MOON: MapReduce On Opportunistic eNvironments

Heshan Lin
Virginia Tech
hlin2@cs.vt.edu

Xiaosong Ma
North Carolina State U.
Oak Ridge Nat'l Lab
ma@cs.ncsu.edu

Jeremy Archuleta
Virginia Tech
jsarch@cs.vt.edu

Wu-chun Feng
Virginia Tech
feng@cs.vt.edu

Mark Gardner
Virginia Tech
mkg@vt.edu

Zhe Zhang
Oak Ridge Nat'l Lab
zhezhang@ornl.gov

ABSTRACT

MapReduce offers an ease-of-use programming paradigm for processing large data sets, making it an attractive model for distributed volunteer computing systems. However, unlike on dedicated resources, where MapReduce has mostly been deployed, such volunteer computing systems have significantly higher rates of node unavailability. Furthermore, nodes are not fully controlled by the MapReduce framework. Consequently, we found the data and task replication scheme adopted by existing MapReduce implementations woefully inadequate for resources with high unavailability.

To address this, we propose MOON, short for *MapReduce On Opportunistic eNvironments*. MOON extends Hadoop, an open-source implementation of MapReduce, with adaptive task and data scheduling algorithms in order to offer reliable MapReduce services on a hybrid resource architecture, where volunteer computing systems are supplemented by a small set of dedicated nodes. Our tests on an emulated volunteer computing system, which uses a 60-node cluster where each node possesses a similar hardware configuration to a typical computer in a student lab, demonstrate that MOON can deliver a *three-fold* performance improvement to Hadoop in volatile, volunteer computing environments.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*Distributed systems*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Fault tolerance*

General Terms

Performance, Reliability

Keywords

MapReduce, Cloud Computing, Volunteer Computing

1. INTRODUCTION

With the advent in high-throughput scientific instruments as well as Internet-enabled collaboration and data sharing, rapid data growth has been observed in many domains, scientific and commercial alike. Processing vast amounts of data requires computational power far beyond the capability of individual workstations. Despite the success of commodity clusters, such platforms are still expensive to acquire and maintain for many institutions, necessitating affordable parallel computing.

Cloud computing continues to increase in its popularity in the scientific computing community because of its flexibility and potential in reducing the investment in HPC infrastructures. However, two major concerns need to be addressed before *public cloud* computing becomes mainstream. First, paying to run on commercial clouds has not yet been accommodated in current computing resource funding and support models. Second, the cost of data movement to and from the clouds over the wide-area network can be expensive and time consuming. In the meantime, the maturation of volunteer computing systems [22, 3, 6, 4] offers a low-cost alternative for building *private clouds* within institutions. However, those volunteer computing systems are traditionally built for CPU-intensive, embarrassingly parallel workloads. With heterogeneous nodes in a come-and-go nature, standard parallel programming models such as MPI do not work well on volunteer computing systems.

The emergence and growing popularity of MapReduce [8] may bring a change to the volunteer computing landscape. MapReduce is a popular programming model for cloud computing, which simplifies large-scale parallel data processing. Its flexibility in work distribution, loosely synchronized computation, and tolerance for heterogeneity are ideal features for opportunistically harnessed volunteer computing resources. While this union is conceptually appealing, a vital issue needs to be addressed – computing resources in volunteer computing systems are significantly more volatile than in dedicated computing environments, where MapReduce has mostly been deployed so far.

For example, while Ask.com per-server unavailability rate is an astonishingly low 0.000455 [25], availability traces collected from an enterprise volunteer computing system [18] showed a very different picture: individual node unavailability rates average around 0.4 with as many as 90% of the resources simultaneously inaccessible (Figure 1). Unlike dedicated systems, software/hardware failure is not the major contributor to resource volatility on volunteer computing systems. Volunteer computing nodes can be shut down at the owners' will. Also, typical volunteer computing frameworks such as Condor [22] consider a computer unavailable for external jobs whenever keyboard or mouse events are detected. In such a volatile environment, it is unclear how well existing MapReduce frameworks perform.

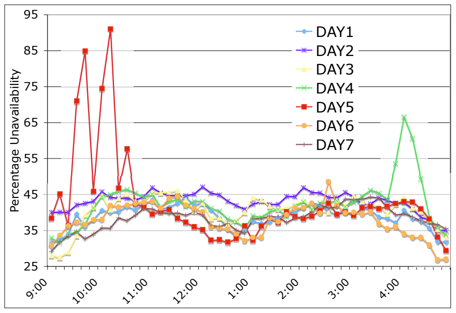


Figure 1: Percentage of unavailable resources measured on a production volunteer computing system.

In this work, we first evaluated Hadoop, a popular, open-source MapReduce run-time system [1], on an emulated volunteer computing system and observed that the volatility of opportunistic resources creates several severe problems. First, the Hadoop Distributed File System (HDFS) provides reliable data storage through replication, which on volatile systems can have a prohibitively high replication cost in order to provide high data availability. For instance, when the machine unavailability rate is 0.4, *eleven replicas* are needed to achieve 99.99% availability for a single data block, assuming that machine unavailability is independent[1]. Handling large-scale *correlated* resource unavailability requires even more replication.

Second, Hadoop does *not* replicate intermediate results, i.e., the output of Map tasks. When a node becomes inaccessible, the Reduce tasks processing intermediate results on this node will stall, resulting in Map task re-execution or even livelock.

Third, Hadoop task scheduling assumes that the majority of the tasks will run smoothly until completion. However, tasks can be frequently suspended or interrupted on volunteer computing systems. The default Hadoop task replication strategy, designed to handle failures, is insufficient to handle the high volatility of volunteer computing platforms.

To mitigate these problems in order to realize the computing potential of MapReduce on volunteer computing systems, we have a created a novel amalgamation of these two technologies to produce MOON— MapReduce On Opportunistic eNvironments. MOON adopts a hybrid resource architecture by provisioning a small set of dedicated, reliable computers to supplement the volatile personal computers. Leveraging such a hybrid architecture, MOON then extends Hadoop's task and data scheduling to greatly improve the QoS of MapReduce services. Together with detailed descriptions of MOON, we present extensive evaluation of its design on an emulated volunteer computing system. Our results show that MOON can deliver as much as a 3-fold speedup to Hadoop, and even finish MapReduce jobs that could not be completed previously in highly volatile environments.

[1]The availability of 11 replicas is $1 - 0.4^{11} = 0.99996$.

2. BACKGROUND

2.1 Volunteer Computing

Many volunteer computing systems have been developed to harness idle desktop resources for high-throughput computing [3, 4, 6, 22]. A common feature shared by these systems is non-intrusive deployment. While studies have been conducted on aggressively stealing computer cycles [20] and its corresponding impact [14], most production volunteer computing systems allow users to donate their resources in a conservative way by not running external tasks when the machine is actively used. For instance, Condor allows jobs to execute only after 15 minutes of no console activity and a CPU utilization level lower than 0.3.

2.2 MapReduce

MapReduce is a programming model designed to simplify parallel data processing [8]. Google has been using MapReduce to handle massive amount of web search data on large-scale commodity clusters. This programming model has also been found effective in other application areas including machine learning [5], bioinformatics [19], astrophysics and cyber-security [12].

A MapReduce application is implemented through two user-supplied primitives: *Map* and *Reduce* . Map tasks take input *key-value pairs* and generate intermediate key-value pairs through certain user-defined computation. The intermediate results are subsequently converted to output key-value pairs in the reduce stage with user-defined reduction processing. Google's MapReduce production systems use its proprietary high-performance distributed file system, GFS [11], to store the input, intermediate, and output data.

2.3 Hadoop

Hadoop is an open-source cluster-based MapReduce implementation written in Java [1]. It is logically separated into two subsystems: the Hadoop Distributed File System (HDFS), and a master-worker MapReduce task execution framework.

HDFS consists of a *NameNode* process running on the master and multiple *DataNode* processes running on the workers. To provide scalable data access, the NameNode only manages the system *metadata*, whereas the actual file contents are stored on the DataNodes. Each file in the system is stored as a collection of equal-sized data blocks. For I/O operations, an HDFS client queries the NameNode for the data block locations, with subsequent data transfer occurring directly between the client and the target DataNodes. Like GFS, HDFS achieves high data availability and reliability through data replication, with the replication degree specified by a *replication factor* (3 by default).

To control task execution, a single *JobTracker* process running on the master manages job status and performs task scheduling. On each worker machine, a *TaskTracker* process tracks the available execution slots: a worker machine can execute up to M Map tasks and R Reduce tasks simultaneously (M and R set to 2 by default). A TaskTracker contacts the JobTracker for an assignment when it detects an empty execution slot on the machine. Tasks of different jobs are scheduled according to job priorities. Within a job, the JobTracker first tries to schedule a non-running task, giving high priority to the recently failed tasks, but if all tasks for this job have been scheduled, the JobTracker speculatively issues backup tasks for slow running ones. These speculative tasks help improve job response time.

3. MOON DESIGN RATIONALE AND ARCHITECTURE OVERVIEW

MOON targets institutional intranet environments, where volunteer personal computers (PCs) are connected with a local area network with relatively high bandwidth and low latency. However, PC availability is ephemeral in such environments. Moreover, large-scale, correlated unavailability can be normal [16]. For instance, many machines in a computer lab will be occupied simultaneously during a lab session. Similarly, an important cultural event may prompt employees to check news websites, resulting in a large percentage of PCs unavailable for volunteer computing.

Observing that opportunistic PC resources are not dependable enough to offer reliable compute and storage services, MOON supplements a volunteer computing system with *a small number of dedicated compute resources*. The MOON hybrid architecture has multiple advantages. *First*, placing a replica on dedicated nodes can significantly enhance data availability without imposing a high replication cost on the volatile nodes, thereby improving overall resource utilization and reducing job response time. For example, the well-maintained workstations in our research lab have had only 10 hours of unscheduled downtime in the past year which is equivalent to a 0.001 unavailability rate. Assuming the average unavailability rate of a volunteer computing system is 0.4 and the failure of each volatile node is independent, achieving 99.99% availability only requires one copy on the dedicated node and three copies on the volatile nodes[2]. *Second*, long-running tasks with execution times much longer than the Mean Time Between Failure (including temporary inaccessibility due to the owner's activities) of volunteered machines may be difficult to finish on purely volatile resources because of frequent interruptions. Scheduling those long-running tasks on dedicated resources can guarantee their completion. *Finally*, with those dedicated nodes, the system can function even when a large percentage of nodes are temporarily unavailable.

MOON is designed to run atop existing volunteer computing systems. For non-intrusive deployment, a MOON worker can be wrapped inside a virtual machine and distributed to each PC, as enabled by Condor [22] and Entropia [6]. Consequently, MOON assumes that no computation or communication progress can be made on a PC when it is actively used by the owner, and it relies on the *heartbeat mechanism* in Hadoop to detect the PC availability.

In addition, there are several major assumptions in the current MOON design:

- As will be discussed in Section 4, we assume that collectively, the dedicated nodes have enough aggregate storage for at least one copy of all active data in the system. We argue that this solution is made practical by the decreasing price of commodity servers and hard drives with large capacity.
- We assume that the security solutions of existing desktop grid systems (e.g., Condor) can be applied to the MOON system. Consequently, we do not directly address the security issue in this paper.
- For general applicability, we conservatively assume that the node unavailability cannot be known a priori. In the future, we will study how to leverage node-failure predictability to enhance scheduling decisions under certain environments.

It is worth noting that this paper aims at delivering a proof-of-concept study of the MOON hybrid design, as well as corresponding task scheduling and data management techniques. The efficacy of the proposed techniques is gauged with expansive performance evaluations. Our initial results shown in this paper demonstrate the merits of the hybrid design and the complex interaction between system parameters. This motivates automatic system configuration based on rigorous performance models, which is part of our immediate plan for future work. Please also note that MOON is designed to support general MapReduce applications and does not make assumptions on job characteristics.

4. MOON DATA MANAGEMENT

In this section, we present our enhancements to Hadoop to provide a reliable MapReduce service from the data management perspective. Within a MapReduce system there are three types of user data – input, intermediate, and output. Input data are processed by Map tasks to produce intermediate data, which are in turn consumed by Reduce tasks to create output data. The availability of each type of data has different QoS implications.

For input data, temporary inaccessibility will stall computation of corresponding Map tasks, whereas loss of the input data will cause the entire job to fail. Intermediate and output data, on the other hand, are more resilient to loss, as they can be reproduced by re-executing the Map and/or Reduce tasks involved. However, once a job has completed, lost output data is irrecoverable if the input data have been removed from the system. In this case, a user will have to re-stage the previously removed input data and re-issue the entire job, acting as if the input data was lost. In any of these scenarios, the completion of the MapReduce job can be substantially delayed.

Note that high *data durability* [7] alone is insufficient to provide high quality MapReduce services. When a desktop computer is reclaimed by its owner, job data stored on that computer still persists. However, a MapReduce job depending on those data will fail if the data is unavailable within a certain execution window of a job. As such, the MOON data management focuses on improving overall *data availability*.

As mentioned in Section 1, we found that existing Hadoop data management is insufficient to provide high QoS on volatile environments for two main reasons.

- The replication cost to provide the necessary level of data availability for input and output data in HDFS on volunteer computing systems is prohibitive when the volatility is high.
- Non-replicated intermediate data can easily become temporarily unavailable for a long period of time or permanently unavailable due to user activity or software/hardware failures on the worker node where the data is stored, thereby unnecessarily forcing re-execution of the relevant Map tasks.

[2]The availability of a data block with one dedicated replica and three volatile replicas is $1 - 0.4^3 \times 0.001 = 0.99994$.

To address these issues, MOON augments Hadoop data management in several ways to leverage the proposed hybrid resource architecture to offer a cost-effective and robust storage service.

4.1 Multi-dimensional, Cost-effective Replication Service

MOON provides a multi-dimensional, dynamic replication service to handle volatile volunteer computing environments as opposed to the static data replication in Hadoop. MOON manages two types of resources – *supplemental dedicated computers* and *volatile volunteer nodes*. The number of dedicated computers is much smaller than the number of volatile nodes for cost-effectiveness purposes. To support this hybrid scheme, MOON extends Hadoop's data management and defines two types of workers: *dedicated* DataNodes and *volatile* DataNodes. Accordingly, the *replication factor* of a file is defined by a pair $\{d, v\}$, where d and v specify the number of data replicas on the dedicated and volatile DataNodes, respectively. For well-maintained dedicated computers with low unavailability rates, d is recommended to be set as 1 for efficient utilization of dedicated resources.

Intuitively, since dedicated nodes have much higher availability than volatile nodes, placing replicas on dedicated DataNodes can significantly improve data availability and in turn minimize the replication cost on volatile nodes. Because of the limited aggregated network and I/O bandwidth on dedicated computers, the major challenge is maximizing the utilization of the dedicated resources to improve service quality while preventing the dedicated computers from becoming a system bottleneck. To this end, MOON's replication design differentiates between various data types at the file level and takes into account the load and volatility levels of the DataNodes.

MOON defines two types of files, i.e., *reliable* and *opportunistic*. Reliable files are used to store data that *cannot* be lost under any circumstances. One or more dedicated copies are always maintained for a reliable file so that it can tolerate outage of a large percentage of volatile nodes. MOON always stores input data and system data required by the job as reliable files. In contrast, *opportunistic files* store transient data that can tolerate a certain level of unavailability and may or may not have dedicated replicas. Intermediate data will always be stored as opportunistic files. On the other hand, output data will first be stored as opportunistic files while the Reduce tasks are completing, and once all are completed they are then converted to reliable files.

The separation of reliable files from opportunistic files is critical in controlling the load level of dedicated DataNodes. When MOON decides that all dedicated DataNodes are nearly saturated, an I/O request to replicate an opportunistic file on a dedicated DataNode will be declined (details described in Section 4.2). Additionally, by allowing output data to be first stored as opportunistic files enables MOON to dynamically direct write traffic towards or away from the dedicated DataNodes as necessary. Furthermore, only after all data blocks of the output file have reached its replication factor, will the job be marked as complete and the output file be made available to users.

To maximize the utilization of dedicated computers, MOON attempts to have dedicated replicas for opportunistic files when possible. When dedicated replicas cannot be maintained, the availability of the opportunistic file is subject to the volatility of volunteer PCs, possibly resulting in poor QoS due to forced re-execution of the related Map or Reduce tasks. While this issue can be addressed by using a high replication degree on volatile DataNodes, such a solution will inevitably incur high network and storage overhead.

MOON addresses this issue by adaptively changing the replication requirement to provide the desired QoS level. Specifically, consider a write request of an opportunistic file with replication factor $\{d, v\}$. If the dedicated replicas are rejected because the dedicated DataNodes are saturated, MOON will dynamically adjust v to v', where v' is chosen to guarantee that the file availability meets the user-specified availability level (e.g., 0.9) pursuant to the node unavailability rate p (i.e., $1 - p^{v'} > 0.9$). If p changes before a dedicated replica can be stored, v' will be recalculated accordingly. Also, no extra replication is needed if an opportunistic file already has a replication degree higher than v'. In the current implementation, p is estimated by having the NameNode monitor the faction of unavailable DataNodes during the past 1 minute. This can be replaced with more accurate/detailed predicting methods.

The rationale for adaptively changing the replication requirement is that when an opportunistic file has a dedicated copy, the availability of the file is high, thereby allowing MOON to decrease the replication degree on volatile DataNodes. Alternatively, MOON increases the volatile replication degree of a file as necessary to prevent forced task re-execution caused by unavailability of opportunistic data.

Similar to Hadoop, when any file in the system falls below its replication factor, this file will be put into a replication queue. The NameNode periodically checks this queue and issues replication requests giving higher priority to reliable files. With this replication mechanism, the dedicated replicas of an opportunistic file will eventually be achieved. What if the system is constantly overloaded with jobs with large amounts of output? While not being handled in the current MOON design, this scenario can be addresed by having the system stop scheduling new jobs from the queue after observing that a job is waiting too long for its output to be coverted to reliable files.

4.2 Prioritizing I/O Requests

When a large number of volatile nodes are supplemented with a much smaller number of dedicated nodes, providing scalable data access is challenging. As such, MOON prioritizes the I/O requests on the different resources. To alleviate read traffic on dedicated nodes, MOON factors in the node type in servicing a read request. Specifically, for files with replicas on both volatile and dedicated DataNodes, read requests from clients on volatile DataNodes will always try to fetch data from volatile replicas first. By doing so, the read requests from clients on the volatile DataNodes will only reach dedicated DataNodes when none of the volatile replicas are available.

When a write request occurs, MOON prioritizes I/O traffic to the dedicated DataNodes according to data vulnerability. A write request from a reliable file will always be satisfied on dedicated DataNodes. However, a write request from an opportunistic file will be declined if all dedicated DataNodes are close to saturation. As such, write requests for reliable files are fulfilled prior to those of opportunistic files when the dedicated DataNodes are fully loaded. This decision process is shown in Figure 2.

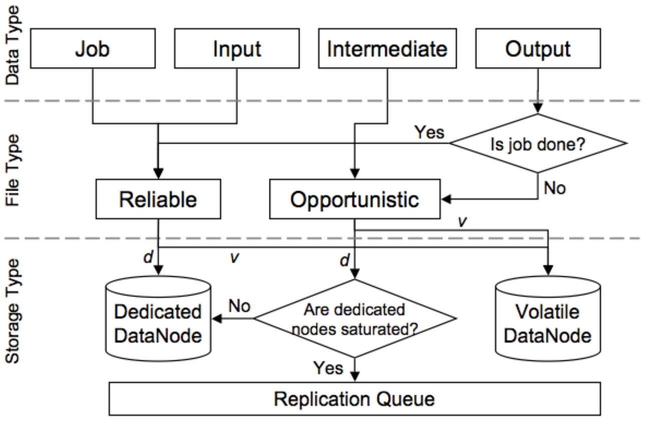


Figure 2: Decision process to determine where data should be stored.

To determine whether a dedicated DataNode is close to be saturated, MOON uses a sliding window-based algorithm as show in Algorithm 1. MOON monitors the I/O bandwidth consumed at each dedicated DataNode and sends this information to the NameNode piggybacking on the *heartbeat* messages. The throttling algorithm running on the NameNode compares the updated bandwidth with the average I/O bandwidth during a past window. If the consumed I/O bandwidth of a DataNode is increasing but only by a small margin determined by a threshold T_b, the DataNode is considered saturated. On the contrary, if the updated I/O bandwidth is decreasing and falls more than threshold T_b, the dedicated node is unsaturated. Such a design is to avoid unnecessary status switching caused by load oscillation. Since there is a delay between when the request is assigned and when the corresponding I/O-bandwidth increment is detected, MOON puts a cap (C_r) on the number of requests that can be assigned to a dedicated DataNode during a throttling window.

4.3 Handling Ephemeral Unavailability

Within the original HDFS, fault tolerance is achieved by periodically monitoring the health of each DataNode and replicating files as needed. If a heartbeat message from a DataNode has not arrived at the NameNode within the *NodeExpiryInterval* the DataNode will be declared dead and its files are replicated as needed.

This fault tolerance mechanism is problematic for opportunistic environments where transient resource unavailability is common. If the *NodeExpiryInterval* is shorter than the mean unavailability interval of the volatile nodes, these nodes may frequently switch between *live* and *dead* states, causing replication thrashing due to HDFS striving to keep the correct number of replicas. Such thrashing significantly wastes network and I/O resources and should be avoided. On the other hand, if the *NodeExpiryInterval* is set too long, the system would incorrectly consider a "dead" DataNode as "alive". These DataNodes will continue to be sent I/O requests until it is properly identified as dead, thereby degrading overall I/O performance as the clients experience timeouts trying to access the nodes.

To address this issue, MOON introduces a *hibernate* state. A DataNode enters the hibernate state if no heartbeat messages are received for more than a *NodeHibernateInterval*, which is much shorter than the *NodeExpiryInterval*. A hibernated DataNode will not be supplied any I/O requests so as to avoid unnecessary access attempts from clients. Observing that a data block with dedicated replicas already has the necessary availability to tolerate transient unavailability of volatile nodes, only opportunistic files without dedicated replicas will be re-replicated. This optimization can greatly save the replication traffic in the system while preventing task re-executions caused by the compromised availability of opportunistic files.

Algorithm 1 I/O throttling on dedicated DataNodes

```
Let $W$ be the throttling window size
Let $T_b$ be the control threshold
Let $bw_k$ be the measured bandwidth at timestep $k$
Input: current I/O bandwidth $bw_i$
Output: setting throttling state of the dedicated node

$avg\_bw = (\sum_{j=i-W}^{i-1} bw_j)/W$
if $bw_i > avg\_bw$ then
  if $(state == unthrottled)$ and $(bw_i < avg\_bw*(1+T_b))$ then
    $state = throttled$
  end if
end if
if $bw_i < avg\_bw$ then
  if $(state == throttled)$ and $(bw_i < avg\_bw * (1 - T_b))$ then
    $state = unthrottled$
  end if
end if
```

5. MOON TASK SCHEDULING

One important mechanism that Hadoop uses to improve job response time is to speculatively issue backup tasks for "stragglers", i.e. slow running tasks. Hadoop considers a task as a straggler if the task meets two conditions: 1) it has been running for more than one minute, and 2) its *progress score* lags behind the average progress of all tasks of the same type by 0.2 or more. The per-task progress score, valued between 0 and 1, is calculated as the fraction of data that has been processed in this task.

In Hadoop, all stragglers are treated equally regardless of the relative differences between their progress scores. The JobTracker (i.e., the master) simply selects stragglers for speculative execution according to the order in which they were originally scheduled, except that for Map stragglers, priority will be given to the ones with input data local to the requesting TaskTracker (i.e., the worker). The maximum number of speculative copies (excluding the original copy) for each task is user-configurable, but set at 1 by default.

Hadoop speculative task scheduling assumes that tasks run smoothly toward completion, except for a small fraction that may be affected by the abnormal nodes. Such an assumption is easily invalid in opportunistic environments; a large number of tasks will likely be suspended or interrupted due to temporary or permanent outages of the volatile nodes. For instance, in Condor, a running external job will be suspended when the mouse or keyboard events are detected. Consequently, identifying stragglers based solely on tasks' progress scores is too optimistic.

- First, when the machine unavailability rate is high, *all* instances of a task can possibly be suspended simultaneously, allowing no progress to be made on that task.

- Second, fast progressing tasks may be suddenly slowed down when a node becomes unavailable. Yet, it may take a long time for suspended tasks with high progress scores to be allowed to have speculative copies issued.
- Third, the natural computational heterogeneity among volunteered nodes, plus additional productivity variance caused by node unavailability, may cause Hadoop to issue a large number of speculative tasks (similar to the observation made in [24]), resulting in a waste of resources and an increase in job execution time.

Therefore, MOON adopts speculative task execution strategies that are *aggressive for individual tasks* to prepare for high node volatility, yet *overall conservative* considering the collectively unreliable environment. We describe these techniques in the rest of this section. Below we will describe our general-purpose scheduling in sections 5.1 and 5.2, and hybrid-architecture-specific augmentations in 5.3.

5.1 Ensuring Sufficient Progress with High Node Volatility

In order to guarantee that sufficient progress is made on all tasks, MOON characterizes stragglers into *frozen tasks* (tasks where *all* copies are simultaneously suspended) and *slow tasks* (tasks that are not frozen, but satisfy the Hadoop criteria for speculative execution). The MOON scheduler composes two separate lists, containing frozen and slow tasks respectively, with tasks selected from the frozen list first. In both lists, tasks are sorted by the progress made thus far, with lower progress ranked higher.

It is worth noting that Hadoop does offer a task fault-tolerant mechanism to handle node outage. The JobTracker considers a TaskTracker *dead* if no heartbeat messages have been received from the TaskTracker for an *TrackerExpiryInterval* (10 minutes by default). All task instances on a dead TaskTracker will be killed and rescheduled. Naively, using a small *TrackerExpiryInterval* can help detect and relaunch inactive tasks faster. However, using a too small value for the *TrackerExpiryInterval* will cause many suspended tasks to be killed prematurely, thus wasting resources.

In contrast, MOON considers a TaskTracker *suspended* if no heartbeat messages have been received from the TaskTracker for a *SuspensionInterval*, which can be set to a value much smaller than *TrackerExpiryInterval*, so that node anomaly can be detected early. All task instances running on a suspended TaskTracker are then flagged *inactive*, in turn triggering frozen task handling. Inactive task instances on such a TaskTracker are not killed right away, in the hope that it returns to normal shortly.

MOON imposes a cap on the number of speculative copies for a task similar to Hadoop. However, a speculative copy will be issued to a frozen task regardless of the number of its copies, so that progress can always be made for each task.

5.2 Two-phase Task Replication

The speculative scheduling approach discussed above only issues a backup copy for a task *after* it is detected as frozen or slow. Such a reactive approach is insufficient to handle fast progressing tasks that become suddenly inactive. For instance, consider a task that runs at a normal speed until 99% complete and then is suspended. A speculative copy will only be issued for this task after the task suspension is detected by the system, upon which the computation needs to be started all over again. To make it worse, the speculative copy may also become inactive before its completion. In the above scenario, the delay in the reactive scheduling approach can elongate the job response time, especially when this happens toward the end of the job.

To remedy this, MOON separates the job progress into two phases, *normal* and *homestretch*, where the *homestretch* phase begins once the number of remaining tasks for the job falls below $H\%$ of the currently available execution slots. The basic idea of this design is to alleviate the impacts of unexpected task interruptions by proactively replicating tasks toward the job completion. Specifically, during the homestretch phase, MOON attempts to maintain at least R *active* copies of *any remaining task* regardless its progress score. If the unavailability rate of volunteer PCs is p, the probability that a task will become frozen decreases to p^R.

The motivation of the two-phase scheduling stems from two observations. First, when the number of concurrent jobs in the system is small, computational resources become more underutilized as a job gets closer to completion. Second, a suspended task will delay the job more toward the completion of the job. To constrain the resources used by task replication, MOON also enforces a limit on the total concurrent speculative task instances for a job to $H\%$ of the available execution slots. No more speculative tasks will be issued if the concurrent number of speculative tasks of a job reaches that threshold. This means that the actual task replication degree gradually increases as the job approaches its completion.

While increasing R can reduce the probability of job freezing, it increases resource consumption. The optimal configuration of $H\%$ and R will depend on how users will want to trade-off resource consumptions and performance. We will evaluate various configurations of the two parameters in Section 6.

5.3 Leveraging Hybrid Resources

MOON attempts to further decrease the impact of volatility during *both* normal and homestretch phases by replicating tasks on the dedicated nodes. When the number of tasks in the system is smaller than the number of dedicated nodes, a task will be always be scheduled on dedicated nodes if there are empty slots available. Doing this allows us to take advantage of the much more reliable CPU resources available on the dedicated computers (as opposed to using them as pure data servers).

Intuitively, tasks with a dedicated speculative copy are given *lower* priority in receiving additional task replicas, as these nodes tend to be much more reliable. More specifically, when selecting a task from the slow task list as described in Section 5.1, the ones without a dedicated replica will be considered first. Similarly, tasks that already have a dedicated copy do not participate the homestretch phase, thus saving task replication cost. As another consequence, long running tasks that have difficulty in finishing on volunteer PCs because of frequent interruptions will eventually be scheduled and guaranteed completion on the dedicated nodes.

6. PERFORMANCE EVALUATION

On production volunteer computing systems, machine availability patterns are commonly non-repeatable, making it difficult to fairly compare different strategies. Meanwhile, traces cannot easily be manipulated to create different node availability levels. In our experiments, we emulate a volunteer computing system with synthetic node availability traces, where node availability level can be adjusted.

We assume that node outage is mutually independent and generate unavailable intervals using a normal distribution, with the mean node-outage interval (409 seconds) extracted from the aforementioned Entropia volunteer computing node trace [18]. The unavailable intervals are then inserted into 8-hour traces following a Poisson distribution such that in each trace, the percentage of unavailable time is equal to a given node unavailability rate. At runtime of each experiment, a monitoring process on each node reads in the assigned availability trace, and suspends and resumes all the Hadoop/MOON processes on the node accordingly [3].

Our experiments are executed on System X at Virginia Tech, comprised of Apple Xserve G5 compute nodes with dual 2.3GHz PowerPC 970FX processors, 4GB of RAM, 80 GByte hard drives. System X uses a 10Gbs InfiniBand network and a 1Gbs Ethernet for interconnection. To closely resemble volunteer computing systems, we only use the Ethernet network in our experiments. Arguably, such a machine configuration is similar to those in many student labs today. Each compute node runs the GNU/Linux operating system with kernel version 2.6.21.1. The MOON system is developed based on Hadoop 0.17.2.

Our experiments use two representative MapReduce applications, i.e., `sort` and `word count`, that are distributed with Hadoop. The configurations of the two applications are given in Table 1[4]. For both applications, the input data is randomly generated using tools distributed with Hadoop.

Table 1: Application configurations.

Application	Input Size	# Maps	# Reduces
`sort`	24 GB	384	$0.9 \times AvailSlots$
`word count`	20 GB	320	20

6.1 Speculative Task Scheduling Evaluation

We first evaluate the MOON scheduling design using two important job metrics: 1) job response time and 2) the total number of duplicated tasks issued. The job response time is important to user experiences. The second metric is important as extra tasks will consume system resources as well as energy. Ideally, we want to achieve short job response time with a low number of speculative tasks.

On opportunistic environments both the scheduling algorithm and the data management policy can largely impact the job response time. To isolate the impact of speculative task scheduling, we use the `sleep` application distributed with Hadoop, which allows us to simulate our two target applications with faithful Map and Reduce task execution times, but generating only insignificant amount of intermediate and output data (two integers per record of intermediate and zero output data).

[3]In our implementation, the task suspension and resume is achieved by sending the STOP and CONT signals to the targeting processes.

[4]These two applications with similar data input sizes were also used in other MapReduce studies, e.g., [24].

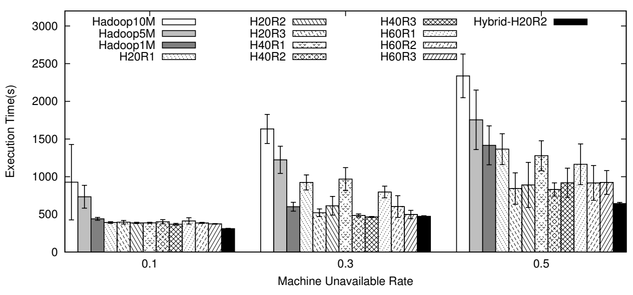
We feed the average Map and Reduce execution times from `sort` and `word count` benchmarking runs into `sleep`. We also configure MOON to replicate the intermediate data as reliable files with one dedicated and one volatile copy, so that intermediate data are always available to Reduce tasks. Since `sleep` only deals with a small amount of intermediate data, the impact of data management is minimal.

The test environment is configured with 60 volatile nodes and 6 dedicated nodes, resulting in a 10:1 of volatile-to-dedicated (V-to-D) node ratio (results with higher V-to-D node ratio will be shown in Section 6.3). We compare the original Hadoop task scheduling policy and the MOON scheduling algorithm described in Section 5. For the Hadoop default scheduling, we control how quickly it reacts to node outages by using 1, 5, and 10 (default) minutes for *TrackerExpiryInterval*. These polices are denoted as Hadoop1M, Hadoop5M and Hadoop10M, respectively. With even larger values of *TrackerExpiryInterval*, the Hadoop performance gets worse and hence those results are not shown here. For MOON, We use 1 minute for *SuspensionInterval*, and 10 minutes for *TrackerExpiryInterval* for a fair comparison. Recall from Section 5.2 that there are two parameters in MOON to control the aggressiveness of the two-phase scheduling: 1) the homestretch threshold $H\%$ and 2) the number of active copies R. To demonstrate the impacts of the selection of the two parameters, we vary H from 20 to 40 to 60. For each H value, R is increased from 1 to 3. Finally, we also test the enhancement with hybrid resource awareness (as described in Section 5.3) for $H = 20$ and $R = 2$.

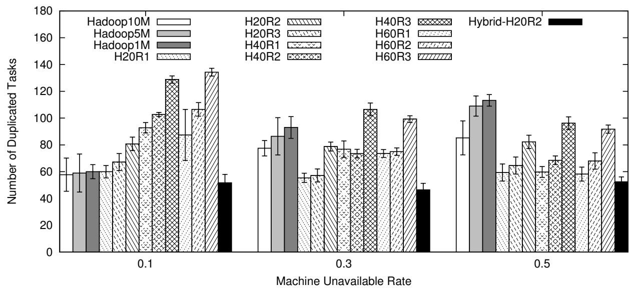
Figure 3(a) shows the execution time for the `sort` application with increasing node availability rates. For the Hadoop scheduling, it is clear that the job execution time reduces as *TrackerExpiryInterval* decreases. This is because with a shorter *TrackerExpiryInterval*, the JobTracker can detect the node outage sooner and issue speculative copies to the executing tasks on the unavailable nodes. In Hadoop, a TaskTracker is considered *dead* if no heartbeat messages have been sent from it within the *TrackerExpiryInterval*, and in turn, all running tasks on the TaskTracker will be killed and rescheduled. Consequently, the reducing in execution time by decreasing *TrackerExpiryInterval* will inevitably come at a cost of higher numbers of task replicas, as shown in Figure 3(b). Thus, the default Hadoop scheduling is not flexible to simultaneously achieving short job response time and a low quantity of speculative tasks.

With two-phase scheduling, the job response time is comparable among all configurations at 0.1 node unavailability rate. However, when the node unavailability rate gets higher, it is clear that increasing R from 1 to 2 can deliver considerable improvements for a same H value, due to the decreasing probability of a task being frozen toward the end of job execution. However, further increasing R to 3 does not help in most cases because the resource contention caused by the extra task replicas offsets the benefit of reducing task-suspension. In fact, the job response time deteriorates when R increases from 2 to 3 in some cases.

Interestingly, increasing H does not bring in significant decrease in job response time, suggesting 20% of the available slots are sufficient to accommodate the task replicas needed for the test scenarios. In terms of the number of speculative tasks, as expected, the number of duplicated tasks generally increases as higher H or R values are used. However, the number of duplicated tasks issued at various H values be-

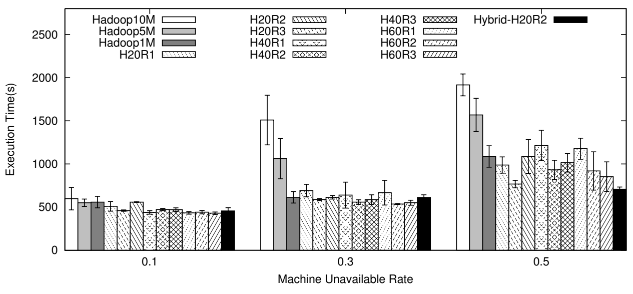


(a) Execution Time

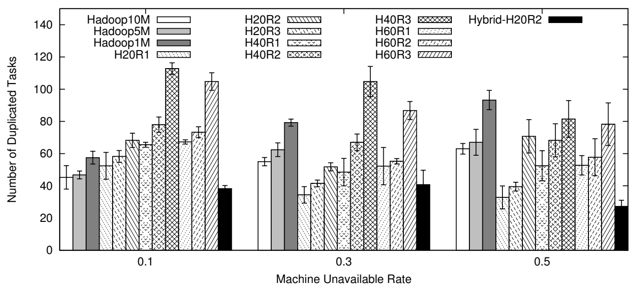


(b) Number of Duplicated Tasks

Figure 3: `Sort` execution profile with Hadoop and MOON scheduling policies.



(a) Execution Time



(b) Number of Duplicated Tasks

Figure 4: `Wordcount` execution profile of Hadoop and MOON scheduling policies.

comes closer as the node unavailability rate increases. Recall that speculative tasks will only be issued after all original tasks have been rescheduled. As a result, the candidate tasks for speculative execution are in the last batch of executing original tasks. As the node unavailability level increases, the number of available slots decreases and so does the number of candidate tasks for speculative execution. The fact that the number of duplicated tasks is comparable across different H levels at 0.5 node unavailability rate suggests that at this volatile level the number of speculative tasks issued is smaller than 20% of the available slots.

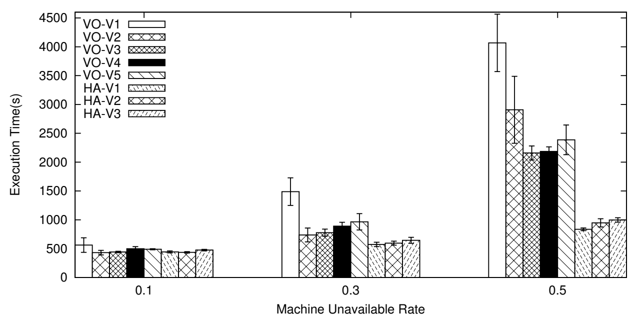
One advantage of the MOON two-phase scheduling algorithm is that it provides the necessary knobs for users to tune the system for overall better performance under certain resource constraint, i.e., by allowing aggressive replication for individual tasks yet retaining control of the overall replication cost. According to the execution profile of both execution time and the number of speculative tasks, without enabling the hybrid-aware enhancement, H20R2 delivers an overall better performance with relatively lower replication cost among various MOON two-phase configurations. Compared to the Hadoop default scheduling, H20R2 outperforms Hadoop1M in job response time by 10%, 13% and 40% at 0.1, 0.3 and 0.5 node unavailability rates, respectively. Meanwhile, H20R2 issues slightly more (11%) duplicated tasks than Hadoop at 0.1 node unavailability rate, but saves 38% and 57% at 0.3 and 0.5 node availability rates, respectively. Further, H20R2 with the hybrid-aware enhancement brings in additional savings in job execution time and task replication cost. Particularly, Hybrid-H20R2 runs 23% faster and issues 19% less speculative tasks than H20R2 at 0.5 node unavailability rate. In summary, when tuned properly, MOON scheduling can achieve significantly better performance than the Hadoop scheduling with comparable or lower replication cost.

Figure 4 shows the execution profile of the `word count` application. The overall trends of default Hadoop scheduling are very similar to those of the `sort` application. For the MOON two-phase scheduling, while the overall performance trends are still similar, one noticeable difference is that the performance differences between various configurations are smaller at 0.3 node unavailability rate. One possible reason is that `word count` has a much smaller number of reduce tasks. Interestingly, among MOON configurations without hybrid-aware enhancement, H20R2 again achieves an overall better performance and lower replication cost, indicating the possibility of having a common configuration for a class of applications. Similarly, MOON H20R2 delivers considerable performance improvements (up to 29%) to Hadoop1M but with a much lower task replication cost (up to 58%). Hybrid-H20R2 again delivers additional performance gain and saving at replication cost to H20R2.

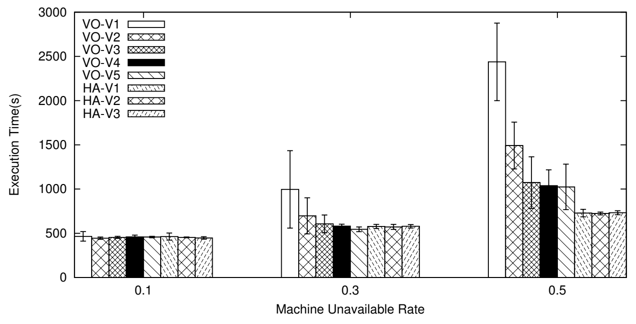
Overall, we found that the default Hadoop scheduling policy may enhance its capability of handling task suspensions in opportunistic environments, but at the cost of shortening *TrackerExpiryInterval* and issuing more speculative tasks. The two-phase scheduling and hybrid-aware scheduling approaches in MOON provide effective tuning mechanism for users to achieve overall significant improvements over Hadoop, especially when the node unavailability is high.

6.2 Replication of Intermediate Data

In a typical Hadoop job, there is a *shuffle* phase in the beginning of a Reduce task. The shuffle phase copies the



(a) `sort`



(b) `word count`

Figure 5: Compare impacts of different replication policies for intermediate data on execution time.

corresponding intermediate data from all Map tasks and is time-consuming even in dedicated environments. On opportunistic environments, achieving efficient shuffle performance is more challenging, given that the intermediate data could be unavailable due to frequent machine outage. In this section, we evaluate the impact of MOON's intermediate data replication policy on shuffle efficiency and consequently, job response time.

We compare a *volatile-only* (VO) replication approach that statically replicates intermediate data only on volatile nodes, and the *hybrid-aware* (HA) replication approach described in Section 4.1. For the VO approach, we increase the number of volatile copies gradually from 1 (`VO-V1`) to 5 (`VO-V5`). For the HA approach, we have MOON store one copy on dedicated nodes when possible, and increase the minimum volatile copies from 1 (`HA-V1`) to 3 (`HA-V3`). Recall that in the HA approach, if the data block does not yet have a dedicated copy, then the number of volatile copies of a data block is dynamically adjusted such that the availability of a file reaches 0.9.

These experiments use 60 volatile nodes and 6 dedicated nodes. To focus solely on intermediate data, we configure the input/output data to use a fixed replication factor of $\{1, 3\}$ across all experiments. Also, the task scheduling algorithm is fixed at Hybrid-H20R2, which was shown to deliver overall better performance under various scenarios.

In Hadoop, a Reduce task reports a fetch failure if the intermediate data of a Map task is inaccessible. The JobTracker will reschedule a new copy of a Map task if more than 50% of the running Reduce tasks report fetch failures for the Map task. We observe that with this approach, the reaction to the loss of Map output is too slow, and as a result, causing hourly long execution time for a job as a reduce task would have to acquire data from hundreds of Map outputs. We remedy this by having the JobTracker issue a new copy of a Map task if 1) three fetch failures have been reported for the Map task and 2) there is no active replicas of the Map output.

Figure 5(a) shows the results of `sort`. As expected, enhanced intermediate data availability through the VO replication clearly reduces the overall execution time. When the unavailability rate is low, the HA replication does not exhibit much additional performance gain. However, HA replication significantly outperforms VO replication when the node unavailability level is high. While increasing the number of volatile replicas can help improve data availability on a highly volatile system, this incurs a high performance cost caused by the extra I/O. As a result, there is no further execution time improvement from `VO-V3` to `VO-V4`, and from `VO-V4` to `VO-V5`, the performance actually degrades. With HA replication, having at least one copy written to dedicated nodes substantially improves data availability, with a lower overall replication cost. More specifically, `HA-V1` outperforms the best VO configuration, i.e., `VO-V3` by 61% at the 0.5 unavailability rate.

With `word count`, the gap between the best HA configuration and the best VO configuration is small. This is not surprising, as `word count` generates much smaller intermediate/final output and has much fewer Reduce tasks, thus the cost of fetching intermediate results can be largely hidden by Map tasks. Also, increasing the number of replicas does not incur significant overhead. Nonetheless, at the 0.5 unavailability rate, the HA replication approach still outperforms the best VO replication configuration by about 32.5%.

To further understand the cause of performance variances of different policies, Table 2 shows the execution profile collected from the Hadoop job log for tests at 0.5 unavailability rate. We do not include all policies due to space limit. For `sort`, the average Map execution time increases rapidly as higher replication degrees are used in the VO replication approach. In contrast, the Map execution time does not change much across different policies for `word count`, due to reasons discussed earlier.

The most noticeable factor causing performance differences is the average *shuffle* time. For `sort`, the average shuffle time of `VO-V1` is much higher than other policies due to the low availability of intermediate data. In fact, the average shuffle time of `VO-V1` is about 5 times longer than that of `HA-V1`. For VO replication, increasing the replication degree from 1 to 3 results in a 54% improvement in the shuffle time, but no further improvement is observed beyond this point. This is because the shuffle time is partially affected by the increasing Map execution time, given that the shuffle time is measured from the start of a reduce task till the end of copying all related Map results. For `word count`, the shuffle times with different policies are relatively close except with `VO-V1`, again because of the smaller intermediate data size.

Finally, since the fetch failures of Map results will trigger the re-execution of corresponding Map tasks, the average number of killed Map tasks is a good indication of the intermediate data availability. While the number of killed Map tasks decreases as the VO replication degree increases, the

Table 2: Execution profile of different replication policies at 0.5 unavailability rate.

	sort				word count			
Policy	VO-V1	VO-V3	VO-V5	HA-V1	VO-V1	VO-V3	VO-V5	HA-V1
Avg Map Time (s)	21.25	42	71.5	41.5	100	110.75	113.5	112
Avg Shuffle Time (s)	1150.25	528	563	210.5	752.5	596.25	584	559
Avg Reduce Time (s)	155.25	84.75	116.25	74.5	50.25	28	28.5	31
Avg #Killed Maps	1389	55.75	31.25	18.75	292.25	32.5	30.5	23
Avg #Killed Reduces	59	47.75	55.25	34.25	18.25	18	15.5	12.5

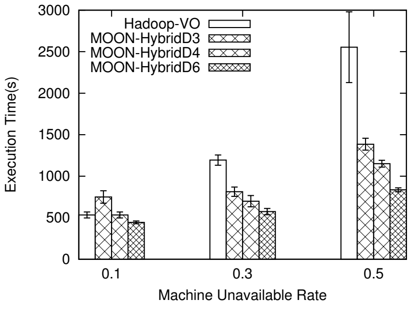
HA replication approach in general results in a lower number of Map task re-executions.

6.3 Overall Performance Impacts of MOON

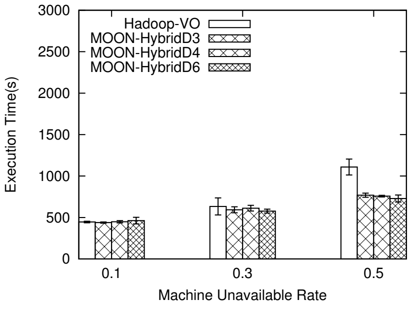
To evaluate the impact of MOON on overall MapReduce performance, we establish a baseline by augmenting Hadoop to replicate the intermediate data and configure Hadoop to store six replicas for both input and output data, to attain a 99.5% data availability when the average machine unavailability is 0.4 (selected according to the real node availability trace shown in Figure 1). For MOON, we assume the availability of a dedicated node is at least as high as that of three volatile nodes together with independent failure probability. That is, the unavailability of dedicated node is less than 0.4^3, which is not hard to achieve for well maintained workstations. As such, we configure MOON with a replication factor of $\{1, 3\}$ for both input and output data.

In testing the native Hadoop system, 60 volatile nodes and 6 dedicated nodes are used. These nodes, however are all treated as volatile in the Hadoop tests as Hadoop cannot differentiate between volatile and dedicated. For each test, we use the VO replication configuration that can deliver the best performance under a given unavailability rate. It worth noting that we do not show the performance of the default Hadoop system (without intermediate data replication), which was *unable* to finish the jobs under high machine unavailability levels, due to intermediate data losses and high task failure rate.

The MOON tests are executed on 60 volatile nodes with 3, 4 and 6 dedicated nodes, corresponding to a 20:1, 15:1 and 10:1 V-to-D ratios. The intermediate data is replicated with the HA approach using $\{1, 1\}$ as the replication factor. As shown in Figure 6, MOON clearly outperforms Hadoop-VO for 0.3 and 0.5 unavailable rates and is competitive at a 0.1 unavailability rate, even for a *20:1 V-to-D ratio*. For `sort`, MOON outperforms Hadoop-VO by a factor of 1.8, 2.2 and 3 with 3, 4 and 6 dedicated nodes, respectively, when the unavailability rate is 0.5. For `word count`, the MOON performance is slightly better than augmented Hadoop, delivering a speedup factor of 1.5 compared to Hadoop-VO. The only case where MOON performs worse than Hadoop-VO is for the `sort` application at the 0.1 unavailability rate and the V-to-D node ratio is 20:1. This is due to the fact that the aggregate I/O bandwidth on dedicated nodes is insufficient to quickly absorb all of the intermediate and output data; as described in Section 4.1, a reduce task will not be flagged as complete until its output data reaches the predefined replication factor (including 1 dedicate copy). Note that while our experiments focus on the performance of executing MapReduce jobs, one important advantage of MOON lies in its ability of tolerating large-scale correlated node unavailability, which is impractical for MapReduce on pure volatile resources.



(a) `sort`



(b) `word count`

Figure 6: Overall performance of MOON vs. Hadoop with VO replication

7. DISCUSSION

While automatic system configuration is out of the scope of the paper, our performance evaluations provide some insights about the trade-off in provisioning the hybrid resources. Particularly, users will be interested in knowing how much dedicated resources are needed for a certain volunteer computing system. Compared to MapReduce on pure volatile resources, MOON can achieve high data availability with much lower replication cost, and in turn, significantly reduce the cost of Map task re-execution caused by the loss of intermediate data. However, despite MOON's scalable I/O scheduling design, the limited I/O bandwidth on the dedicated nodes could cause an extra delay in the task execution. Therefore, the dedicated resources should be provisioned such that a good balance is achieved between the saving of Map task re-execution and the extra write delay on dedicated nodes.

However, an optimal hybrid resource configuration depends on a myriad of factors including the Map and Reduce task execution time, the node unavailability rate, the I/O bandwidths on both volatile and dedicated nodes, and the sizes of different types of MapReduce data etc., letting alone the dynamics in the system created by the speculative task executions. Note that the more I/O intensive is an application, the more dedicated nodes are required to achieve good performance, as suggested the performance differences between the `sort` and `word count` applications. In practice, a good provisioning configuration can be found by performance-tuning the most data-intensive application on the target system. In fact, the `sort` benchmark is a good tuning candidate as it has a very high I/O-to-compute ratio.

Due to the testbed constraint, the current MOON design does not explore optimizations that take into account the network topologies. In the future, we plan to investigate task scheduling and data placement policies for large systems with multi-level network hierarchies. For instance, one possibility is to improve the aggregate bandwidth between volatile nodes and dedicated nodes by distributing dedicated computers in different subnets where the computers are interconnected with high-bandwidth networks.

8. RELATED WORK

Several storage systems have been designed to aggregate idle disk spaces on desktop computers within local area network environments [2, 15, 23, 9]. Farsite [2] aims at building a secure file system service equivalent to centralized file system on top of untrusted PCs. It adopts replication to ensure high data reliability, and is designed to reduce the replication cost by placing data replicas based on the knowledge of failure correlation between individual machines. Glacier [15] is a storage system that can deliver high data availability under large-scale correlated failures. It does not assume any knowledge of the machine failure patterns and uses erasure code to reduce the data replication overhead. Both Farsite and Glacier are designed for typical I/O activities on desktop computers and are not sufficient for high-performance data-intensive computing. Freeloader [23] provides a high-performance storage system. However, it aims at providing a read-only caching space and is not suitable for storing mission critical data.

Gharaibeh et. al. proposed a low-cost reliable storage system built on a combination of scavenged storage of desktop computers and a set of low-bandwidth dedicated storage such as Automated Tape Library (ATL) or remote storage system such as Amazon S3 [10]. Their prosed system focuses sole on storage and mainly supports read-intensive workloads. Also, the storage scavenging in their system does not consider the unavailability caused by the owner activities on a desktop computer. While also adopting a hybrid resource provisioning approach, MOON handles both computation and storage as well as investigates the interactions between the two within the MapReduce programming model.

There have been studies in executing MapReduce on grid systems, such as GridGain [13]. There are two major differences between GridGain and MOON. First, GridGain only provides computing service and relies on other data grid systems for its storage solution, whereas MOON provides an integrated computing and data solution by extending Hadoop. Second, unlike MOON, GridGain is not designed to provide high QoS on opportunistic environments where machines will be frequently unavailable. Sun Microsystems' Compute Server technology is also capable of executing MapReduce jobs on a grid by creating a master-worker task pool where workers iteratively grab tasks to execute [21]. However, based on information gleaned from [21], it appears that this technology is intended for use on large dedicated resources, similarly to Hadoop.

When executing Hadoop in heterogenous environments, Zaharia et. al. discovered several limitations of the Hadoop speculative scheduling algorithm and developed the LATE (Longest Approximate Time to End) scheduling algorithm [24]. LATE aims at minimizing Hadoop's job response time by always issuing a speculative copy for the task that is expected to finish last. LATE was designed on heterogeneous, *dedicated* resources, assuming the task progress rate is constant on a node. LATE is not directly applicable to opportunistic environments where a high percentage of tasks can be frequently suspended or interrupted, and in turn the task progress rate is not constant on a node. Currently, the MOON design focuses on environments with homogeneous computers. In the future, we plan to explore the possibility of combining the MOON scheduling principles with LATE to support heterogeneous, opportunistic environments.

Finally, Ko et al. discovered that the loss of intermediate data may result in considerable performance penalty in Hadoop even under dedicated environments [17]. Their preliminary studies suggested that simple replication approaches, such as relying on HDFS's replication service used in our paper, could incur high replication overhead and is impractical in dedicated, cluster environments. In our study, we show that in opportunistic environments, the replication overhead for intermediate data can be well paid off by the performance gain resulted from the increased data availability. Future studies in more efficient intermediate data replication will of course well complement the MOON design.

9. CONCLUSION AND FUTURE WORK

In this paper, we presented MOON, an adaptive system that supports MapReduce jobs on opportunistic environments, where existing MapReduce run-time policies fail to handle frequent node outages. In particular, we demonstrated the benefit of MOON's data and task replication design to greatly improve the QoS of MapReduce when running on a hybrid resource architecture, where a large group of volatile, volunteer resources is supplemented by a small set of dedicated nodes.

Due to testbed limitations in our experiments, we used homogeneous configurations across the nodes used. Although node unavailability creates natural heterogeneity, it did not create disparity in hardware speed (such as disk and network bandwidth speeds). In our future work, we plan to evaluate and further enhance MOON in heterogeneous environments. Additionally, we would like to deploy MOON on various production systems with different degrees of volatility and evaluate a variety of applications in use on these systems. Lastly, this paper investigated single-job execution, and it would be interesting future work to study the scheduling and QoS issues of concurrent MapReduce jobs on opportunistic environments.

10. ACKNOWLEDGMENTS

This research is supported in part by NSF grants CNS-0546301, CNS-0915861, and CSR-0916719 as well as Xiaosong Ma's joint appointment between North Carolina State

University and Oak Ridge National Laboratory. We thank Advanced Research Computing at Virginia Tech (ARC@VT) for the use of System X cluster in support of this research.

11. REFERENCES

[1] Hadoop. http://hadoop.apache.org/core/.

[2] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J.Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[3] D. Anderson. Boinc: A system for public-resource computing and storage. *Grid Computing, IEEE/ACM International Workshop on*, 0, 2004.

[4] Apple Inc. Xgrid. `http://www.apple.com/server/macosx/technology/xgrid.html`.

[5] S. Chen and S. Schlosser. Map-reduce meets wider varieties of applications meets wider varieties of applications. Technical Report IRP-TR-08-05, Intel Research, 2008.

[6] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63, 2003.

[7] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient Replica Maintenance for Distributed Storage Systems. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 4–4, Berkeley, CA, USA, 2006. USENIX Association.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[9] G. Fedak, H. He, and F. Cappello. Bitdew: a programmable environment for large-scale data management and distribution. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[10] A. Gharaibeh and M. Ripeanu. Exploring Data Reliability Tradeoffs in Replicated Storage Systems. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 217–226, New York, NY, USA, 2009. ACM.

[11] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proceedings of the 19th Symposium on Operating Systems Principles*, 2003.

[12] M. Grant, S. Sehrish, J. Bent, and J. Wang. Introducing map-reduce to high end computing. 3rd Petascale Data Storage Workshop, Nov 2008.

[13] GridGain Systems, LLC. Gridgain. http://www.gridgain.com/.

[14] A. Gupta, B. Lin, and P. A. Dinda. Measuring and understanding user comfort with resource borrowing. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 214–224, Washington, DC, USA, 2004. IEEE Computer Society.

[15] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*, May 2005.

[16] B. Javadi, D. Kondo, J.-M. Vincent, and D. P. Anderson. Mining for Statistical Models of Availability in Large-Scale Distributed Systems: An Empirical Study of SETI@home. In *17th IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, September 2009.

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

[18] D. Kondo, M. Taufe, C. Brooks, H. Casanova, and A. Chien. Characterizing and evaluating desktop grids: an empirical study. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004.

[19] A. Matsunaga, M. Tsugawa, and J. Fortes. Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics. Microsoft eScience Workshop, 2008.

[20] J. Strickland, V. Freeh, X. Ma, and S. Vazhkudai. Governor: Autonomic throttling for aggressive idle resource scavenging. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing*, 2005.

[21] Sun Microsystems. Compute server. https://computeserver.dev.java.net/.

[22] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 2004.

[23] S. Vazhkudai, X. Ma, V. Freeh, J. Strickland, N. Tammineedi, and S. Scott. Freeloader: Scavenging desktop storage resources for bulk, transient data. In *Proceedings of Supercomputing*, 2005.

[24] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.

[25] M. Zhong, K. Shen, and J. Seiferas. Replication degree customization for high availability. *SIGOPS Oper. Syst. Rev.*, 42(4), 2008.