under "NO SPLIT", RCMP does not split reducers.

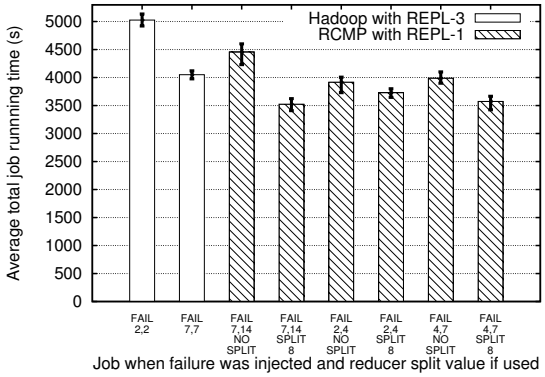

Figure 8: Hadoop vs RCMP under double failures with 1 map slot and 1 reduce slots. See also caption of Figure 7.

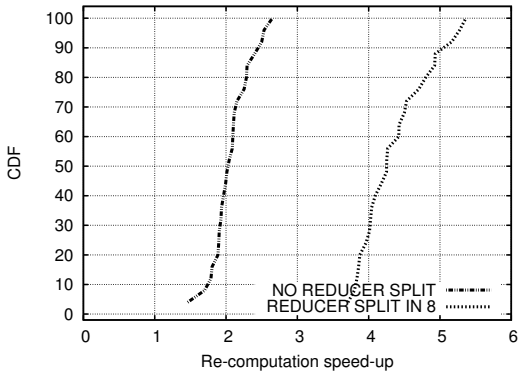

Figure 9: Re-computation speed-up for experiment in Figure 7-Left

Re-computation speed-up Figure 9 shows how fast jobs are re-computed in the experiment in Figure 7-Left compared to the initial run of the jobs. Without reducer splitting, simply by re-using persisted outputs, RCMP yields a 2x speed-up on average. However, with splitting the speed-up more than doubles.

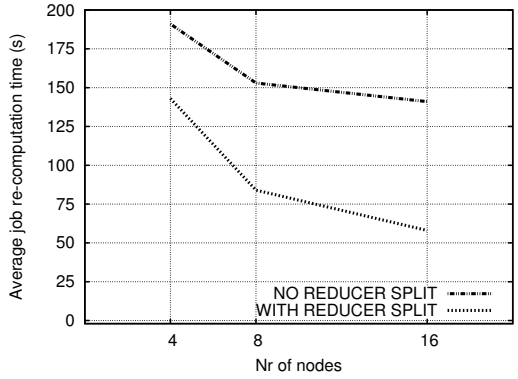

Figure 10: Efficiently using all available resources for re-computation with splitting

6.3 Maximizing resource use

In this experiment we vary the number of compute nodes. However, each compute still processes 4GB of data (16 blocks of 256MB each). When a failure occurs, the 4GB on the failed node need to be re-generated. We want to quantify the benefit that RCMP yields by efficiently using the available compute node parallelism for re-generation.

Figure 10 presents the results. Without reducer splitting, increasing the number of nodes does not provide great benefit in this experiment. The reason is that one compute node needs to fully re-compute the reducer that was on the failed node. The rest of the nodes have idle reducer slots. The benefit seen without reducer splitting stems from the fact that the map computation is divided over more nodes and is completed in fewer waves compared to the initial run. For reducer splitting, a split ratio equal to the initial number of nodes is used. This provides significant benefits. With splitting, re-computation is able to benefit more from an increase in the number of nodes, as each node performs a diminishing amount of reducer work.

6.4 Mitigating hot-spots

Figure 11 shows the negative effects of hot-spots in the runs from Figure 7-Right in which failures are injected at job 7 and RCMP

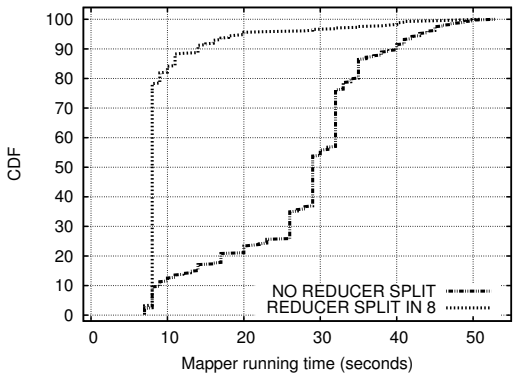

Figure 11: Splitting reducer hot-spots and accelerates mappers

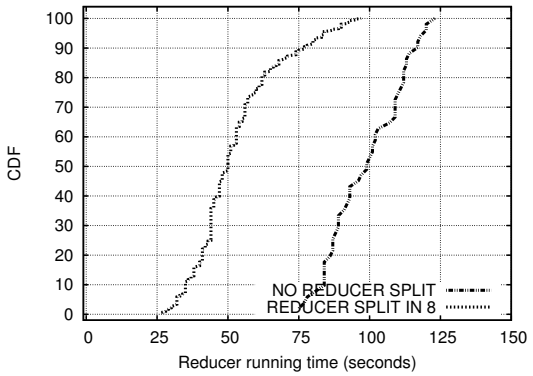

Figure 12: Splitting accelerates reducers

uses 2 map and 2 reduce slots per node. All 9 nodes remaining after the failure are used for re-computation and attempt simultaneously to read the map input from one node. This significantly increases mapper running time. Reducer splitting mitigates contention and in the process also improves reducer running time as seen in Figure 12.

6.5 Benefits of re-computing the minimum number of reducers

In this experiment we seek to understand the contribution of the reducer phase re-computation in the speed-up provided by RCMP. We are primarily interested in isolating the benefit caused by the potential reduction in the number of reducer waves during re-computation compared to the initial stage. Reducer splitting is not used, and to isolate the benefits of the reducer phase re-computation, no map outputs are re-used. All mappers are re-computed.

We vary the number of reducer waves in the initial run (1, 2, 4) by varying the total number of computed reducers (10, 20, 40) and keeping the number of reducer slots to 1. We inject a single failure at job 7. For the re-computed jobs all re-computed reducers (1, 2 or 4) fit in 1 wave. We present two cases, one based on the experiment setting used so far where network speed is fast. In the other case we simulate a very slow network by artificially introducing a 10s delay at the end of each shuffle transfer.

The results are shown in Figure 13. If the network is the bottleneck, the speed-up increases linearly with the decrease in the number of reducer waves re-computed. This is because the map output computation is insignificant compared to the bottlenecked shuffle phase and thus each reducer wave in the initial run takes roughly

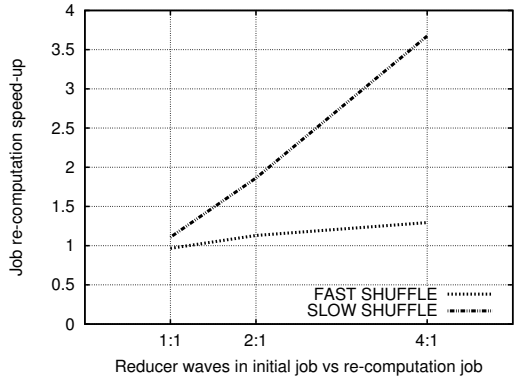

Figure 13: Speed-up obtained from having fewer reducers waves during re-computation

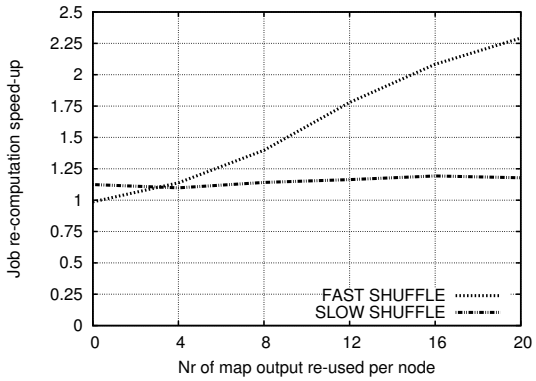

Figure 14: Speed-up obtained from re-using various number of mappers

the same amount of time to complete. In comparison, when the network is fast, the speed-up increases sub-linearly. The reason is that, both for the initial job and for the re-computation, there is substantial overlap between the map computation and the shuffle phase of the first reducer wave. This causes the first reducer wave to be more time-consuming that the rest of the waves.

6.6 Effect of re-computing various number of mappers

In this experiment we isolate the benefits obtained from re-using map outputs. As in the previous experiment we use the standard environment as well as the simulation of a slow network. The x-axis in Figure 14 is the number of map outputs persisted per job per node. The maximum should be roughly 16 since each node is expected to run roughly 16 mappers.

Figure 14 shows that if the network is the bottleneck, then no matter how many map outputs are re-used there is no improvement in speed-up. This is because finishing the map phase faster does not decrease the time necessary to complete the network-bottlenecked shuffle. On the other hand, if the network is very fast then the shuffle finishes shortly after the last map output is computed, resulting in a near-linear increase in speed-up. The speed-up still increases when nodes are allowed to persist more than 16 map outputs per job because there is variability in the number of tasks computed by a node. Some may end up running more than 16 mappers.

7. RELATED WORK

Failure resilience for big-data jobs There is limited work focusing specifically on problems related to failure resilience for big-data jobs. Perhaps the closest to RCMP is the work on Resilient Distributed Datasets (RDDs) [36]. RDDs are a general purpose distributed memory abstraction for sharing data in data center applications. RDDs provide fault-tolerance by logging the transformations used to build a dataset and then using this lineage information to guide re-computation on failures. There are several important differences between RDDs and RCMP. First, RDDs are geared towards applications that can fit most of their data in memory while RCMP focuses on the general case where data may not fit in memory and thus needs to be written to stable storage. Second, the lineage information allows RDDs to determine *what* to re-compute on failure. RCMP also needs to determine *what* to re-compute but goes beyond that and also focuses on *how* to re-compute. That is, RCMP is designed to address specific challenges faced when performing re-computations: maximizing resource use and mitigating hot-spots. RDD does not deal with such challenges. Third, in this paper we quantitatively analyze the overhead of replication and the benefits of re-computation, while the work on RDDs only briefly mentions that a checkpoint-based fault recovery mechanism like replication would be expensive for their experiments. Note that RDD also mentions the term "efficient fault tolerance". However, its meaning is completely different compared to RCMP. RDDs are deemed to provide efficient failure resilience by comparing against solutions that use shared-memory as a distributed memory abstraction which are expensive to provide failure resilience for. In contrast, RCMP provides efficient re-computation based failure resilience by going beyond simple re-computations. RCMP has mechanisms that address *how* to perform re-computation intelligently and efficiently.

FTopt [33] is a cost-based fault-tolerance optimizer that automatically selects the best strategy for each operator in a query plan in a manner that minimizes the expected processing time with failures for the entire query. FTopt focuses on three failure resilience strategies. The first is NONE, when no special action is taken. The second is called MATERIALIZE and is similar to replication. The third is called CHCKPT. It allows a computation to roll-back to a previous state, but the computation would have to re-compute *everything* it has done since check-pointing. FTopt does not provide insights into the benefits and challenges of re-computation.

Re-using previously computed outputs There is also related work on optimizing computations by leveraging outputs previously computed by similar computations. Some big-data computations are amenable to such optimizations because they have similarities in computation (shared sub-computations) and similarities in input (same input or a sliding window of the input data). The challenge that this related work tackles is determining and maximizing the opportunities for data re-use. While RCMP also re-uses previously computed outputs it does not face the same challenges because under failures it need to perform the same computation on the same input. However, RCMP goes beyond data re-use and focuses on *how* to best perform the computations that cannot benefit from re-use. At a higher level, RCMP's focus is also different. RCMP deals with the problem of providing failure resilience for applications while prior work in this area focuses on improvements in performance and storage utilization.

In this space, Nectar [21] is a system that automates and unifies the management of data and computation in data centers. In Nectar, data and computation are treated interchangeably by associating the data with the computation that produced it. Thus, duplicate computations can be avoided by re-using previously cached results. Nectar uses fingerprints of the computation and the input to determine similarity to previous runs. Nectar provides a cache server that allows the lookup of stored entries based on the fingerprints. To provide its functionality Nectar automatically and transparently re-writes programs so that they can cache intermediate results and take advantage of the cached results. ReStore [17] offers capabilities similar to Nectar for workflows of MapReduce jobs.

Improving the I/O efficiency of big-data jobs By providing ways to efficiently deal with failures when they occur, RCMP obviates the need for using expensive I/O to provide proactive failure resilience. Thus, RCMP improves the overall I/O efficiency of big-data jobs. The related work in this area is complementary to RCMP. It improves the I/O efficiency for different phases of a MapReduce job. The existence of these related ideas further amplifies the need for the benefits provided by RCMP. If all the other phases of a MapReduce job are highly optimized, then the relative overhead of using replication as a failure resilience strategy becomes even higher.

Rhea [19] optimizes the map input phase for jobs that selectively use input. Rhea uses static program analysis techniques to detect which data will be accessed by a particular job. Based on this analysis Rhea creates storage-side filters that attempt to discard as early as possible the input records that the computation will not use. ThemisMR [28] optimizes the internal execution of map and reduce tasks. ThemisMR ensures that only the minimum number of I/O operations (a read and a write) are performed for each record. It does this with the help of a custom memory manager that allows it to process records completely once they are read without resorting to swapping or writing spill files. ThemisMR also incorporates I/O optimizations designed to minimize seeks and deliver near-sequential disk I/O to applications. Camdoop [15] optimizes the shuffle phase by performing in-network aggregation.

8. CONCLUSION

We presented RCMP, a system that uses re-computation as a first-order failure resilience strategy for big-data analytics. RCMP is geared at efficiently executing multi-job I/O-intensive MapReduce computations. It leverages previously persisted outputs to speed-up re-computed jobs and during re-computations it ensures that performance bottlenecks are mitigated. During re-computations, RCMP can efficiently utilize the available compute node parallelism and can mitigate hot-spots. Experimentally, we showed that compared to using replicatio, RCMP provides significant benefits when failures are not present, while still finishing multi-job computations comparably or faster under single and double storage data loss events.

9. REFERENCES

[1] Contrail. http://sourceforge.net/apps/mediawiki/contrail-bio/index.php?title=Contrail.

[2] Hadoop. http://hadoop.apache.org/.

[3] Petabyte-scale Hadoop clusters (dozens of them) . http://www.dbms2.com/2011/07/06/petabyte-hadoop-clusters/.

[4] STIC - Shared Tightly Integrated Cluster. http://www.rcsg.rice.edu/stic/.

[5] SUG@R - Shared University Grid at Rice. http://www.rcsg.rice.edu/sugar/.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. SIGCOMM 2008*.

[7] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proc. HotOS 2011*.

[8] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proc. OSDI 2010*.

[9] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Nobody ever got fired for buying a cluster. In *Microsoft Technical Report MSR-TR-2013-2*.

[10] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proc. IMC 2010*.

[11] E. Bortnikov, A. Frank, E. Hillel, and S. Rao. Predicting execution bottlenecks in map-reduce clusters. In *Proc. HotCloud 2012*.

[12] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In *Proc. Eurosys 2012*.

[13] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Proc. MASCOTS 2011*.

[14] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proc. NSDI 2010*.

[15] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: exploiting in-network aggregation for big data applications. In *Proc, NSDI 2012*.

[16] J. Dean and S. Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[17] I. Elghandour and A. Aboulnaga. Restore: Reusing results of mapreduce jobs. In *Proc. VLDB 2012*.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. SOSP 2003*.

[19] C. Gkantsidis, D. Vytiniotis, O. Hodson, D. Narayanan, F. Dinu, and A. Rowstron. Rhea: Automatic filtering for unstructured cloud storage. In *Proc. NSDI 2013*.

[20] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *Proc. SIGCOMM 2009*.

[21] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: automatic management of data and computation in datacenters. In *Proc. OSDI 2010*.

[22] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. In *Proc. VLDB 2011*.

[23] D. Jiang, A. K. H. Tung, and G. Chen. Map-join-reduce: Toward scalable and efficient data analysis on large clusters.

[24] H. Lim, H. Herodotou, and S. Babu. Stubby: a transformation-based optimizer for mapreduce workflows. In *Proc. VLDB 2012*.

[25] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *Proc. OSDI 2012*.

[26] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *Proc. SIGCOMM 2009*.

[27] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. SIGMOD 2008*.

[28] A. Rasmussen, M. Conley, R. Kapoor, V. T. Lam, G. Porter, and A. Vahdat. Themis: An i/o efficient mapreduce. In *Proc. SOCC 2010*.

[29] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. Tritonsort: a balanced large-scale sorting system. In *Proc. NSDI 2011*.

[30] J. Shafer, S. Rixner, and A. L. Cox. The hadoop distributed filesystem: Balancing portability and performance. In *Proc. ISPASS 2010*.

[31] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. In *Proc. VLDB 2009*.

[32] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *Proc. ICDE 2010*.

[33] P. Upadhyaya, Y. Kwon, and M. Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *Proc. SIGMOD 2011*.

[34] T. White. Hadoop - the definitive guide. O'Reilly Media, 3rd edition, 2012.

[35] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proc. EuroSys 2010*.

[36] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI 2012*.