Harvesting Underutilized Resources to Improve Responsiveness and Tolerance to Crash and Silent Faults for Data-intensive Applications

Debashis Ganguly, Mohammad H. Mofrad,
Taieb Znati, Rami Melhem, John R. Lange
Department of Computer Science
University of Pittsburgh
{debashis, hasanzadeh, znati, melhem, jacklange}@cs.pitt.edu

***Abstract*—Low latency is critical for emerging data-intensive real-time analytic and interactive single-wave applications. As the complexity and heterogeneity of cloud computing continue to increase, so will the frequency of errors, which will manifest themselves in unpredictable ways with a significant impact on the correctness of the computing and responsiveness of cloud services. In this paper, we propose a new data-centric computational model to improve the responsiveness of the data-intensive applications to crash faults and augment its ability to deal with silent errors to ensure computational accuracy. The basic tenet of the proposed model is a task replication scheme, which interweaves the processing of a replicated data split among multiple distributed tasks, with each task consuming data at a different offset. In the absence of a failure, the concurrent execution of the tasks ensures complete processing of the data split, with a significant reduction in the total execution time. In the case of an error, however, the remaining tasks take over the execution of the unfinished work and finish on time. The proposed scheme also guarantees timely detection and correction of silent data corruptions along with crash faults. We demonstrate the effectiveness of our scheme by extending Hadoop's MapReduce code base as a case study. Results show a performance improvement of 50% over Hadoop's Speculative Execution when dealing with crash-faults and an improvement of 33% when dealing with silent errors in case of no failure.**

***Keywords*-fault-tolerance, data-intensive computing, replication, crash faults, silent errors;**

I. Introduction

The Apache Hadoop software library [1] is an open-source framework designed to leverage clusters of commodity hardware to support the storage and processing of large scale data sets in a distributed environment using simple programming models [2]. It abstracts away much of the complexity of distributed processing to provide a functionally rich environment capable of handling a massive number of concurrent tasks for structured and unstructured data analysis. In addition to its ability to scale to thousands of servers, Hadoop provides robust fault tolerance mechanisms to ensure high data availability and successful completion of tasks. The basic tenets that drove its design, however, are being challenged by recent advances in technology, such as the Internet-of-things, and the emergence of new time-sensitive applications, which rely on real-time analytic to convert massive amounts of data into actionable insights. Moreover, technology trends, combined with the increasingly high demands for cloud services, is leading to exponential growth, both in complexity and heterogeneity of the computing infrastructure. As a result of the multifold increase in the number of computing components in the data-centers, system failure in the future will be the norm rather than an exception [3].

Low latency is critical for Hadoop to go beyond batch processing and support the execution of real-time analytics, at scale, *correctly* and in a *timely manner* [4]. Faults of any nature can significantly prolong the job completion time. An extensive study done at Google datacenters shows that about a third of machines and over 8% DIMMs faced with at least one correctable error per year and the annual occurrence of uncorrectable error was 1.3% per machine and 0.22% per DIMM [5]. A similar large scale study, conducted at Microsoft over a million consumer PCs, established that faults in CPUs and chip subsystems occur frequently [6]. While the processing engines, like MapReduce, Spark, Flink, Samza, in Hadoop ecosystem can transparently handle *crash faults* by reexecuting tasks on separate nodes, the speculative nature of task reexecution, however, is likely to extend significantly the job's response time. Furthermore *soft errors* are likely to become more prevalent with decrease in feature size along with supply voltages to chips [7]. For mission critical applications, it is very important to guarantee the correctness of computing against *silent data corruption*. To guarantee responsiveness and reliability expected by the future time-sensitive applications at exascale, Hadoop must fundamentally change its approach to failure – from one of focusing on slow task execution to one of embracing higher levels of crash faults and silent data errors.

This paper addresses the above challenges and deals with crash faults and silent failures uniformly. To this end, a novel data-centric fault-tolerance scheme is proposed. The scheme relies on the *interweaved execution of a suite of task replicas processing the same data split, but starting from different offsets*. The scheme relies on *hardware-replication* to achieve the required level of tolerance to failures.

The availability of idle resources, a key enabler of the

2159-6190/17 $31.00 © 2017 IEEE
DOI 10.1109/CLOUD.2017.74

proposed scheme, is widely documented in the literature. The study [8], conducted at Facebook, shows that cluster utilization was less than 50% for 92% of the time, with a CPU and memory utilization below 20%. Moreover, operating cost of datacenters incurred from energy consumption is largely proportional to peak-utilization rather than actual aggregated utilization [9]. This further motivates the use of resource replication to enhance clusters' tolerance to failures [8].

In summary, the work makes following contributions:

- We develop the fault-tolerant scheme to tolerate both crash faults and silent data errors in a uniform manner with minimal overhead.
- We present theoretical model that shows our scheme achieves the target response time of the underlying application, in the presence of failures, and a significant reduction in response time in the absence of any fault.
- We present a prototype implementation of our scheme as an extension to Hadoop's MapReduce processing engine. The code repositories are made available as open source along with configuration files and logs from experimentation [1].
- We evaluate the implemented prototype on Bridges system for three mini-applications to validate the claims made by theoretical models.

The remainder of the paper is organised as follows. Section II reviews works related to the proposed scheme followed by the algorithmic approach and design details in Section III and prototype implementation of the model in Section IV. Section V outlines extensive empirical evaluation of the implemented prototype. Section VI concludes the work proposed in this paper.

II. Literature Review

Approaches used to tolerate failures in large scale systems broadly rely on time or hardware redundancy. Time-redundancy based approaches reexecute backup copies of the same tasks to recover from failures. Speculative execution, used to tolerate failures in Hadoop and MapReduce environments, is a typical example of time redundancy. Experimental studies have shown that in case of heterogeneous virtualized environments speculative execution can degrade performance significantly [2], [10].

Launching backup copy speculatively affects response time adversely. Hence, identifying an outlier as early as possible to launch a speculative copy can significantly improve latency of a job. To this end, researchers have moved towards more aggressive strategies to launch backup replicas of tasks to deal with outliers. In their work, Zaharia et al presented a new scheduling algorithm called Longest Approximate Time to End (LATE) [10]. This scheme speculatively executes the task which the run-time estimates to finish farthest into the future. Their experimental evaluation on 200 heterogeneous virtual machines of Amazon EC2 shows that this greedy approach can improve Hadoop's response time by a factor of 2. Similarly, Ganesh et al present a scheduling algorithm called Mantri, a resource-aware restart scheme that replicates tasks preferentially by analyzing the cause of delay and comparing the opportunity cost of re-execution versus replication [11]. The implementation of this probabilistic scheme on Bing server shows that the algorithm can improve response time by 32%. However, differentiating between a task running on a comparatively slow running node and an actual straggler in a commodity cloud environment is challenging. Ganesh et al took speculative execution to its logical extreme to run full clones of jobs to deal with outliers. Their experiments demonstrated that for small jobs their aggressive cloning strategy called Dolly can improve job completion time by 47% and 39% compared to LATE and Mantri respectively while increasing resource utilization by only 3% [8].

The second approach is hardware redundancy, which rely on the availability of idle resources to simultaneously execute replicas of the same task [12]. This approach has been extensively used for latency-sensitive applications [13], [14].

However, none of the above schemes deals with arbitrary silent errors. Schneider introduced the concept of state machine as a general method for implementing fault-tolerant services in distributed systems, to deal with crash faults and Byzantine faults [15]. Castro and Liskov demonstrated the feasibility of Byzantine fault-tolerance for proactive failure recovery in online services [16]. Following Castro's and Liskov's work, Veronese et al presented two asynchronous Byzantine fault-tolerant state machine replication algorithms [17]. Note that the state machine based approach proposed in [16] requires $3f + 1$ replicas to deal with f Byzantine faults; whereas the two asynchronous algorithms proposed in [17] require only $2f + 1$ task replicas.

Our proposed scheme is motivated by the scheme proposed by Costa et al [18], which combines both time and resource redundancy to enhance MapReduce's tolerance to deal with Byzantine faults. Their scheme executes $f + 1$ replicas in parallel to tolerate f arbitrary faults. Upon completion of these tasks, their scheme compares output signatures to detect silent data errors. If a majority consensus is reached, the computation proceeds with its next phases, until the job successfully completes its execution. Otherwise, the system executes an additional f replicas to recover from the failure. [18] optimizes resource utilization assuming a low error rate. Whereas, our proposed scheme leverages hardware redundancy to guarantee responsiveness of cloud services for both crash faults and silent data error uniformly, and achieves significant reduction in response time in case of no failure.

[1] https://github.com/DebashisGanguly/FTMR

III. Data-interweaved Replication

Based on the criticality of the jobs, consumers and Cloud Service Provider (CSP) negotiate a Service Level Agreement (SLA) which reflects the required level of fault tolerance. Our scheme defines two resiliency parameters namely, *natureOfFault* and *numberOfFaults* to specify the SLA-agreed upon level of fault-tolerance. The parameter *natureOfFault* is used to specify one of two possible fault-tolerant execution modes, *crash faults*, and *silent data-errors*. The parameter *numberOfFaults*, alternatively denoted as *f*, allows the user to specify the "scope" of failure, in terms of the number of faults the system must tolerate while successfully completing the execution of the job. Given the low probability of error in current system, an ideal default value for f is 1. Henceforth, while explaining the details of our scheme we will focus on case studies with $f = 1$. Combined together, the two resiliency parameters capture the critical nature of the job, in relation to the propensity of the system to the type and frequency of failures. Aimed with the user defined parameter values, the scheme uses a simple majority voting mechanism to detect silent failures.

Our scheme leverages the availability of idle resources in cloud clusters to guarantee resiliency and responsiveness by replicating tasks processing the underlying data. Note that to recover from f_c number of consecutive crash faults a system needs f_c+1 replicas. Whereas, to detect f_s number of faulty copies of tasks affected by silent data error, a system needs to compare the results against another f_s replicas of same task and 1 additional replica to recover from. Thus to deal with f_s silent data errors, a system needs $2f_s+1$ replicas. Combining above two cases, in general, we can reach Equation 1 to determine the tasks fault-tolerance replication factor.

$$R = f_c + 2f_s + 1 \tag{1}$$

where:

f_c = Number of crash faults,
f_s = Number of silent data errors,
R = Number of replicas per task needed to withstand above set of faults.

Based on Equation 1, upon receiving a job the system creates R replicas per task. Our scheme also leverages the fact that the input data is replicated for fault tolerance in distributed storage system for batch processing of big dataset. The system runs on the assumption that the replication factor of underlying distributed file system should be R as well. This guarantees that all the replicas can run on distinct data locations in parallel from the beginning.

In our proposed scheme replicas work collaboratively to achieve common goal of processing the replicated split in inter-weaved fashion, but from different offset determined by the logical index of the replica. The life cycle of the replicas consuming the split is divided in multiple logical phases, which is determined by the replication factor R. For example, if the replication factor is 3 and the split consists of 30 records, then replica 1, 2, and 3 will respectively start processing records 1 to 10, 11 to 20, and 21 to 30 in their first phase of execution. After each phase replicas commit their partial output into a stable storage. Then, a hash based signature is generated out of the partial commit and sent to the centralized Voting System. The latter keeps track of the signatures indexed by the replica id and the id of the logical sub-partition to determine nature of fault and the next course of action. The phase, at which voting system determines what action to be performed next, is based on the mode in which the system is configured. The specifics of each mode are described hereafter.

1) Crash Fault mode: In this mode, the system is configured to tolerate crash faults. Based on Equation 1, we can determine the replication factor, R to be $f_c + 1$. Thus, the system initiates $f+1$ replicas per task, where the number of tasks is determined by the number of input splits. A crash fault is identified upon receiving no response from a given replica. In this mode, the health of replicas is monitored after each phase. If after the first phase, all the replicas are alive and have finished their first logical sub-splits, then the Voting System asks the replicas to finish their execution and consolidate all partial outputs.

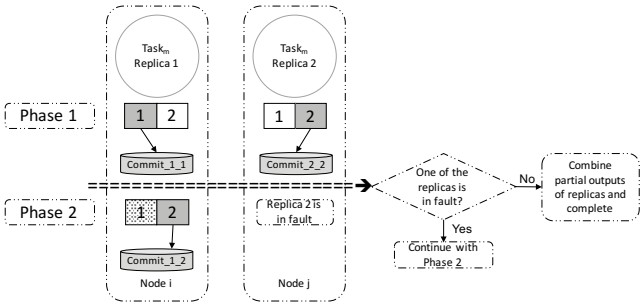


Figure 1: Inter-weaved execution of a replicated data split among two replicas in different phases of their life cycle with the ability of recovering from single crash-fault

For example, the system, described in the Figure 1, can tolerate single crash-fault, i.e., $f = 1$. Thus, the framework initiates 2 replicas per task. Given $R = 2$, the physical data block is logically split into two halves. Replica 1 and 2 start processing logical partitions 1 and 2 respectively in phase 1 of their execution. It is to be noted that no voting is needed to ensure correctness of output in this mode and thus generation and communication of signatures on partial outputs is no longer necessary.

Numerical Analysis of Performance: Assuming that it takes t amount of time to process the whole split by a single task, processing each logical sub-split in each phase by a single replica will take $t/2$ unit of time for a system with replication factor 2. It can be easily generalized that with

$R = f+1$, each phase takes $\frac{1}{f+1}t$ unit of time. In case of no failure, the execution completes at the end of the first phase; thus in case of no failure, all replicas will collaboratively finish processing the input chunk in $\frac{1}{f+1}t$ time or $t/2$ time, as in the example case. This significantly reduces overall job response time. Whereas, in case of failure the response latency execution never exceeds t unit of time and thereby always adhere to the time specified by the SLA.

However, speculative execution model, like one in traditional MapReduce, relies on time redundancy for failure recovery and runs a back-up copy of a task only upon detecting a failure or delay in response. Let us assume that a set of running tasks has failed just before committing the output at the end of their processing. The speculative execution would create and launch back-up tasks for the failed tasks at that time of failure. Hence, in the worst case with f number of crash faults for the same set of tasks repetitively, the response time can be stretched up to ft.

It is worth noted that the traditional MapReduce at any given instance requires only one set of nodes to run the tasks (say n); however, our scheme needs R times the number of resources, i.e., Rn or $(f+1)n$ nodes to run all replicas. If we compare our scheme with more common form of replication without interleaved data processing, full replication also needs $(f+1)n$ resources and always needs t unit of time for both without fault and with fault scenarios. Although our scheme requires same number of resources as traditional full replication scheme; in common case, where there is no failure and the system is provisioned to withstand single crash fault, it reduces the response time by 50%. This is summarized in Table I.

Scheme	Replication Factor	Number of Nodes	Response Time	
			No Failure	Crash Fault
Speculative Execution (SE)	1	n	t	ft
Full Replication (FRFT)	$R = f+1$	$(f+1)n$	t	t
Data-interweaved Replication (DIRFT)	$R = f+1$	$(f+1)n$	$\frac{1}{f+1}t$	t

Table I: Comparisons of theoretical performance bound in crash fault mode for different schemes

2) Silent Data Error Mode: In this mode, the system assumes that there is no crash faults in any replica and all replicas execute with almost equal speed. The system further assumes that the intermediate data can be corrupted by silent errors. From Equation 1, replication factor R can be determined as $2f_s + 1$. Hence, the framework initiates $2f+1$ replicas per task. Compared to the crash-fault mode, after each phase of the execution a hash-based signature is generated and communicated to the centralized Voting System. At the end of the $(f+1)^{th}$ phase, each logical sub-split is processed by $(f+1)$ distinct replicas. As a result, the Voting System is in possession of $(f+1)$ copies of signatures for each logical sub-partition. The Voting System then compares the signatures from different replicas per sub-split to detect any silent data corruption.

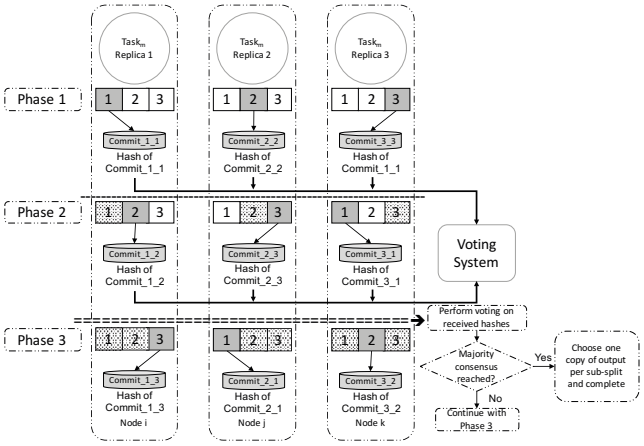


Figure 2: Inter-weaved execution of a replicated data split among three replicas in different phases of life cycle for silent data error mode with ability of recovering from single fault

For example, the system described in Figure 2 is configured to handle single silent data corruption and thus 3 replicas per task are launched to execute on three different data locations of same split. At phase 1, replicas 1, 2, and 3 consume sub-splits 1, 2, and 3 respectively and hash-based signatures of these partial outputs are sent to the Voting System. Similarly at phase 2, replicas move to their corresponding logical sub-splits and the same process is repeated. At the end of the second phase, Voting System performs comparisons on the received signatures. If a majority consensus is achieved, replicas complete execution and only a single copy of output per sub-split is kept, discarding the rests. Whereas when silent data error is detected, the faulty replica is discarded and the immediate previous replica is instructed to continue processing the left logical sub-split. Thus, at the end of third phase or in general after $(2f+1)^{th}$ phase, the system can recover from the silent data corruption.

Numerical Analysis of Performance: Assuming that a single task takes t unit of time to process the whole split, each phase takes $\frac{1}{2f+1}t$ unit of time having $2f+1$ replicas processing $2f+1$ logical sub-split in parallel. In case a majority consensus is reached and no data corruption is detected, the job completes at the end of $(f+1)^{th}$ phase with a reduced response time of $\frac{f+1}{2f+1}t$ unit of time. In contrast, a traditional full replication scheme, with no work sharing, will take t unit of time. However in presence of silent data error, both full replication and our scheme take up to t unit of time to complete. Note that both of the schemes use the same number of nodes, i.e, $(2f+1)n$ to execute the replicas in parallel. As for the example in Figure 2, with no silent data corruption the job completes at $\frac{2t}{3}$ time and in case of failure it is bounded by t.

However, the scheme proposed by Costa et al [18] op-

timizes the resource requirements for common case of no failure. Their scheme needs only $(f+1)n$ nodes compared to our and the traditional full replication schemes. Their scheme launches replicas in a two step fashion. In the first step, $f+1$ replicas are launched with each replica taking t time to consume the whole physical data block. At the end of the first step the voting is performed to determine faults. In case of no fault, their scheme takes t unit of time. Whereas if a silent data error is detected, f copies are launched to recover from the error which again takes up to t unit of time. Hence, in case of failure, the response time of the job stretches up to $2t$. This whole argument is summarized in the Table II.

Scheme	Replication Factor	Number of Nodes	Response Time	
			No Failure	Crash Fault
Byzantine Fault-tolerant MapReduce (BFT) [18]	$R = 2f + 1$	$(f + 1)n$	t	$2t$
Full Replication (FRFT)	$R = 2f + 1$	$(2f + 1)n$	t	t
Data-interweaved Replication (DIRFT)	$R = 2f + 1$	$(2f + 1)n$	$\frac{f+1}{2f+1}t$	t

Table II: Comparisons of theoretical performance bound in silent data error mode for different schemes

IV. Prototype

In Apache Hadoop ecosystem, processing frameworks and engines define a set of components responsible for operating on big data. Over the past decade, all major frameworks evolved to provide fault-tolerance for components like worker node running executors, and driver node running driver program in Spark Streaming, containers, and application manager in Samza, and ApplicationMaster, NodeManager, ResourceManager in YARN. However, the processing engine, actually operating on data, still relies on the *time redundancy* to deal with crash faults. Our proposed scheme addresses the concern of fault-tolerance against both crash faults and silent data errors uniformly for the tasks responsible for processing input data blocks. We prove the applicability of our proposed model by extending the original Hadoop MapReduce.

The changes introduced to MapReduce code base by our implementation can be summarized as following. Since our scheme replicates tasks, along with *SplitId* and *AttemptId*, we have added ReplicaId to differentiate between replicas of the same task. Since, *JobTracker* maintains information of a running job, as an object of *JobInProgress*, which becomes centralized store of information, the *Voting System* is created as a private member of *JobInProgress*. *JobInProgress* is also responsible for initializing and scheduling of tasks. Hence, we added logic of creation and placement of replicas of tasks to the same. *SHA-256* based signatures are generated on partial outputs and sent along with *heartbeat* signal to reduce communication overhead and also are not stored on the stable storage along with partial outputs. We have added a parameterized fault injection module to the *JobTracker*. Crash faults are emulated deterministically by injecting exceptions at the end of task completion. Silent data corruptions are emulated by tampering the signatures sent along with *heartbeat* to the centralized *VotingSystem*. As the *VotingSystem* compares the signatures on sub-splits to detect any fault, we enforce a logical barrier between multiple phases while removing any optimization to overlap executions of phases. Moreover, as our scheme launches all replicas to run in parallel, we have also disabled *Speculative Execution* of tasks.

The prototype framework can be configured by setting appropriate values to the following parameters in *mapred-site.xml*: (i) *mapred.map.tasks.fault.inject*, a boolean type to denote whether the framework emulates fault injection or not, (ii) *mapred.map.tasks.fault.tolerance*, an integer value specifying the number of faults that can be tolerated without aborting the job, (iii) *mapred.map.tasks.fault.nature*, an integer value to determine the nature of faults with 1 representing crash faults, and 2 for silent-data errors, and (iv) *mapred.map.tasks.fault.percent*, to signify the percentage of tasks that will be affected by fault emulation.

V. Evaluation

A. Experimental Setup

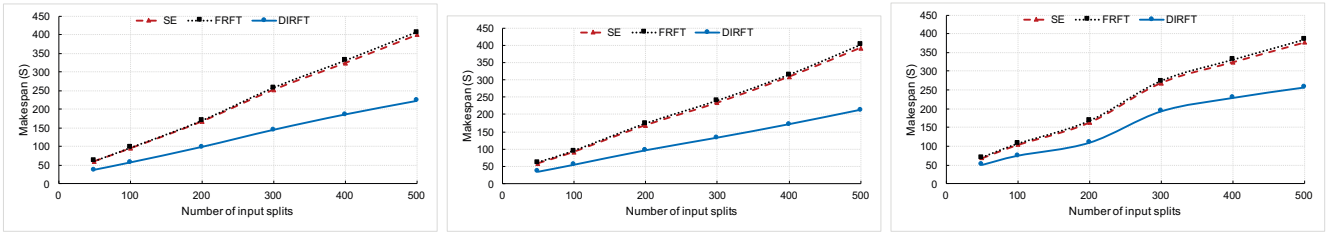
A set of experiments are performed on the Bridges system at the Pittsburgh Supercomputing Center (PSC). Each node has 128 GB RAM and 28 Intel(R) Xeon(R) CPU E5-2695 v3 cores, running at 2.30GHz. Each node is divided equally in two NUMA zones. The size of the L1 instruction and data cache is 32KB each and the L2 cache size is 256KB per core. The L3 cache per NUMA socket is 32MB. The operating system running on each node is CentOS 7.2.1511, with 3.10.0 Linux kernel. Communication between nodes is achieved using Gigabit Ethernet.

B. Benchmark

The *GridMix2* Hadoop benchmark is a combination of synthetic jobs, which models Cloud workloads. In order to test our proposed algorithm, three sub-benchmarks, namely *StreamSort*, *Combiner*, and *JavaSort*, are used. These sub-benchmarks process uncompressed data. It is to be noted that the behavior induced by the uncompressed workload does not affect tasks and daemons. This ensures that the behavior across different runs is repeatable and unaffected by the nature of the benchmark workload.

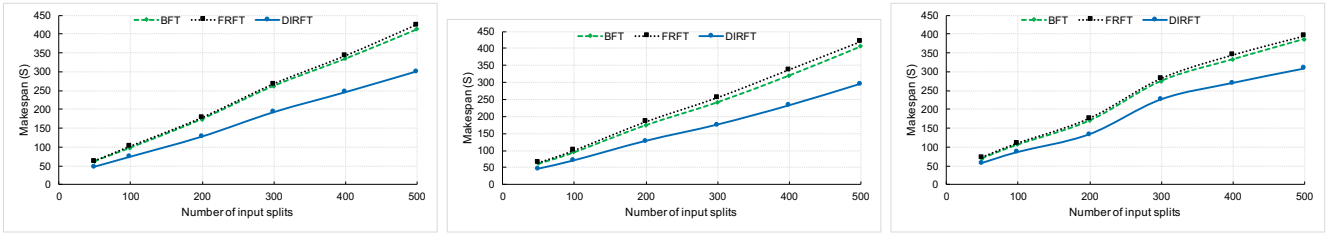
C. Baseline

We compare the proposed *Data-interweaved Replication (DIRFT)* scheme to (i) *MapReduce Speculative Execution (SE)*, (ii) the traditional *Full Replication (FRFT)* scheme, which doesn't interweave the processing of input data from different offsets, and (iii) the *Byzantine Fault-tolerance (BFT)* scheme proposed by Costa et al [18]. We measure the makespan of the jobs using the same system configurations running on different code deployments. The number of nodes for different code bases is varied, such that a constant ratio



(a) StreamSort (b) JavaSort (c) Combiner

Figure 3: Makespan vs. split size for three different benchmarks in crash-fault mode



(a) StreamSort (b) JavaSort (c) Combiner

Figure 4: Makespan vs. split size for three different benchmarks in silent data error mode

of number of running tasks to node is maintained. For all experiments, the split size is set to be 129MB.

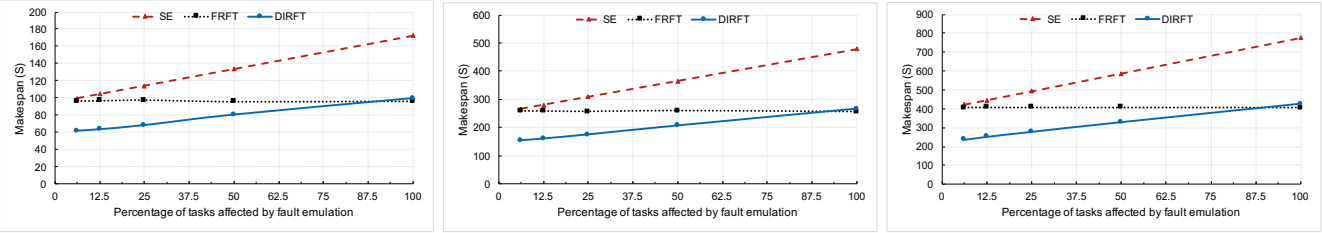
D. Sensitivity of split size on response time with no fault injection

The objective of this experiment is to analyze the impact of variations in split sizes on the makespan of the jobs, with no task failure. Assuming low fault rates, this experiment models the common behavior typically observed in a lifetime of a Cloud Computing cluster. In this experiment, we vary the split size from 50 to 500 and compare the job execution time for three mini-applications in GridMix2 benchmarks, executing on different versions of code. For this experiment, all code bases are configured in both crash-fault and silent data error mode. A single fault is assumed, resulting in a system is configured with $f = 1$. The statistics, collected from the execution of the three benchmarks, show similar trends in terms of performance behavior. The discussion of each fault-mode follows.

1) Crash-fault mode: Using the parameters listed in Table I, we provision the experimental test bed with 10, 20, and 20 nodes for SE, FRFT, and DIRFT respectively. This is to ensure a constant ratio of number of tasks to nodes for different code bases. The results depicted in Figure 3 show that DIRFT on an average achieves 39% to 45% performance improvement over SE and FRFT respectively for all the benchmarks. FRFT shows similar performance as SE with an additional operating overhead of 2%. Note that, the deviation from the expected theoretical latency reduction in by 50% can be attributed to the operating overhead due to running multiple copies of tasks in parallel.

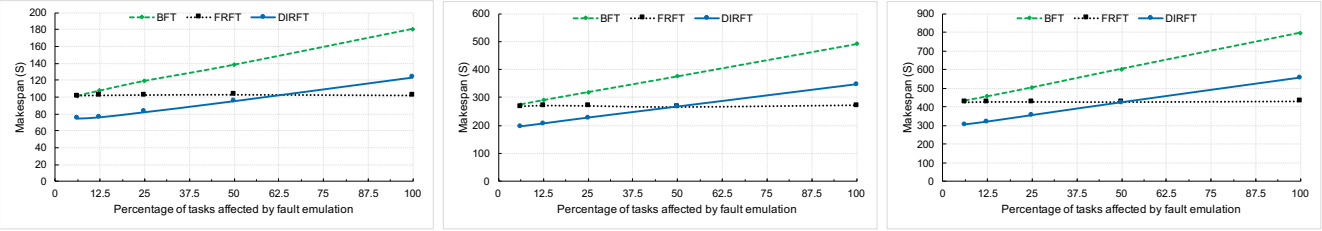
2) Silent data error mode: Following the BFT, FRFT, and DIRFT parameters specified in Table II, we provision the experimental test bed with 20, 30, and 30 nodes, respectively. Note that a separate node is used in all experiments to run the centralized system components, namely JobTracker and NameNode. From Figure 4, we can see BFT and FRFT show similar performance with FRFT, incurring on an average 3.5% operating overhead across all the benchmarks compared to baseline scheme of BFT. DIRFT outperforms BFT by 25% to 28% on an average for split size ranging from 50 to 500 across all benchmark in contrast to the theoretical latency reduction by 33%. This can be attributed to the incurred communication overhead between replicas and the centralized Voting System.

The key enabler of our scheme is hardware redundancy and availability of idle resources. If all other schemes would have been provisioned with the same resources as SE, it would multiply the ratio of tasks to node compared to the current setup. Thus, the replicas will be forced to timeshare underlying hardware resources, thereby failing to achieve the observed improvement in latency. Moreover, our results argue the need for a bigger split size, e.g., 129MB in our current setup. This is due to the fact the with smaller split sizes, time to process input splits is smaller and thus the operating overhead of initiating and managing multiple replicas and the communication cost will dominate the makespan. This will diminish the effectiveness of the proposed scheme.



(a) StreamSort with 100 input splis (b) StreamSort with 300 input splis (c) StreamSort with 500 input splis

Figure 5: Makespan vs. percentage of tasks affected by fault emulation in crash-fault mode.



(a) StreamSort with 100 input splis (b) StreamSort with 300 input splis (c) StreamSort with 500 input splis

Figure 6: Makespan vs. percentage of tasks affected by fault emulation in silent data error mode.

E. Sensitivity of percentage of tasks in fault on response time

This experiment is designed to assess the impact of the percentage of tasks affected by fault emulation on the job makespan. The experiment runs a single benchmark, StreamSort, for three different split sizes namely, 100, 300, and 500. We vary the fault percentage from 6.25 to 100 in geometric progression and compare the makespans of the jobs running on different code bases. Although the percentage of tasks affected by faults in a real environment can never go to 100%, we present our findings to validate the theoretical upper bound on the performance of the different schemes. Findings of the individual fault-modes performance are discussed next.

1) Crash-fault mode: Figure 5 presents the results. As FRFT initiates execution of all replicas in parallel, the makespan of jobs for FRFT is not affected by the variation of fault percentage. Results show that, SE exceeds the targeted response latency by 3% to 79%, 3.21% to 86.6%, and 3.95% to 90.5% for split sizes of 100, 300, and 500, respectively, with percentage of faults varying from 6.25% to 100%. This is because SE re-executes more tasks, with higher percentage of faults. As DIRFT interweaves the execution of replicas on different logical sub-splits, it theoretically never violates the SLA, even in case of 100% fault-percentage. However, the experimental results show that DIRFT exceeds the agreed upon response time by 3.4%, 4.2%, and 4.5% for three different split sizes, respectively.

2) Silent data error mode: The results presented in Figure 6, show that the overhead incurred by BFT in comparison to FRFT ranges from 0% to 77.2%, 2.4% to 81.6%, and 1.7% to 85.3% for split sizes of 100, 300, and 500 respectively. The results also show that DIRFT achieves on average 31% improvement in latency over BFT, assuming that 100% tasks are affected by faults. Table II suggests that even with 100% failure rate, DIRFT should not exceed agreed upon response time. However, the experimental results show that, DIRFT incurs operating overhead of 21%, 28%, and 30% over theoretical bounds assuming 100% task failure rate.

VI. Conclusion

Data-interweaved fault tolerance scheme is proposed as a computational model to deal uniformly with crash faults and silent data errors in data-intensive environments. Experimental evaluation of the prototype implementation shows that in case of no failure, the scheme achieves a significant reduction in latency. Moreover, in presence of failure, the scheme closely guarantees the SLA-specified latency while incurring minimal operating cost. What, differentiates DIRFT from its traditional counterparts, is that the replicas collaboratively process the input data in an inter-weaved fashion. The ability of the scheme to leverage the fact that the data is already replicated in distributed storage for fault-tolerance and to execute the replicas on logical sub-splits from different offsets increases the degree of parallelism. Although the prototype is based on MapReduce, DIRFT, however, is not limited to a specific implementation. The scheme can be incorporated into other large data processing

frameworks, such as Spark and Flink, to achieve similar fault-tolerance performance goals.

ACKNOWLEDGMENT

This material is based in part upon work supported by the Department of Energy under Grant Number DOE-411366. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575 [19], [20]. Specifically, it used the Bridges system, which is supported by NSF award number ACI-1445606, at the Pittsburgh Supercomputing Center (PSC).

REFERENCES

[1] "Hadoop," http://hadoop.apache.org/, Accessed July 04, 2016.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[3] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *Proceedings of the International Conference on Dependable Systems and Networks*, ser. DSN '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 249–258.

[4] "Speed matters for google web search," https://services.google.com/fh/files/blogs/google_delayexp.pdf, Accessed July 04, 2016.

[5] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: A large-scale field study," in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '09. New York, NY, USA: ACM, 2009, pp. 193–204.

[6] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: An empirical analysisof hardware failures on a million consumer pcs," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 343–356.

[7] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, ser. HPCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 243–247.

[8] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Why let resources idle? aggressive cloning of jobs with dolly," in *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, ser. HotCloud'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 17–17.

[9] S. Govindan, A. Sivasubramaniam, and B. Urgaonkar, "Benefits and limitations of tapping into stored energy for datacenters," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 341–352.

[10] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42.

[11] 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 *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 265–278.

[12] A. Vulimiri, O. Michel, P. B. Godfrey, and S. Shenker, "More is less: Reducing latency via redundancy," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XI. New York, NY, USA: ACM, 2012, pp. 13–18.

[13] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta, "Making cloud intermediate data fault-tolerant," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 181–192.

[14] W. T. Tsai, P. Zhong, J. Elston, X. Bai, and Y. Chen, "Service replication strategies with mapreduce in clouds," in *2011 Tenth International Symposium on Autonomous Decentralized Systems*, March 2011, pp. 381–388.

[15] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[16] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[17] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient byzantine fault-tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, Jan 2013.

[18] P. Costa, M. Pasin, A. N. Bessani, and M. Correia, "Byzantine fault-tolerant mapreduce: Faults are not just crashes," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, Nov 2011, pp. 32–39.

[19] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr, "Xsede: Accelerating scientific discovery," *Computing in Science Engineering*, vol. 16, no. 5, pp. 62–74, Sept 2014.

[20] N. A. Nystrom, M. J. Levine, R. Z. Roskies, and J. R. Scott, "Bridges: A uniquely flexible hpc resource for new communities and data analytics," in *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, ser. XSEDE '15. ACM, 2015, pp. 30:1–30:8.