Making Cloud Intermediate Data Fault-Tolerant*

Steven Y. Ko* Imranul Hoque† Brian Cho† Indranil Gupta†

*Dept. of Computer Science
Princeton University
Princeton, NJ, USA
steveko@cs.princeton.edu

†Dept. of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, USA
{ihoque2, bcho2, indy}@illinois.edu

ABSTRACT

Parallel dataflow programs generate enormous amounts of distributed data that are short-lived, yet are critical for completion of the job and for good run-time performance. We call this class of data as *intermediate data*. This paper is the first to address intermediate data as a first-class citizen, specifically targeting and minimizing the effect of run-time server failures on the availability of intermediate data, and thus on performance metrics such as job completion time. We propose new design techniques for a new storage system called ISS (Intermediate Storage System), implement these techniques within Hadoop, and experimentally evaluate the resulting system. Under no failure, the performance of Hadoop augmented with ISS (i.e., job completion time) turns out to be comparable to base Hadoop. Under a failure, Hadoop with ISS outperforms base Hadoop and incurs up to 18% overhead compared to base no-failure Hadoop, depending on the testbed setup.

Categories and Subject Descriptors

D.4.3 [**File Systems Management**]: Distributed File Systems

General Terms

Design, Performance, Reliability

Keywords

Replication, Intermediate Data, Interference Minimization, MapReduce

1. INTRODUCTION

Parallel dataflow programming frameworks such as MapReduce [13], Dryad [24], Pig [31], and Hive [16] have gained popularity for large-scale parallel data processing. Today, many organizations are using Hadoop, an open-source implementation of MapReduce, for their data processing needs, e.g., A9.com, AOL, Facebook, The New York Times, Yahoo!, etc. [32]. Parallel programs written in these frameworks run inside a variety of data centers ranging from private clouds (e.g., Microsoft's AdCenter logs [15], the M45 cloud, etc.) to public clouds (e.g., AWS's EC2, Cloud Computing Testbed at University of Illinois, etc.).

This paper focuses on an often-ignored citizen that arises inside clouds during the execution of such parallel dataflow programs: *intermediate data*. Intermediate data is defined as the data which is generated during the execution of a parallel dataflow program directly or indirectly from the input data, but excludes the final output data as well as the input data.

More concretely, a parallel dataflow program consists of multiple stages of computation, and involves communication patterns from each stage to its succeeding stage(s). For example, Figure 1 shows an example dataflow graph of a Pig program [31], which is compiled as a sequence of MapReduce jobs. The communication pattern is either all-to-all (from a Map stage to the next Reduce stage) or one-to-one (from a Reduce stage to the next Map stage). Another dataflow framework – Dryad [24] – is also stage-based in some ways and involves significant amounts of inter-stage communication (although the details are different from Pig and Hadoop).

In all these frameworks, at run time, intermediate data is produced as an output from the collective tasks of each stage of the dataflow program, and is used as an input for the tasks of the next stage. For instance, in Hadoop or MapReduce, the output of the Map stage serves as input to the next Reduce stage, and is thus intermediate data. Similarly, in a multi-stage computation (e.g., Pig), the output of a Reduce stage that serves as input for the next Map stage, is also intermediate data.

Aside from being distributed amongst the servers of the datacenter, the aggregate amount of intermediate data generated during a dataflow program can be enormous. A single MapReduce job may generate intermediate data in a ratio of 1:1 compared to the input (e.g., sort) or higher (e.g., wordcount, which generates extra tuple attributes). A chain of 10 MapReduces may generate intermediate:input data ratio as high as 10:1. Even with one MapReduce job, if the job joins two tables (e.g, with Pig's join primitive), the output becomes quadratic in size compared to the input. Consider the fact that typical input sizes into these parallel programs are massive, e.g., Yahoo!'s Web graph generation

*This work was supported in part by NSF grants IIS 0841765 and CNS 0448246.

SoCC'10, June 10–11, 2010, Indianapolis, Indiana, USA.

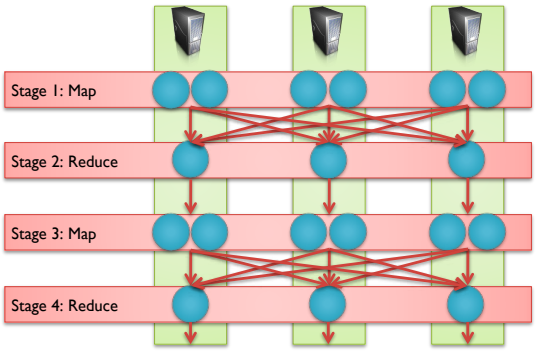

Figure 1: An example of Pig executing a linear sequence of MapReduce stages. The Shuffle phase involves all-to-all data transfers, while local data is used between each Reduce and the next Map.

called *WebMap* takes 280TB of input, Facebook processes 55TB per day [19]. This means that the scale of intermediate data may be several orders of magnitude larger than input of output data, and thus range in many GBs or TBs.

Classical studies on traditional file systems [3, 37] have revealed the extent of intermediate data inside them. These studies showed that short-lived files (e.g., temporary .o files) comprise the majority of files both in the file system as well as in its workload. We found some of these characteristics in cloud intermediate data as well – it is short-lived, used immediately, written once and read once. Yet, there are some new characteristics – the blocks are distributed, large in number, large in aggregate size, and a computation stage cannot start until all its input intermediate data has been generated by the previous stage. This large-scale, distributed, short-lived, computational-barrier nature of intermediate data creates network bottlenecks because it has to be transferred in-between stages [13].

This means that cloud intermediate data is critical to completion and performance of the parallel dataflow program. Unless appropriate intermediate data is available, tasks of the next stage may not be able to start. In some cases, the entire next stage may not be able to start, e.g., the Reduce stage in Hadoop. Worse still, our experiments indicate that server failures, which are the norm in clouds, can lead to loss of intermediate data. This prolongs job completion times significantly. For instance, we found that in a small-scale Hadoop application, a single failure could prolong completion time by 50%.

Failures are a major concern during run-time execution of dataflow programs. Reported experiences with dataflow frameworks in large-scale environments indicate that both transient (i.e., fail-recovery) and permanent (i.e., fail-stop) failures are prevalent, and will only exacerbate as more organizations process larger data with more and more stages. For example, Google reports 5 average worker deaths per MapReduce job in March 2006 [12], and at least one disk failure in every run of a 6-hour MapReduce job with 4,000 machines [29]. Yahoo! reports WebMap has grown to a chain of 100 MapReduce jobs [19].

Existing solutions to handling cloud intermediate data fall at two ends of a spectrum. On one end, the data is stored locally on the native file system, and is fetched remotely by tasks of the next stage (store-local approach). Data is not replicated. For instance, this approach is used for Map outputs in Hadoop and Pig, as well as in Dryad. This extreme of the spectrum is efficient, but not fault-tolerant: failure of a server storing intermediate data causes the re-execution of affected tasks. On the opposite end of the spectrum, data is written back to the distributed file system (DFS approach), where it is automatically replicated. This ensures fault-tolerance but incurs significant network overhead and thus prolongs job run times. For instance, this approach is used for Reduce outputs in Hadoop and Pig.

In the cloud intermediate data problem, our goal is to achieve as good performance as the store-local approach described above, while achieving fault-tolerance that is closer to that of the DFS approach above. This problem is challenging. Consider the store-local approach for a multi-stage dataflow program. We discovered that in this approach, a failure of a single server can result in what we call *cascaded re-execution*: some tasks in *every stage from the beginning* have to be re-executed sequentially up to the stage where the failure happened (more details are in Section 3.2). Our challenge then is to augment the fault-tolerance of this approach while keeping the overheads low.

We address this challenge by proposing three techniques, implementing them for Hadoop, and experimentally evaluating the same. In a nutshell, our techniques include: (i) asynchronous replication of intermediate data, (ii) replication of selective intermediate data, and (iii) exploiting the inherent bandwidth heterogeneity of datacenter topology to do the same. Our techniques have to create fault-tolerance while *minimizing interference encountered by foreground network traffic* which is carrying intermediate data that is directly critical to job completion time.

Our main contribution is a new storage system, called ISS (Intermediate Storage System) that implements the above three techniques. ISS replicates both the Map and Reduce outputs with significantly less overhead compared to base Hadoop, and thus prevents cascaded re-execution for multi-stage MapReduce programs. Under no failure, the performance of Hadoop augmented with ISS (i.e., job completion time) is comparable to base Hadoop. In a transient failure injection experiment, ISS has incurred only 9% overhead compared to base no-failure Hadoop, while base Hadoop with one transient failure has incurred 32% overhead. Under a permanent failure, ISS has incurred 0%-18% overhead compared to base no-failure Hadoop, while base Hadoop with one permanent failure has incurred 16%-59% overhead. Finally, while the implementation and evaluation of this paper leverages today's most popular open-source dataflow programming framework (i.e., Hadoop), we believe that our techniques are applicable to a wide variety of parallel programming frameworks that are not yet open-sourced (e.g., Dryad).

This paper is an extended version of our workshop paper [25].

2. BACKGROUND: MAPREDUCE

Since much of our discussion involves MapReduce, we briefly summarize how this base system works. The goal of our discussion here is not to have a comprehensive introduction to MapReduce, but rather to provide a primer before the rest of the paper.

Overview of MapReduce.

The MapReduce framework is a runtime system that uti-

lizes a cluster of machines, often dedicated to the framework. The framework takes two input functions, `Map` and `Reduce`, written by a programmer, and executes each function in parallel over a set of distributed data files. A distributed file system such as GFS [17] or HDFS [21] is used to store input and output data. There are currently two implementations of MapReduce – the original MapReduce from Google that has not been released to the public, and an open-source implementation called Hadoop from Yahoo!.

Three Phases of MapReduce.

There are three phases that every MapReduce program execution can be divided into. They are Map, Shuffle, and Reduce. Each phase utilizes every machine dedicated to the MapReduce framework. The phases are the following:

1. Map: The Map phase executes the user-provided Map function in parallel over the MapReduce cluster. The input data is divided into chunks and stored in a distributed file system, e.g., GFS or HDFS. Each Map task reads some number of chunks from the distributed file system and generates intermediate data. This intermediate data is used as the input to the Reduce phase. In order to execute the Reduce function in parallel, the intermediate data is again partitioned and organized into a number of chunks. These chunks are stored locally on the nodes that generate them.

2. Shuffle: The Shuffle phase moves the intermediate data generated by the Map phase among the machines in the MapReduce cluster. The communication pattern is all-to-all as shown in Figure 1. Each piece of intermediate data is sent to the appropriate Reduce task. Due to this all-to-all nature of communication, this phase heavily utilizes the network, and very often is the bottleneck [13].

3. Reduce: The Reduce phase executes the user-provided Reduce function in parallel over the MapReduce cluster. It stores its output in the distributed file system. If there is only one MapReduce job, this output is the final output. However, in a multi-stage MapReduce, this output is the intermediate output for the next MapReduce job. This helps the user run a chain of MapReduce jobs, e.g., Yahoo!'s *WebMap* [19].

The most important aspect of these three phases with respect to our discussion in the rest of the paper is that the intermediate data is stored locally in the Map phase, transferred remotely in the Shuffle phase, and read into the Reduce phase.

3. WHY STUDY INTERMEDIATE DATA?

In this section, we discuss some salient characteristics of intermediate data, and outline the requirements for a system that manages it as a first class citizen.

3.1 Characteristics of Intermediate Data

Persistent data stored in classical file systems ranges in size from small to large, is likely read multiple times, and is typically long-lived. In contrast, intermediate data generated in cloud programming paradigms has unique characteristics. Through our study of MapReduce, Dryad, Pig, etc., we have gleaned three main traits that are common to intermediate data in all these systems. We discuss them below.

Size and Distribution of Data: Unlike traditional file system data, the intermediate data generated by cloud computing paradigms has: (1) a large number of blocks, (2) a large aggregate size between consecutive stages, and (3) distribution across a large number of nodes. As discussed earlier in Section 1, a chain of n MapReduces may generate intermediate:input data in a ratio of anywhere up to (and perhaps even higher than) n:1.

Write Once-Read Once: Intermediate data typically follows a write once-read once pattern. Each block of intermediate data is generated by one task only, and read by one task only. For instance, in MapReduce, each block of intermediate data is produced by one Map task, belongs to a region (or partition), and is transmitted to the unique Reduce task assigned to the region.

Short-Lived and Used-Immediately: Intermediate data is short-lived because once a block is written by a task, it is transferred to (and used immediately by) the next task. For instance, in Hadoop, a data block generated by a Map task is transferred during the Shuffle phase to the block's corresponding Reduce task.

In summary, intermediate data is critical to completing the job, i.e., no intermediate data can be lost before use. It is thus critical to the key performance metric, namely job completion time. The above three characteristics morph into major challenges at runtime when one considers the effect of failures. For instance, when tasks are re-executed due to a failure, intermediate data may be read multiple times or generated multiple times, prolonging the lifetime of the intermediate data. Thus, failures lead to additional overhead for generating, writing, reading, and storing intermediate data, eventually increasing job completion time.

3.2 Effect of Failures

We discuss the effect of failures on dataflow computations. Consider the dataflow computation in Figure 1 using Pig. Now, suppose that a failure occurs on a node running task t at stage n (e.g., due to a disk failure, a machine failure, etc.). Since Pig relies on the local filesystem to store intermediate data, this failure results in the loss of all the intermediate data from stage 1 to $(n-1)$ stored locally on the failed node (see Figure 1). When a failure occurs, Pig will reschedule the failed task t to a different node available for re-execution. However, the re-execution of t cannot proceed right away, because some portion of its input is lost by the failed node.

More precisely, the input of task t is generated by all the tasks in stage $(n-1)$ including the tasks run on the failed node. Thus, those tasks that ran on the failed node have to be re-executed to regenerate the lost portion of the input for task t. In turn, this requires re-execution of tasks run on the failed node in stage $(n-2)$, and inductively this cascades all the way back to stage 1. Thus, some tasks in every stage from the beginning will have to be re-executed sequentially up to the current stage.

We call this phenomenon as *cascaded re-execution*, and Figure 2 illustrates this problem. Although we present this problem using Pig as a case study, most dataflow frameworks with multiple stages will suffer from this problem.

Figure 3 shows the effect of a single failure on the runtime of a Hadoop job (i.e., a two-stage job with 1 Map and 1

Name	Topology	Bandwidth	# of Nodes	Input Data	Workload
S-Unif	1 core switch connecting 4 LANs (5 nodes each)	100 Mbps	20	2GB/Node	Sort
M-Het	1 core switch connecting 4 LANs (20 nodes each)	100 Mbps Top-of-the-Rack Switch 1 Gbps Core Switch	80	2GB/Node	Sort
M-Unif	Same as M-Het	1 Gbps	80	2GB/Node	Sort

Table 1: Experimental Setup (All Using Emulab; More Details in Section 7)

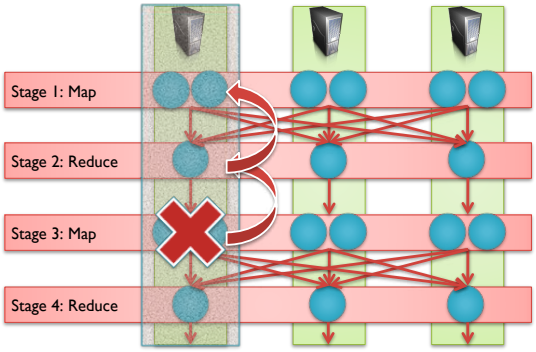

Figure 2: A Cascaded Failure Example

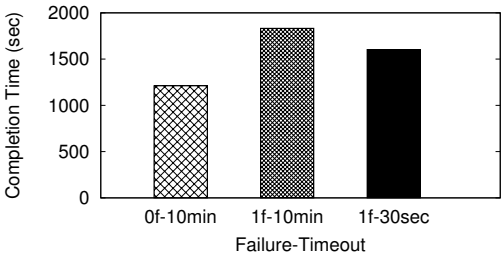

Figure 3: Effect of a Failure on a Hadoop Job (S-Unif)

Reduce). [1] The failure is injected at a random node immediately after the last Map task completes. The leftmost bar depicts the runtime without failures. The middle bar shows the runtime with 1 failure, when Hadoop's node failure detection timeout is 10 minutes (the default) – *a single failure causes a 50% increase in completion time.* Further reducing the timeout to 30 seconds does not help much – the runtime degradation is still high (33%). [2]

To understand this further, Figures 4 and 5 show the number of tasks over time, for two bars from Figure 3: 0f-10min and 1f-30sec. Since Hadoop creates one task per input chunk, there are typically many more Map and Reduce tasks than available machines. This has two implications in the execution of Hadoop. First, each machine typically processes more than one Map task and one Reduce task. Second, some Map tasks finish earlier than others, so the Shuffle phase may overlap with the Map phase as shown in both Figure 4 and 5.

Figure 4 shows clearly the barrier – Reduce tasks do not start until the Shuffle is (almost) done around t=925 sec. We made several observations from the experiment of Figure 5: (1) a single node failure caused several Map tasks to be re-executed (starting t=925 sec), (2) a renewed Shuffle phase started after these re-executed Maps finish (starting t=1100 sec), and (3) Reduces that were running on the failed node and that were not able to Shuffle data from the failed node, got re-executed as well towards the end (t=1500 sec).

[1]Experimental setup is shown in Table 1

[2]The main reason that Hadoop chose the 10-minute timeout is to suppress false-positives, i.e., to avoid mistakenly declaring a node as a dead node while it is still alive. Thus, we use the default timeout for the rest of the experiments.

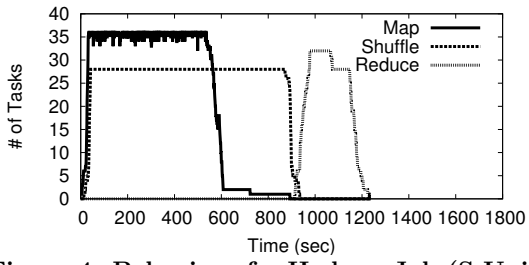

Figure 4: Behavior of a Hadoop Job (S-Unif)

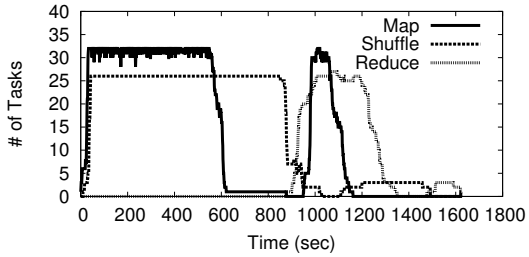

Figure 5: Behavior of a Hadoop Job under 1 Failure (S-Unif)

While the above experiment shows cascaded re-execution within a single MapReduce stage, it indicates that in multi-stage dataflow computations, a few node failures will cause far worse degradation in job completion times.

3.3 Requirements

Based on the discussion so far, we believe that the problem of managing intermediate data generated during dataflow computations, deserves deeper study as a first-class problem. Motivated by the observation that the main challenge is dealing with failure, we arrive at the following two major requirements that any effective intermediate storage system needs to satisfy: *availability* of intermediate data, and *minimal interference* on foreground network traffic generated by the dataflow computation. We elaborate below.

Data Availability: A task in a dataflow stage cannot be executed if the intermediate input data is unavailable. A system that provides higher availability for intermediate data will suffer from fewer delays for re-executing tasks in case of failure. In multi-stage computations, high availability is critical as it minimizes the effect of cascaded re-execution (Section 3.2).

Minimal Interference: At the same time, data availability cannot be pursued over-aggressively. In particular, since intermediate data is used immediately, there is high network contention for foreground traffic of the intermediate data transferred to the next stage, e.g., by Shuffle in MapReduce [13]. An intermediate data management system needs to minimize interference on such foreground traffic, in or-

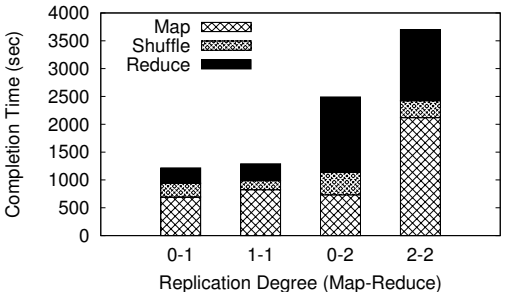

Figure 6: Using HDFS (S-Unif): Varying Replication Degree (i-j) for Output of Map (i) and Reduce (j)

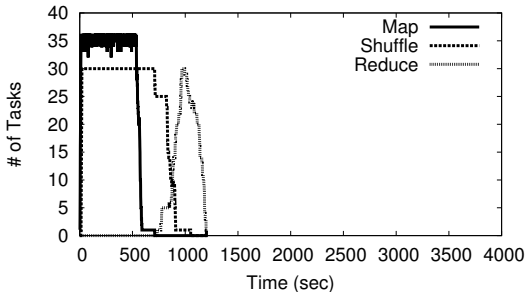

Figure 7: Map Replication: 0, Reduce Replication: 1 (S-Unif)

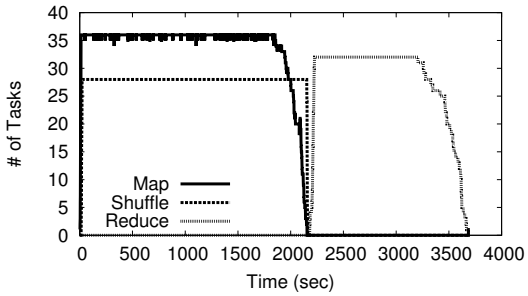

Figure 8: Map Replication: 2, Reduce Replication: 2 (S-Unif)

der to keep the job completion time low, especially in the common case of no failures.

4. EXPLORING THE DESIGN SPACE

Existing approaches to managing intermediate data are at two ends of a spectrum – they are either store-local or DFS. Store-local means storing intermediate data locally at the outputting node and having it read remotely. Current dataflow frameworks thus use purely *reactive* strategies to cope with node failures or other causes of data loss. Hence, in Hadoop, there is no mechanism to ensure intermediate data availability. The loss of Map output data results in the re-execution of those Map tasks, with the further risk of cascaded re-execution (Section 3.2).

In contrast, the other end of the spectrum – DFS – uses a distributed file system that replicates intermediate data. The unanswered question towards this end of the spectrum is: how much interference does the replication process cause to the foreground job completion time? Thus, a natural approach to satisfying both requirements is to start with an existing distributed file system, determine how much interference it causes, and reason about how one can lower the interference. We do so next.

4.1 Replication Overhead of HDFS

As our first step, we experimentally explore the possibility of using a distributed file system especially designed for data-intensive environments. We choose HDFS, which is used by Hadoop to store the input to the Map phase and the output from the Reduce phase. We modified Hadoop so that HDFS was also used to store the intermediate output from the Map phase.

Figure 6 shows four bars, each annotated i-j, where i is the replication degree within HDFS for Map output (i=0 being the default local write-remote read) and j the replication degree for Reduce output. When one incorporates HDFS to store Map data into HDFS, there is only a small increase in completion time (see 1-1 vs. 0-1). This is because the only additional overheads are HDFS metadata that point to the local copy of the output already stored at the Map node.

Increasing the Reduce replication degree to 2, on the other hand (see 0-2 vs. 0-1) doubles the job completion time. [3] Further, replicating Map output increases the completion time by a factor of about 3 compared to the default (see 2-2 vs. 0-1). To delve into this further, we compare the timeline of tasks run by Hadoop without replication in Figure 7 and with replication in Figure 8. We observe that the Map runtime increases by a factor of over 3, Shuffle runtime by a factor of 2, and Reduce runtime by a factor of around 2.

Hence, we conclude that using HDFS as-is will not work, because the interference it generates degrades job completion time significantly.

[3]This plot seems to indicate why the Pig system (built atop Hadoop) uses a default replication degree of 1 for Reduce.

4.2 Background Transport Protocols

Another approach to reduce the interference may be to use a background transport protocol beneath HDFS (such as TCP-Nice [36] or TCP-LP [26]), so that we could replicate intermediate data without affecting foreground traffic. We discuss this possibility qualitatively here. We focus our discussion around TCP-Nice, a well-known background transport protocol. However, we believe our discussion below is generally applicable to any background transport protocol.

TCP-Nice allows a flow to run in the "background" with little or no interference to normal flows. These background flows only utilize "spare" bandwidth unused by normal flows. This spare bandwidth exists because there is local computation and disk I/O performed in both Map and Reduce phases. Thus, we could put replication flows in the background using TCP-Nice, so that they would not interfere with the foreground traffic such as Shuffle.

The first drawback with this approach is that TCP-Nice (as well as any background transport protocol) is designed as a general transport protocol. This means that it does not assume (and is thus unable to utilize) the knowledge of applications using it and the environments in which it is operating. For example, TCP-Nice does not know which flows are foreground and which are not. Thus, TCP-Nice gives background flows lower priority than any other flow in the network. This means that a background replication flow will get a priority lower than Shuffle flows, as well as other flows unrelated to the dataflow application, e.g., any ftp or http traffic going through the same shared core switch of a data center.

In addition, the replication of intermediate data should be done as quickly as possible in order to provide maximum protection against failures. This is because cascaded re-execution can be stopped only after intermediate data is completely replicated. However, TCP-Nice minimizes interference at the expense of network utilization, which could slow down the replication process. In fact, the original paper on TCP-Nice [36] makes clear that network utilization is not a design goal.

Finally, TCP-Nice only deals with data transfer part of replication. Since replication is a process that involves more than just data transfer, there are other opportunities (e.g., replica placement, data selection, etc.) that one can explore in addition to data transfer rate-control. We explore these other opportunities to reduce interference as we discuss in the next section.

5. THREE TECHNIQUES FOR LOWERING INTERFERENCE

Any intermediate data storage system needs to have design decisions regarding three questions. These questions are: 1) how much consistency the system guarantees, 2) where to place a replica, and 3) which data to replicate. In this section, we examine these three questions with the primary focus on interference reduction.

We describe three techniques that are incorporated in our ISS system. These techniques leverage a couple of observations. First, a dataflow programming framework typically runs in a datacenter-style environment, where machines are organized in racks and connected with a hierarchical topology. Second, as we discussed in Section 3.1, intermediate data generated by dataflow programming frameworks is written once by a single writer, and read once by a single reader.

Asynchronous Replication.

Our first technique is asynchronous replication, which allows writers to proceed without waiting for replication to complete. In comparison, HDFS uses synchronous replication, where writers are blocked until replication finishes. These represent two design points with regards to consistency. HDFS can guarantee strong consistency because if a writer of block A returns, all the replicas of block A are guaranteed to be identical, and any reader of block A can read any replica. However, the disadvantage of this approach is that the performance of writers might suffer because they have to be blocked.

Asynchronous replication cannot provide the same level of consistency guarantee as synchronous replication, since even if a writer of block A returns, a replica of block A may still be in the process of replication. However, the performance of writers can be improved due to its non-blocking nature. For example, in Hadoop, asynchronous replication allows Map and Reduce tasks to proceed without blocking.

Since we aim to minimize performance interference, we employ asynchronous replication in ISS. In essence, strong consistency is not required for dataflow programming frameworks because there are only a single writer and a single reader for intermediate data, as discussed in Section 3.1.

Rack-Level Replication.

Typically, a dataflow programming framework runs in a datacenter-style environment, where machines are organized in racks, and the network topology is hierarchical, e.g., 2-level with top-of-the rack switches and a core switch [9]. Figure 9 shows an example.

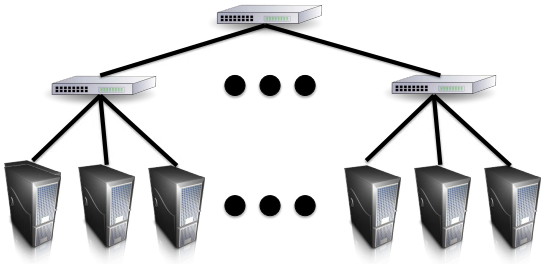

Figure 9: A 2-Level Topology Example

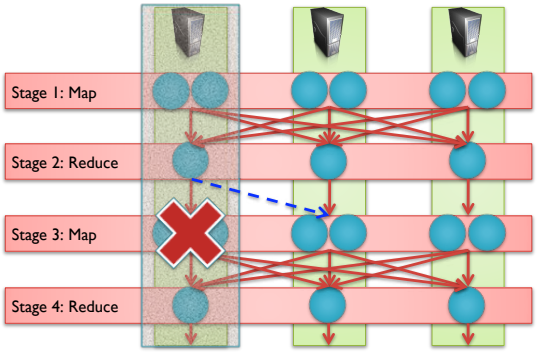

Figure 10: A Failure Example

In a hierarchical topology, the bottleneck is the core switch because it is shared by many racks and machines. Thus, there is inherent heterogeneity in bandwidth – inter-rack bandwidth is scarce compared to intra-rack bandwidth. This is especially true in the Shuffle phase of MapReduce. Since the communication pattern among the nodes is all-to-all in the Shuffle phase, the core switch is heavily utilized during this phase, while top-of-the-rack switches are under-utilized.

Based on this observation, we employ rack-level replication, where replicas of an intermediate data block are always placed among the machines in the same rack. The advantage of this approach is that it lowers data transfer going through the bandwidth-scarce core switch. This reduces the chance of interfering with foreground dataflow computations. The disadvantage is that it cannot tolerate rack failures, because if a whole rack fails, all replicas get lost. However this turns out to be an issue of low concern because in practice, rack failures are rare, e.g., Google reports it only experiences roughly 20 rack failures per year [11]. Thus, the MTTF (Mean Time To Failure) of rack failures is considerably longer than the lifetime of intermediate data.

Selective Replication.

As discussed in Section 3.2, the problem of cascaded re-execution is caused by the loss of intermediate data. More precisely, the exact cause is the loss of intermediate data that is *consumed locally*.

Figure 10 illustrates this point. If there is a machine failure at the Map phase (e.g., stage 3), affected Map tasks can be restarted immediately if we replicated the intermediate data locally generated by the previous Reduce tasks on the same machine (indicated by downward arrows). This observation is generally applicable to any stage, for example,

between Map and Reduce stages. The reason is two-fold. First, other pieces of intermediate data generated outside of the failed machine can be re-fetched any time. This assumes that the intermediate data is not removed during the execution of a job – which is true for the current Hadoop implementation. Second, the Shuffle process naturally replicates the intermediate data between Map and Reduce stages except the intermediate data consumed locally. We can observe this from Figure 10 between Stage 1 and Stage 2, where the Shuffle process transfers most of the intermediate data from one machine to a different machine (indicated by downward arrows).

Thus, we choose to benefit from this observation by only replicating locally-consumed data. This can result in significant savings while replicating Map outputs. For instance, if k machines are utilized for Reduce, only $1/k$-th of the total Map outputs is consumed locally. This reduces the total amount of replicated data significantly, and can be replicated quickly.

Particularly in the Map stage, the amount of intermediate data that needs to be replicated can be reduced significantly this way. However, since Reduce outputs are almost always locally consumed, this technique will be of little help in reducing the replication interference of Reduce data.

Qualitatively, these three hypotheses appear to help reducing the interference. However, the question is not *if* these techniques will help, but *how much*. Thus, we experimentally study how much each technique can help in reducing the interference in Section 7.

6. ISS DESIGN

ISS (Intermediate Storage System) incorporates three replication techniques discussed in Section 5 in order to minimize performance interference. In this section, we provide an overview of the design of ISS.

6.1 Interface and Design Choices

ISS is implemented as an extension to HDFS. Concretely, we add an implementation of a rack-level replication mechanism that asynchronously replicates locally-consumed data (i.e., all three techniques from Section 5). ISS takes care of all aspects of managing intermediate data, including writing, reading, replicating, and shuffling. Thus, a dataflow programming framework that uses ISS does not need to perform the Shuffle phase at all - ISS replaces the Shuffle phase. ISS seamlessly transfers the intermediate data from writers (e.g., Map tasks) to readers (e.g., Reduce tasks). ISS extends the API of HDFS as we summarize in Table 2.

We discuss next some important design choices in ISS. These design decisions are tailored toward programming convenience for dataflow programming frameworks. Our design choices are:

1) All files are immutable. Once a file is created and closed, it can only be opened for read operations. Thus, the protocol for a writer is create-write-close, and the protocol for a reader is open-read-close.
2) The file becomes visible immediately after it is created.
3) To prevent replicas from diverging, there can be only one writer at a time, but multiple readers are allowed. Restricting the number of writers is not a problem for ISS because the access pattern of intermediate data is a single writer and a single reader.
4) When a reader opens a file, the reader is blocked until the file is completely written by the writer of that file and transferred to the reader's local disk. The reader only waits for the first copy is completely written, as ISS replicates asynchronously.

6.2 Hadoop Example

To illustrate how a dataflow programming framework can utilize ISS, consider Hadoop as an example. Each Map task creates intermediate files using iss_create(). The Hadoop's master node, which is in charge of scheduling and coordinating MapReduce tasks, keeps track of all the file names that Map tasks create. Since this is similar to the Shuffle phase in the current Hadoop, our approach is not a significant departure from the current Hadoop. However, we benefit from this approach because the Shuffle phase is automatically performed by ISS. Thus, we can reduce the overhead and complexity of the Shuffle process from the master. Each Map task then proceeds with iss_write() to write the intermediate files. In the meantime, each Reduce Task learns all the file names through the master, then uses iss_open(); this might wait until the files are completely copied to the local disk. After fetching the files, the Reduce task proceeds with iss_read() which operates on a local version of the file.

6.3 Implementation

We have implemented three techniques in Hadoop and HDFS. In order to implement rack-level replication as well as selective replication, we have utilized extra information available in Hadoop. First, Hadoop allows each node to report its rack by manual configuration. We utilize this information for rack-level replication. Second, locally-consumed intermediate data between Map and Reduce stages can be identified using two pieces of information. The first is related to the question of which Reduce task accesses which part of the intermediate data. This can be answered easily in Hadoop, since each Reduce task processes one contiguous partition in the intermediate key space, and this partitioning is known to Map tasks. Thus, in ISS, each Map task divides its output into partitions (one partition for each Reduce task), and writes one file per partition. Each Reduce task only reads the partition (i.e., the file) that it is supposed to process. The second piece of information is the location of each Reduce task. This information is available in ISS, since we keep track of the locations of all writers and readers.

7. EVALUATION

In this section, we experimentally evaluate the effectiveness of ISS, and thus the techniques of Section 5. Our main metric is job completion time, but we also evaluate performance characteristics such as behavior under failures and replication completion time. We evaluate each technique separately first. Then we evaluate how all three techniques work together in various scenarios, with and without failures. All our experiments are performed under three settings in Emulab described in Table 1.

7.1 Methodology

The workload we use for all experiments is sorting using `Sort` in Hadoop. The input data is generated by `RandomWriter` in Hadoop that generates random (key, value) pairs. As shown in Table 1, the input size is 2GB per node

Name	Description
int iss_create(String pathName)	Creates a new intermediate file
int iss_open(String pathName, int off, int len)	Opens an existing intermediate file
int iss_write(int fd, char[] buf, int off, int len)	Writes the content of a buffer to an intermediate file at the given offset, and replicates it in the same rack
int iss_read(int fd, char[] buf, int off, int len)	Reads the content of an intermediate file to a buffer from the given offset
int iss_close(int fd)	Closes an intermediate file

Table 2: API Extension for ISS (Written in POSIX-Style)

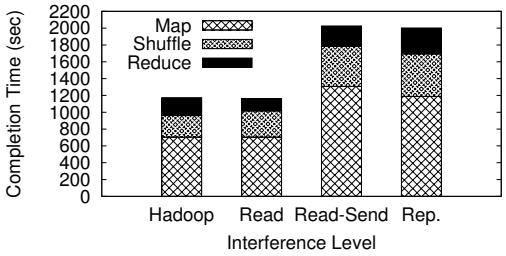

Figure 11: Asynchronous Replication with HDFS (S-Unif)

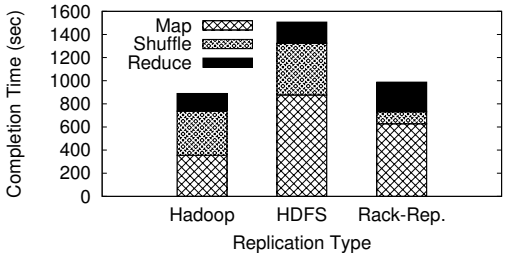

Figure 12: Rack-Level Replication (M-Het)

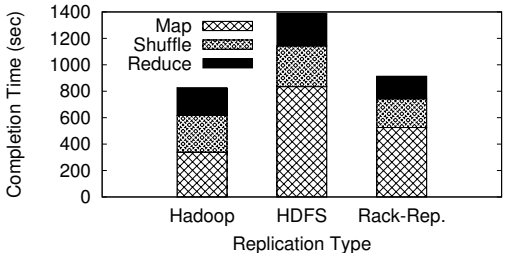

Figure 13: Rack-Level Replication (M-Unif)

and uniformly distributed over all nodes. All bar plots except Figure 19 show the results averaged over 5 runs. Each plot for failure injection experiments (including Figure 19) shows the result of one run. All experiments have been done on Emulab with its PC3000 machines.

7.2 Asynchronous Replication

HDFS replication works synchronously and reports data as stored only when replication is complete. This leads to Map tasks blocking for the HDFS replication to complete. Hence, we modify HDFS to make the replication asynchronous.

Figure 11 shows average completion times of MapReduce under four experimental settings. The purpose of this experiment is (1) to examine the performance of asynchronous replication, and (2) to understand where the sources of interference lie. Thus, the four bars are presented in the order of increasing degree of interference.

In the left-most experiment (labeled as *Hadoop*), we use the original Hadoop that does not replicate the intermediate data, hence there is no interference due to replication. In the right-most experiment (labeled as *Rep.*), we use the modified version of HDFS to asynchronously replicate the intermediate data to a remote node and store it on the remote disk. Although the degree of interference is less than synchronous replication, performance still degrades to the point where job completion time takes considerably longer.

The middle two experiments help to show the source of the performance hit by breaking down HDFS replication into its individual operations. In the second experiment (labeled as *Read*), we take only the first step of replication, which is to read the Map output from the local disk. In the next experiment (labeled as *Read-Send*), we use the modified version of HDFS to asynchronously replicate the intermediate data over the network, but *without* physically writing at the remote disk. This involves a local disk read and a network transfer, but no disk writes.

When we only read the intermediate data, there is hardly any difference in the overall completion time (*Hadoop* vs. *Read*). However, when the replication process starts using the network (*Read* vs. *Read-Send*), there is a significant overhead that results in doubling the completion time. This is primarily due to the increase in the Shuffle phase's execution time. [4] The increase in the Map finish time in *Read-Send* is also due to the network interference, since some Maps need to fetch their inputs from remote nodes. Finally, we notice that the interference of disk writes is very low (*Read-Send* vs. *Rep.*). In summary, disk reads and writes are not bottlenecks, but the network interference is. This motivates our next two techniques.

7.3 Rack-Level Replication

By default, HDFS places the first copy on the local machine, and the next copy on a machine located in a remote rack. However, as we reasoned in Section 5, there is potentially higher bandwidth availability within a rack. Thus, we need to quantify how much reduction of interference we can achieve by replicating within a rack.

Figures 12 and 13 show the results for two different settings. We have actually performed our experiments in a few more settings in order to quantify the benefit of rack-level replication in various scenarios, but the results were similar. [5] The left-most bar (labeled as *Hadoop*) shows the result with the original Hadoop. It does not perform any replication of the intermediate data (the Map outputs). The middle bar (labeled as *HDFS*) uses HDFS to replicate the interme-

[4]The plot shows the finish time of each phase, but does not show the initial start time for Shuffle and Reduce; the phases in fact overlap as seen in Figure 4 and 5.

[5]More specifically, we have used 20 machines spread over 4 LANs, 60 machines in one LAN, and 80 machines in one LAN, with various bandwidth combinations using 100Mbps and 1Gbps switches. This is a set of experiments beyond Table 1.

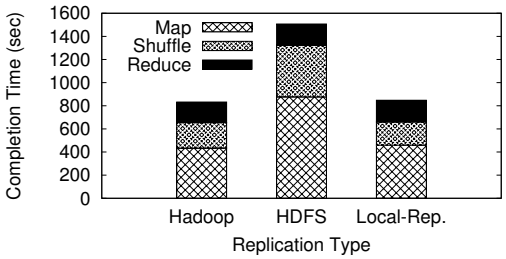

Figure 14: Locally-Consumed Data Replication (M-Het)

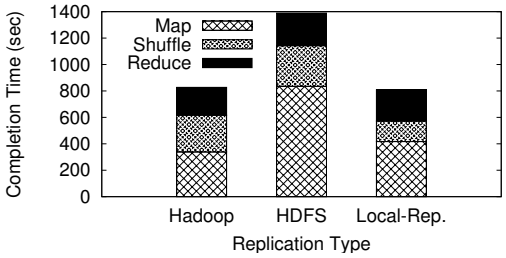

Figure 15: Locally-Consumed Data Replication (M-Unif)

diate data with the replication degree of 2. The right-most bar (labeled as *Rack-Rep.*) replicates the intermediate data within the same rack with a replication degree of 2.

These plots show that we can achieve significant reduction in network interference just by replicating within the same rack. The actual completion time varies depending on the configuration, but the increase only ranges from 70 seconds to 100 seconds. In contrast, when using HDFS for replication, completion times increase nearly twice as long.

As discussed in Section 5 and Section 7.2, this reduction is possible because 1) the network is the main source of interference, and 2) often there is idle bandwidth within each rack.

7.4 Locally-Consumed Data

The third possibility for reducing the interference is to only replicate locally-consumed data. As discussed in Section 5, this is possible because the Shuffle phase transfers intermediate data in an all-to-all manner, which leads to natural replication.

Figures 14 and 15 show the results. Both plots show that there is very little overhead when we replicate the locally-consumed data. This is due to the amount of data that needs to be replicated. If we replicate only the locally-consumed data, the amount of replicated data is reduced by $\frac{1}{k}$ assuming uniform partitioning, where k is the number of total machines.

7.5 Performance of ISS Under Failure

As discussed in Section 3.2, the original Hadoop re-executes Map tasks when it encounters failures in the Reduce phase. This is illustrated in Figure 16.

We show the results with two types of failures in five failure scenarios. The first failure type is a permanent machine failure, where a machine is killed during an experiment and it never gets back up. The second failure type is a transient failure, where a machine is killed, but is rebooted and re-registered with the Hadoop master node as an available machine after one minute. In all our experiments, each failure was injected immediately after the Shuffle phase was over.

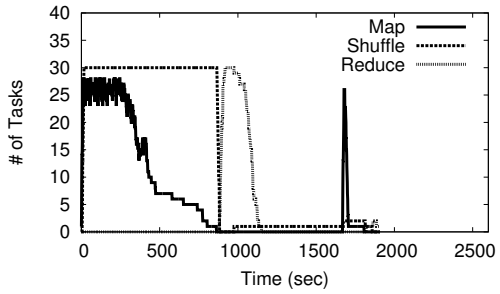

Figure 16: Hadoop (HDFS Replication=1) with One Machine Failure Injected (S-Unif)

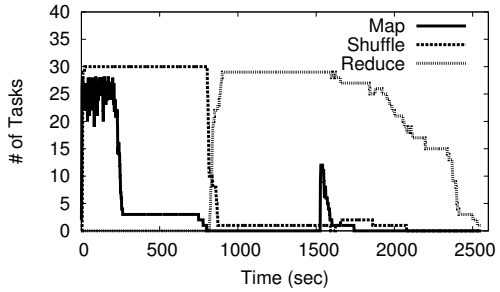

Figure 17: Hadoop (HDFS Replication=2) with One Machine Failure Injected (S-Unif)

The five scenarios we studied are the following: 1) one permanent machine failure with the original Hadoop using HDFS with the Reduce output replication degree of 1 (shown in Figure 16), 2) one permanent machine failure with the original Hadoop using HDFS with the Reduce output replication degree of 2 (shown in Figure 17), 3) one permanent machine failure with Hadoop augmented with ISS to replicate both the Map and Reduce outputs with the degree of 2 (shown in Figure 18), 4) one transient failure with the original Hadoop using HDFS with the Reduce output replication degree of 1 (shown in Figure 20), and 5) one transient failure with Hadoop augmented with ISS to replicate both the Map and Reduce outputs with degree of 2 (shown in Figure 21).

In Figure 16, a machine failure is injected at around time $t = 900$ at the beginning of the Reduce phase. This failure goes undetected until the failed machine's heartbeat timeout expires (the default timeout value is 10 minutes). Thus, at around time $t = 1600$, we can see that Map tasks are re-assigned to a different machine and re-executed, shown by the re-surge of the bold line. In Figure 17, we can observe a similar behavior of Hadoop with a significantly longer job completion time. As discussed in Section 4.1, the replication overhead of HDFS is the main cause of this difference.

This behavior would be different if the intermediate data generated by the failed node were available at some other node. The effect of failure would be masked because of "speculative execution" already implemented in Hadoop. In a nutshell, speculative execution detects any task that is making slow progress compared to other tasks, and redundantly executes the detected slow task on a faster machine. Thus, if the intermediate data is available, speculative execution can identify tasks that were running on the failed machine as slow tasks and redundantly execute them on different machines. However, the speculative execution does not help if the intermediate data is not available as in Fig-

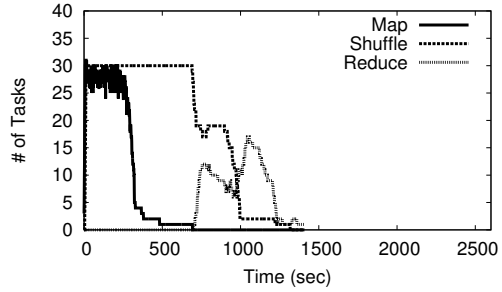

Figure 18: Hadoop Augmented with ISS with One Machine Failure Injected (S-Unif)

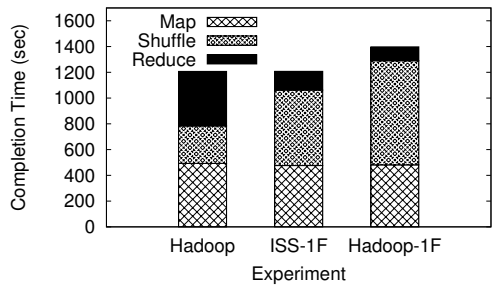

Figure 19: Performance Comparison with a Machine Failure (M-Unif)

ure 16, since not even speculated tasks can execute when intermediate data is lost.

Figure 18 demonstrates the performance of ISS under a permanent machine failure. A machine failure is injected at around time $t = 800$. Comparing Figure 18 to Figure 16, using ISS reduces the completion time to 1414 seconds from 1908 seconds, showing approximately 26% speedup. Comparing Figure 18 to Figure 17, using ISS reduces the completion time to 1414 seconds from 2553 seconds, showing 45% speedup. Compared to the average completion time of Hadoop without any failure shown in Figure 6 (the left most bar), the completion time of Hadoop with ISS under one failure increases only approximately 18% in Figure 18. However, the completion time of Hadoop without ISS under one failure in Figure 16 increases approximately 59%, which is quite significant.

Figure 19 shows the performance of ISS in the M-Unif setting. The left most bar (labeled as *Hadoop*) shows the job completion time without any failure, while both the middle bar (labeled as *ISS-1F*) and the right most bar (labeled as *Hadoop-1F*) show the results with one permanent machine failure. Unlike other bar plots, this plot only shows the results of one run for each experiment over the same input. Nonetheless, this particular case shows that ISS (*Hadoop vs. ISS-1F*) is capable of masking the effect of a permanent machine failure almost completely.

Figure 20 and Figure 21 show the results with a transient failure. In the current implementation of Hadoop, if a machine crashes and reboots, it registers itself as a new available node with the Hadoop master. Thus, even if a job execution encounters a transient failure, it is not able to recover from it; all Map and Reduce tasks have to be re-started just as the case for permanent failures. Figure 20 shows that even with a transient failure, Hadoop waits until it detects the failure and re-executes affected Map tasks and Reduce tasks at around time $t = 1400$. With ISS, speculative execution still kicks in no matter how the rebooted

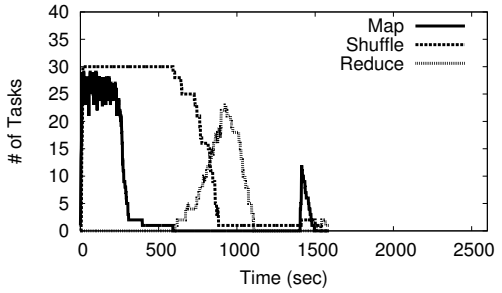

Figure 20: Hadoop (HDFS Replication=1) with One Transient Failure Injected (S-Unif)

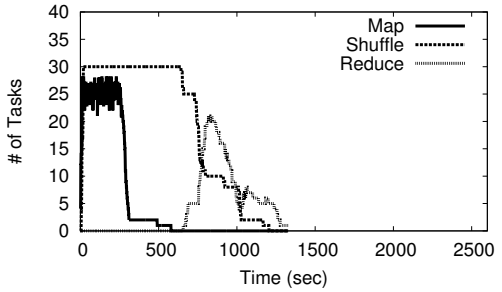

Figure 21: Hadoop Augmented with ISS with One Transient Failure Injected (S-Unif)

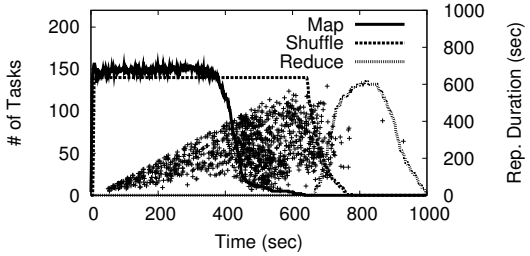

Figure 22: Replication Completion Time (M-Het)

node behaves. In fact, since the total number of available machines is the same after recovering from the transient failure, there is little difference in performance. For example, the slowdown from the original Hadoop without any failure shown in Figure 6 to Hadoop augmented with ISS with one transient failure shown in Figure 21 is only about 9%.

7.6 Replication Completion Time

Replication completion time is an important metric since it shows the "window of vulnerability", i.e., the period of time during which ISS cannot provide data availability for an intermediate data block (if a failure happens during this period, Hadoop has to resort back to its re-execution to regenerate the lost intermediate blocks). This is shown in Figures 22 and 23. In both figures, we plot the time taken by each block (size: 128MB) to be completely replicated (shown by crosses) along with the timeline of a foreground MapReduce job (shown by lines). We have chosen to show the performance of asynchronous rack-level replication mechanism that replicates all the intermediate data generated by the Map tasks, in order to picture the complete process of replication. We show one specific run for each of two settings: M-Het (Figure 22) and M-Unif (Figure 23).

We observe a general trend that the replication time takes longer for each block towards the end of the Shuffle phase. This is due to lower bandwidth availability, since the net-

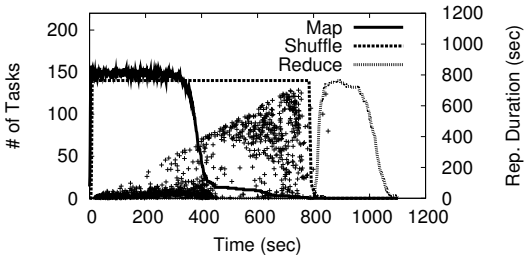

Figure 23: Replication Completion Time (M-Unif)

work is heavily utilized during the Shuffle phase. In Figure 22, the last replicated block finishes at around time $t = 950$, shortly before the Reduce phase ends.

In contrast, in Figure 23, the last replicated block finishes at around time $t = 850$, shortly after the Shuffle phase ends. This difference is due to the bandwidth available within a rack. M-Het uses 100 Mbps switches, while M-Unif uses 1Gbps switches. However, we can see that even in a bandwidth-scarce environment such as M-Het, the replication finishes before the Reduce phase.

Figure 22 and 23 also show why ISS minimizes interferences – replication times are short early on when Map tasks are executing because there is much bandwidth available (time $4 = 0$ to $t = 400$). However, once the Shuffle phase starts to fully operate ($t = 400$ to $t = 700$), ISS replication slows down thus minimizing interference with the foreground flows that is critical to job completion time.

8. RELATED WORK

Storage and File Systems Many traditional distributed storage and file systems are often used in cloud computing, e.g., NFS [30] and AFS [23], due to the historical widespread usage and the familiar POSIX interface these systems expose. Recently, storage systems designed for large-scale data centers has been proposed, e.g., Dynamo [14], PNUTS [10], BigTable [7], GFS [17], HDFS [21], Sinfonia [2], HBase [20], Cassandra [6], Boxwood [28], Ursa Minor [1], DDS [18], Chubby [5], etc. These systems have explored different design trade-offs such as data abstraction, consistency, availability, and read/write performance, However, there has been little work on designing storage systems for intermediate data.

Optimistic Replication Saito et al. give a comprehensive overview of optimistic replication [34]. Optimistic replication provides higher availability and performance, while sacrificing strong consistency guarantees. Systems that employ optimistic replication allow inconsistent replicas temporarily [35, 33]. The use of optimistic replications in these systems are usually motivated by the challenges in their running environments, e.g., mobile computing, wide-area data replication, etc., where bandwidth and latency of the underlying network prohibits providing both acceptable performance and good consistency. Although our asynchronous replication technique belongs to this class of replication techniques, it is motivated by performance interference. Traditional file/storage systems generally achieve single-copy consistency [22, 4], i.e., replicas are only accessible when all updates are completely written. This single-copy consistency is what the current HDFS achieves with its synchronized replication.

Interference Minimization Interference minimization has been studied in different layers of computer systems. A few transport layer protocols have been proposed that minimize interference among flows, e.g., TCP-Nice and TCP-LP [36, 26]. These protocols do not require any QoS support from the network, and yet minimize interference between background flows and foreground flows as we discussed in Section 4.2. TFS (Transparent File System) is a file system designed to support contributory storage applications, e.g., P2P file sharing applications [8]. TFS allows these applications to contribute as much storage space as possible without interfering with other applications that use the file system. SGuard is a distributed stream processing engine that checkpoints the state of stream processing nodes [27]. SGuard minimizes the interference of this checkpointing process to the foreground stream processing.

9. CONCLUSION

We have shown the need for, presented requirements towards, and designed a new intermediate storage system (ISS) that treats intermediate data in dataflow programs as a first-class citizen in order to tolerate failures. We have shown experimentally that the existing approaches are insufficient in satisfying the requirements of data availability and minimal interference. We have also shown that our asynchronous rack-level selective replication mechanism is effective, and masks interference very well. Our failure injection experiments show that Hadoop with ISS can speed up the performance by approximately 45% compared to Hadoop without ISS. Under no failures, ISS incurs very little overhead compared to Hadoop. Under a failure, ISS incurs only up to 18% of overhead compared to Hadoop with no failures.

10. ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their insightful feedback on this paper.

11. REFERENCES

[1] M. Abd-El-Malek, W. V. C. II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: Versatile Cluster-based Storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '05)*, 2005.

[2] M. K. Aguilera, A. Merchant, M. A. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *Proceedings of the ACM Symposium on Operating systems principles (SOSP)*, 2007.

[3] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. *SIGOPS Operating Systems Review (OSR)*, 25(5), 1991.

[4] P. A. Bernstein and N. Goodman. The failure and recovery problem for replicated databases. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, 1983.

[5] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[6] Cassandra: A Structured Storage System on a P2P Network. http://code.google.com/p/the-cassandraproject.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[8] J. Cipar, M. D. Corner, and E. D. Berger. TFS: a Transparent File System for Contributory Storage. In *Proceedings of the 5th USENIX conference on File and Storage Technologies (FAST '07)*, 2007.

[9] Cisco. Cisco Data Center Infrastructure 2.5 Design Guide. http://www.cisco.com/en/US/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCI_SRND.pdf.

[10] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2008.

[11] J. Dean. Software Engineering Advice from Building Large-Scale Distributed Systems. http://research.google.com/people/jeff/stanford-295-talk.pdf.

[12] J. Dean. Experiences with MapReduce, an Abstraction for Large-Scale Computation. In *Keynote I: PACT*, 2006.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[14] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the ACM Symposium on Operating systems principles (SOSP)*, 2007.

[15] Dryad Project Page at MSR. http://research.microsoft.com/en-us/projects/Dryad/.

[16] Facebook. Hive. http://hadoop.apache.org/hive/.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating systems principles (SOSP)*, 2003.

[18] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th USENIX Symposium on Opearting Systems Design and Implementation (OSDI '00)*, 2000.

[19] Hadoop Presentations. http://wiki.apache.org/hadoop/HadoopPresentations.

[20] HBase. http://hadoop.apache.org/hbase.

[21] HDFS (Hadoop Distributed File System). http://hadoop.apache.org/core/docs/r0.20.0/hdfs_user_guide.html.

[22] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[23] J. H. Howard. An Overview of the Andrew File System. In *Proceedings of the Annual USENIX Winter Technical Conference*, 1988.

[24] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs From Sequential Building Blocks. In *Proceedings of the 2007 EuroSys Conference (EuroSys)*, 2007.

[25] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. On Availability of Intermediate Data in Cloud Computations. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.

[26] A. Kuzmanovic and E. W. Knightly. TCP-LP: Low-Priority Service via End-Point Congestion Control. *IEEE/ACM Transactions on Networking*, 14(4):739–752, 2006.

[27] Y. Kwon, M. Balazinska, and A. Greenberg. Fault-tolerant Stream Processing using a Distributed, Replicated File System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB '08)*, 2008.

[28] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proceedings of the 6th USENIX Symposium on Opearting Systems Design and Implementation (OSDI '04)*, 2004.

[29] Sorting 1PB with MapReduce. http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.

[30] Network File System (NFS) version 4 Protocol. http://tools.ietf.org/html/rfc3530.

[31] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD)*, 2008.

[32] Powered by Hadoop. http://wiki.apache.org/hadoop/PoweredBy.

[33] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming Aggressive Replication in the Pangaea Wide-Area File System. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[34] Y. Saito and M. Shapiro. Optimistic Replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[35] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the fifteenth ACM symposium on Operating systems principles (SOSP '95)*, 1995.

[36] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A Mechanism for Background Transfers. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[37] W. Vogels. File System Usage in Windows NT 4.0. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP)*, 1999.