Cluster Comput
DOI 10.1007/s10586-011-0158-7

Reliable MapReduce computing on opportunistic resources

Heshan Lin · Xiaosong Ma · Wu-chun Feng

Received: 2 November 2010 / Accepted: 13 January 2011

Abstract MapReduce offers an ease-of-use programming paradigm for processing large data sets, making it an attractive model for opportunistic compute resources. However, unlike dedicated resources, where MapReduce has mostly been deployed, opportunistic resources have significantly higher rates of node volatility. As a consequence, the data and task replication scheme adopted by existing MapReduce implementations is woefully inadequate on such volatile resources.

In this paper, we propose MOON, short for *MapReduce On Opportunistic eNvironments*, which is designed to offer reliable MapReduce service for opportunistic computing. MOON adopts a hybrid resource architecture by supplementing opportunistic compute resources with a small set of dedicated resources, and it extends Hadoop, an open-source implementation of MapReduce, with adaptive task and data scheduling algorithms to take advantage of the hybrid resource architecture. Our results on an emulated opportunistic computing system running atop a 60-node cluster demonstrate that MOON can deliver significant performance improvements to Hadoop on volatile compute resources and even finish jobs that are not able to complete in Hadoop.

Keywords MapReduce · Cloud computing · Volunteer computing

H. Lin (✉) · W. Feng
Virginia Tech, Blacksburg, USA
e-mail: hlin2@cs.vt.edu

W. Feng
e-mail: feng@cs.vt.edu

X. Ma
Oak Ridge National Laboratory, North Carolina State University, Raleigh, USA
e-mail: ma@cs.ncsu.edu

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 an individual personal computer; it requires a more powerful resource such as a cluster supercomputer. Despite the success of commodity clusters for supercomputing, such dedicated high-performance computing (HPC) platforms are still expensive to acquire and maintain for many institutions, necessitating more affordable parallel computing.

One approach to provide cost-efficient HPC is to harness the unused compute resources from personal computers or server farms, such as clusters and compute clouds. For example, volunteer computing systems [4, 5, 8, 23] allow users to run their jobs on idle cycles donated by the owners of desktop computers. These idle compute cycles are virtually "free" to institutions' operational cost. Idle compute resources can also be available in cluster or cloud computing environments because of resource scheduling and provisioning constraints. Recently, Amazon EC2 has allowed customers to use EC2's unused compute capability with a bid price lower than that of normal on-demand compute instances [2]. Specifically, EC2 periodically sets a price (called "Spot Price") on the unused compute resources. A customer gains access to these resources (via "Spot Instances") if the bid price is higher than the Spot Price and loses access to those resources when the Spot Price goes higher than the bid price.

Since harnessed idle resources can freely come and go, we refer to the computing environments atop those resources as *opportunistic environments*. While efficient computing in opportunistic environments has been extensively studied in

Published online: 27 February 2011

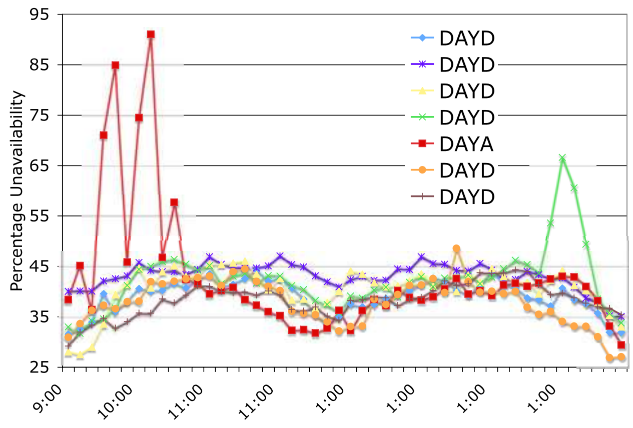


Fig. 1 Percentage of unavailable resources measured on a production volunteer computing system

the past, existing studies mainly focus on CPU-intensive, embarrassingly parallel workloads. Traditional parallel programming models such as MPI, which assume high machine availability, do not work well on opportunistic environments. As such, there lacks standard programming tools for parallel data processing on these environments.

The emergence and growing popularity of MapReduce [10] may bring a change to the landscape of opportunistic computing. 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 available resources. While this union is conceptually appealing, a vital issue needs to be addressed—opportunistic computing systems are significantly more volatile than dedicated computing environments, where MapReduce has mostly been deployed so far.

For example, while Ask.com per-server unavailability rate is an astonishingly low 0.0455% [26], availability traces collected from an enterprise volunteer computing system [19] show a very different picture, as shown in Fig. 1. The figure shows that the unavailability rate of individual nodes averages around 40% with as many as 90% of the resources simultaneously inaccessible. 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 [23] 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.

In this work, we first evaluate Hadoop, a popular, open-source MapReduce run-time system [1], on an emulated opportunistic environment and observe that the volatility of opportunistic resources creates several severe problems.

1. 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.
2. 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.
3. Hadoop task scheduling assumes that the majority of the tasks will run smoothly until completion. However, tasks can be frequently suspended or interrupted on opportunistic environments. The default Hadoop task replication strategy, designed to handle failures, is insufficient to handle the high resource volatility.

To mitigate these problems in order to realize the computing potential of MapReduce on opportunistic environments, we propose a novel system called MOON—MapReduce On Opportunistic eNvironments. MOON adopts a hybrid resource architecture by provisioning a small set of dedicated, reliable computers to supplement the volatile compute nodes. Leveraging such a hybrid architecture, MOON then extends Hadoop's task and data scheduling to greatly improve the quality of service (QoS) of MapReduce. Together with detailed descriptions of MOON, we present an extensive evaluation of its design on emulated opportunistic environments. Our results show that MOON can deliver as much as a *six-fold* speedup to Hadoop and can even finish MapReduce jobs that would not otherwise complete in highly volatile environments.

2 Background

In this section, we present some background information on volunteer computing systems and the MapReduce programming model.

2.1 Volunteer computing

Many volunteer computing systems have been developed to harness idle desktop resources for high-throughput computing [4, 5, 8, 23]. A common feature shared by these systems is non-intrusive deployment. While studies have been conducted on aggressively stealing computer cycles [21] and its corresponding impact [16], 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 [10]. 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 [7], bioinformatics [20], astrophysics and cyber-security [14].

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 [13], 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 system *metadata*, whereas 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

As discussed in Sect. 1, compute resources in opportunistic environments can be highly volatile and thus are not dependable enough to offer reliable compute and storage services needed to support efficient MapReduce computing. Consequently, MOON adopts a hybrid architecture by supplementing volatile compute instances with a set of dedicated compute instances. For cost-effective purpose, MOON assumes that the number of dedicated instances is much smaller than that of volatile ones.

The hybrid resource architecture of MOON 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 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. Suppose the unavailability rate is 0.001 for a dedicated computer,[2] achieving 99.99% availability requires only one copy on the dedicated node and three copies on the volatile nodes.[3] *Second*, long-running tasks with execution times much longer than the mean time between failures (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 these dedicated nodes, the system can function even when a large percentage of nodes are temporarily unavailable.

There are several major assumptions in the current MOON design:

- MOON targets institutional intranet environments or cluster environments, where compute nodes are connected with a local area network with relatively high bandwidth and low latency.
- As will be discussed in Sect. 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 assumption is made practical by the decreasing price of commodity servers and hard drives with large capacity.

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

[2] For example, the well-maintained workstations in our research lab have had only 10 hours of unscheduled downtime in the past which is equivalent to a 0.001 unavailability rate.

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

- We assume that security solutions of existing volunteer computing systems and cloud platforms can be applied to the MOON system. Consequently, we currently do not directly address the security issue.
- 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 future work. Please also note that MOON is designed to support general MapReduce applications and does not make assumptions on job characteristics.

3.1 Example usage scenarios

MOON is designed for a broad range of opportunistic computing environments. Here we give some example usage scenarios of MOON.

- In desktop computing environments, MOON can be deployed atop volunteer computing systems such as Condor [23] and Entropia [8]. Leveraging the cycle-stealing features of volunteer computing systems, MOON processes can be suspended and resumed according to the activities of desktop owners, thus harnessing idle desktop resources for parallel data processing in a non-intrusive manner.
- In cluster environments, MOON can be used to improve resource utilization for a mix of interactive and batched workloads. For instance, the maturity of virtualization technologies has motivated universities to offer on-demand virtual compute instances (e.g., for a computer science class) for education [6]. User requests are typically bursty, leaving the underline cluster underutilized from time to time. Harnessing idle resources in these systems for parallel computing is nontrivial because the idle resources can be claimed by interactive user requests at any time. MOON can be used to piggyback batched parallel workloads on the idle resources.
- In cloud environments, MOON can help users better leverage the low-cost bid instances, e.g., the *Spot Instances* in Amazon EC2 mentioned in Sect. 1. The hybrid architecture of MOON can help save compute progress when Spot Instances become unavailable. MOON can also be used to unify the programing (i.e., using MapReduce) on both regular and bid instances.

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* [9] alone is insufficient to provide high-quality MapReduce services. For instance, in a volunteer computing system, 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, MOON data management focuses on improving overall *data availability*.

As mentioned in Sect. 1, we find 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 opportunistic environments 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.

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

4.1 Multi-dimensional, cost-effective replication

MOON provides a multi-dimensional, dynamic replication service to handle resource volatility as opposed to the static data replication in Hadoop. MOON manages two types of resources—*supplemental dedicated nodes* and *volatile volunteer nodes*. The number of dedicated nodes 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.

Intuitively, since dedicated nodes have a 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 volatile 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 are converted to reliable files once the job is completed.

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 Sect. 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.

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 addressed 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 converted 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. MOON addresses this by prioritizing I/O requests on the different resources. Specifically, to alleviate read traffic on dedicated nodes, MOON factors in the node type in servicing a read request. 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 depicted in Fig. 2.

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 by piggybacking it on the *heartbeat* messages. The throttling algorithm running on the NameNode compares the updated bandwidth with the average I/O bandwidth during the past window. If the consumed

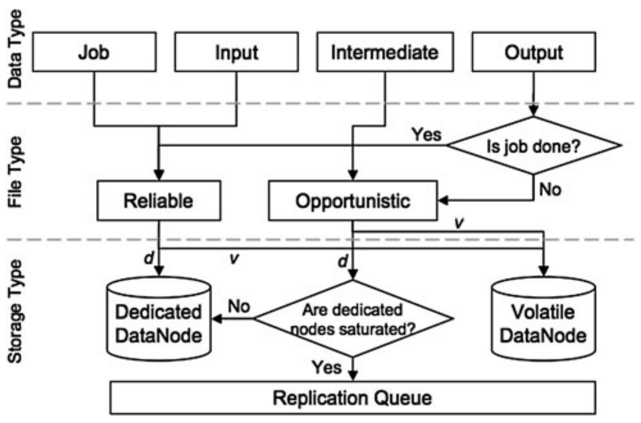


Fig. 2 Decision process to determine where data should be stored

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 ($bw_i < avg\_bw \times (1 + T_b)$) then
        $state = throttled$
    end if
end if
if $bw_i < avg\_bw$ then
    if ($state == throttled$) and ($bw_i < avg\_bw \times (1 - T_b)$) then
        $state = unthrottled$
    end if
end if
```

I/O bandwidth on 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 Adaptive replication

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 unreliable nodes, 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 if availability statistics are available for the deployment environment.

The rationale for the above adaptive replication design 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.

4.4 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 when 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.

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 resource unavailability rate is high, *all* instances of a task can possibly be suspended simultaneously, allowing no progress to be made on that task.
– Second, a fast progressing task may be suddenly slowed down when a node becomes unavailable. Yet, it may take a long time for a suspended task with a high progress score to be eligible for speculative execution.
– 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,[4] 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 Sects. 5.1 and 5.2, and hybrid-architecture-specific augmentations in Sect. 5.3.

[4]A similar observation is made when running Hadoop on heterogeneous environments [25].

5.1 Ensuring sufficient progress under high resource volatility

In order to guarantee that sufficient progress is made on all tasks, MOON characterizes stragglers into *frozen tasks* (tasks whose *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 for speculative execution from the frozen list first. In both lists, tasks are sorted in the order of their progress scores.

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 a *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 the TaskTracker will return 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 program performance. We will evaluate various configurations of the two parameters in Sect. 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 Sect. 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.

Table 1 Application configurations

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

6 Performance evaluation

On production opportunistic computing systems, resource 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. As such, in our experiments, we emulate an opportunistic computing system with synthetic node availability traces, where node availability level can be adjusted.

Our experiments are executed on System X at Virginia Tech, comprised of Apple Xserve G5 compute nodes with dual 2.3 GHz PowerPC 970FX processors, 4 GB of RAM, 80 GByte hard drives. System X uses a 10 Gbs InfiniBand network and a 1 Gbs 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.

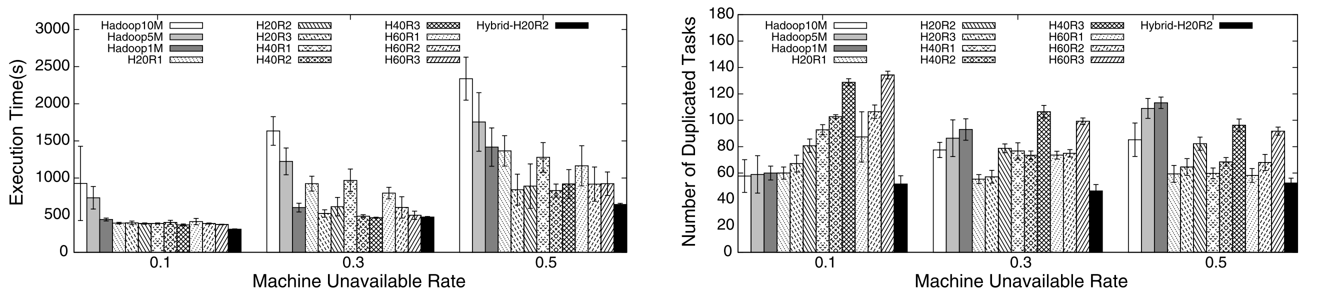
Our experiments use two representative MapReduce applications, i.e., `sort` and `word count`, that are distributed with Hadoop. The configurations of the two applications are given in Table 1.[5] For both applications, the input data is randomly generated using tools distributed with Hadoop.

6.1 Independent resource unavailability

In this section, 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 [19]. 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 run time 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.[6]

[5]These two applications with similar data input sizes were also used in other MapReduce studies, e.g., [25].

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



(a) Execution Time (b) Number of Duplicated Tasks

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

6.1.1 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).

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 Sect. 6.1.3). We compare the original Hadoop task scheduling policy and the MOON scheduling algorithm described in Sect. 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 Sect. 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 Sect. 5.3) for $H = 20$ and $R = 2$.

Figure 3(a) shows the execution time for the `sort` application with increasing node unavailability 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 Fig. 3(b). Thus, the default Hadoop scheduling is not flexible to simultaneously achieve 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. How-

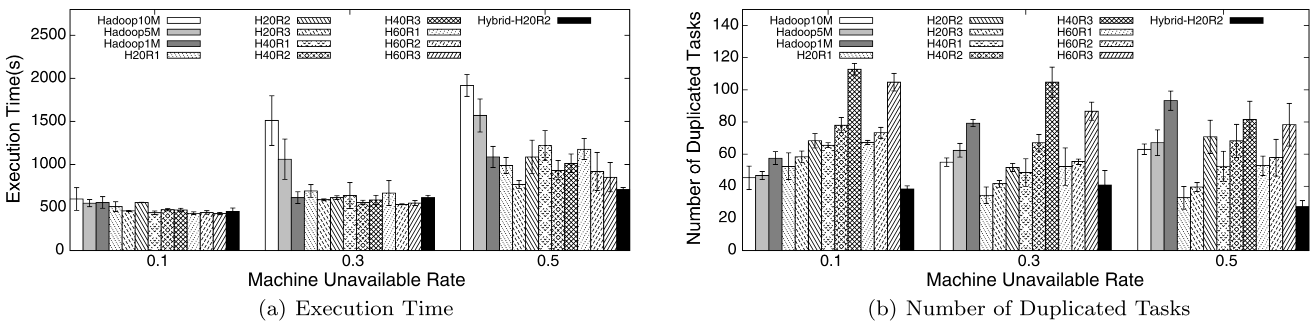


Fig. 4 `Wordcount` execution profile of Hadoop and MOON scheduling policies

ever, the number of duplicated tasks issued at various H values becomes 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 constraints, 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. Furthermore, 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.

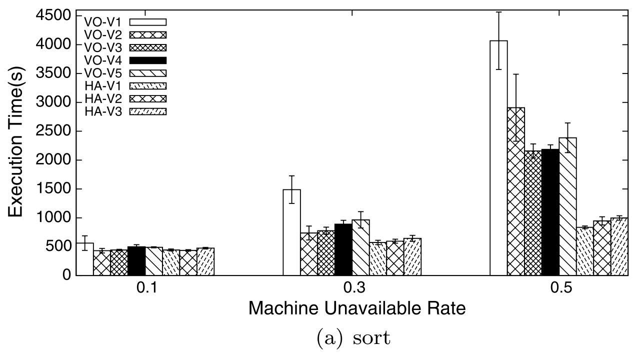
Figure 4 shows the execution profile of the `word count` application. The overall trends of default Hadoop scheduling are very similar to those in the `sort` application. For the MOON two-phase scheduling, while the overall performance trends are still similar, one noticeable difference is that the performance 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 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.

6.1.2 *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 corresponding intermediate data from all Map tasks. This phase is time-consuming even in dedicated environments. On opportunistic environments, achieving efficient shuffle performance is more challenging, because the intermediate data could be unavailable due to frequent node 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 Sect. 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



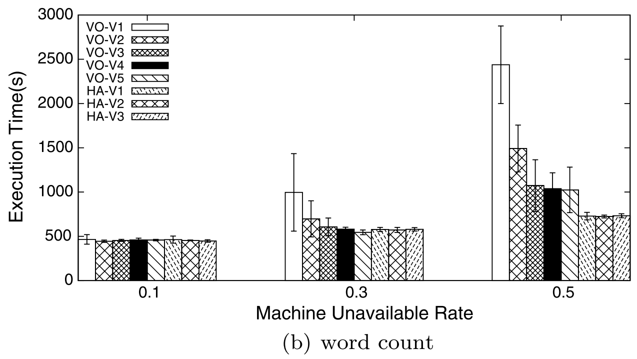


Fig. 5 Compare impacts of different replication policies for intermediate data on execution time

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

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 lost Map output is too slow, and consequently, 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 smaller. 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 the execution of Map tasks. Also, increasing the number of replicas does not incur significant overhead. Nonetheless, at 0.5 unavailability rate, the HA replication approach 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. 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.

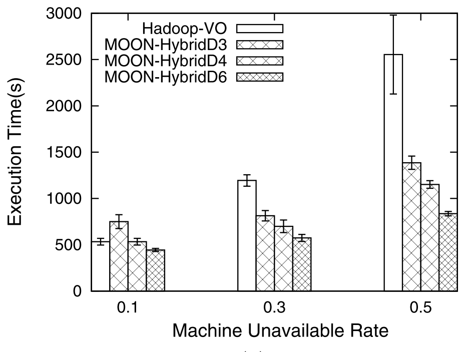
Finally, since the fetch failures of Map results will trigger the re-execution of corresponding Map tasks, the average number of killed Map tasks is a good indication of the intermediate data availability. While the number of killed Map tasks decreases as the VO replication degree increases, the HA replication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

6.1.3 *Overall performance impacts of MOON*

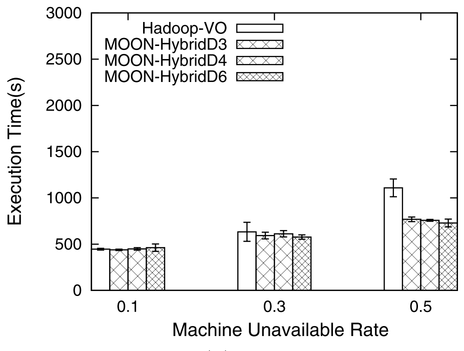
To evaluate the impact of MOON on overall MapReduce performance, we establish a baseline by augmenting Hadoop to replicate the intermediate data and configure Hadoop to store six replicas for both input and output data, to attain a 99.5% data availability when the average node unavailability is 0.4 (selected according to the real node availability trace shown in Fig. 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 to use 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 as Hadoop cannot differentiate between volatile and dedicated nodes. For each test, we use the VO replication configuration that can deliver the best performance under a given unavailability rate. It is 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 node 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



(a) sort



(b) word count

Fig. 6 Overall performance of MOON vs. Hadoop with VO replication

with the HA approach using $\{1, 1\}$ as the replication factor. As shown in Fig. 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 Sect. 4.1, a reduce task will not be flagged as complete until its output data reaches the predefined replication factor (including 1 dedicate copy).

6.2 Correlated resource unavailability

In this experiment, we study the efficacy of the MOON design in the case of correlated unavailabilities. We synthesize a set of node availability traces where groups of nodes alternatively become unavailable. Specifically, the model we used to generate traces takes two parameters: (1) a correlated level represented by P_u, which is the percentage of nodes that are simultaneously unavailable and (2) an unavailable

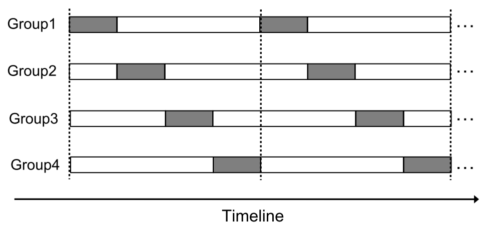


Fig. 7 Machine availability trace generation example ($P_u = 25\%$). The gray box means the group of nodes become off-line for a period of I_u

Table 3 `sort` performance with correlated unavailabilities (measured in seconds)

P_u	10%	30%	50%
Augmented Hadoop	612	2142	>5400 (failed)
MOON	375	504	933

interval I_u. Suppose there are N total nodes, in the traces, $N \times P_u$ nodes form a group, and each group will take turn to be off-line for a period of I_u. Figure 7 gives an example of trace generation when P_u is 25%.

We compare the augmented Hadoop (with intermediate data replication enabled as described in Sect. 6.1.3) and MOON. We use 60 volatile nodes and 6 dedicated nodes, i.e., a V-to-D ratio of 10:1. Similar to the experiment in Sect. 6.1.3, the replication factor is configured as {6} for the augmented Hadoop and {1,3} for MOON. For intermediate data, the augmented Hadoop uses the replication factor that delivers the best performance, and MOON uses a replication factor of {1,1}. We vary P_u from 10% to 30% to 50%, and we fix I_u at 409 seconds, which is the mean unavailable time used in previous experiments.

Table 3 gives the results of the `sort` application. Clearly, MOON can significantly outperforms the augmented Hadoop. Specifically, at a correlated level of 30%, MOON delivers a four-fold speedup over the augmented Hadoop. Furthermore, at a correlated level of 50%, the augmented Hadoop was not able to finish after 5400 seconds because the number of task failures reaches the system threshold. In contrast, MOON finishes within 933 seconds at the same correlated level. The fact that the augmented Hadoop has difficulty in finishing the job is not surprising, because when resource unavailability is correlated, it requires higher replication degree to achieve a same level of data availability compared to the case of independent machine unavailabilities. A too high replication degree can create too much I/O contention in the system, leading to significant slowdown to the program performance. By leveraging the dedicated nodes, MOON can achieve high data availability with much lower replication cost.

The results of the `word count` application is given in Table 4. While MOON still significantly outperforms the

Table 4 `word count` performance with correlated unavailabilities (measured in seconds)

P_u	10%	30%	50%
Augmented Hadoop	664	1206	4807
MOON	429	603	795

augmented Hadoop, the performance difference between the two is slightly smaller than that in the `sort` application. Again, this is because the `word count` application generates less amounts of data and has a smaller number of reduce tasks. Note that at 50% correlated level, MOON delivers a six-fold speedup over the augmented Hadoop.

7 Analytical analysis

In this section, we discuss the advantages of the MOON hybrid design through analytical analysis. Since the major motivation of hybrid resource provisioning is to improve data availability with low replication cost, our model assumes no speculative task execution to avoid overly complicating the models.

In dedicated cluster environments, MapReduce jobs are typically executed in waves. The total execution time of a job with M Map tasks and R Reduce tasks on a cluster of N nodes can be modeled as $\frac{MT_m}{N} + \frac{RT_r}{N}$, where T_m and T_r are the execution time of a Map and Reduce task, respectively. In opportunistic environments, when a node becomes unavailable, the Map tasks complete on that node need to be re-executed to generate the corresponding intermediate data needed by Reduce tasks. Assuming the unavailability rate of the intermediate data is α under a certain data replication policy, on average $M\alpha$ tasks need to be re-executed at each wave of Reduce task execution. As such, the overall execution time of a MapReduce job on a volunteer computing system with average node unavailability rate λ is:

$$\begin{aligned} T_{total} &= \frac{MT_m}{N(1-\lambda)} + \frac{R}{N(1-\lambda)}\left(\frac{M\alpha T'_m}{N(1-\lambda)} + T_r\right) \\ &= \frac{MT_m}{N(1-\lambda)} + \frac{MR\alpha T'_m}{N^2(1-\lambda)^2} + \frac{RT_r}{N(1-\lambda)} \qquad (1) \end{aligned}$$

where T'_m is the execution time of a re-executed Map task (note that on average only $N(1-\lambda)$ nodes are available for running tasks). The three terms in (1) represent the time spent on executing the original Map tasks, the re-executed Map tasks and the Reduce tasks, respectively. The execution time of an original Map task can be modeled as $T_m = t_{mi} + t_{mc} + t_{mo}$, where t_{mi}, t_{mc} and t_{mo} refer to the input, compute and output time of a Map task. Similarly, the execution time of a Reduce task $T_r = t_{ri} + t_{rc} + t_{ro}$. T'_m is slightly different than T_m in the output time, which will be explained in details in the following discussions.

Case 1: Replication on Volatile Nodes Only. We first take a look at the performance impacts of data replication in a system with volatile nodes only. Let v_m be the replication degree of intermediate data. The output time of a Map task, i.e, the time spent on writing the intermediate data will increase as v_m grows. Assuming that the number of Map tasks is much larger than the number of available slots in the system, there will be $N(1-\lambda)$ concurrent Map tasks executing at each wave. Therefore, the average output time of a Map task $t_{mo} = \frac{S_m v_m N(1-\lambda)}{BN(1-\lambda)} = \frac{S_m v_m}{B}$, where S_m is the size of the output of a Map task and B is the I/O bandwidth (the smaller between the network and disk I/O bandwidth) of a compute node.

The replication degree of intermediate data also affects the performance of re-executed Map tasks. With v_m replicas, the unavailability rate of a block of intermediate data $\alpha = \lambda^{v_m}$. Therefore, the number of concurrent re-executed Map tasks is $M\lambda^{v_m}$ at each wave. The output time of a re-executed Map task $t'_{mo} = \frac{S_m v_m M\lambda^{v_m}}{BN(1-\lambda)}$.

Let v_r be the replication degree of the output data. For jobs with a large number of reduce tasks, at each wave there are $N(1-\lambda)$ concurrent Reduce tasks. Similar to the writing of intermediate data, we have $t_{ro} = \frac{S_r v_r}{B}$, where S_r is the size of the output data of a Reduce task.

According to (1) and the analysis above, the execution of a MapReduce task in opportunistic environments with pure volatile nodes is:

$$T_{total}(V) = T_{Org\_Map} + T_{Re\_Map} + T_{Reduce} \tag{2}$$

where T_{Org_Map}, T_{Re_Map} and T_{Reduce} refer to the time spent on executing the original Map tasks, the re-executed Map tasks and the Reduce tasks, and their definitions are given in (3), (4) and (5).

$$T_{Org\_Map} = \frac{M}{N(1-\lambda)}\left(t_{mi} + t_{mc} + \frac{S_m v_m}{B}\right) \tag{3}$$

$$T_{Re\_Map} = \frac{RM\lambda^{v_m}}{N(1-\lambda)}\left(t_{mi} + t_{mc} + \frac{S_m v_m M\lambda^{v_m}}{BN(1-\lambda)}\right) \tag{4}$$

$$T_{Reduce} = \frac{R}{N(1-\lambda)}\left(t_{ri} + t_{rc} + \frac{S_r v_r}{B}\right) \tag{5}$$

To guarantee certain quality of service, the availability of the output data needs to meet a minimum requirement A specified by users, i.e., $1-\lambda_r^v \geq A$. As v_m increases, the execution time of the original Map tasks will increase as suggested by (3), and the execution time of reexecuted Map tasks will decrease according to (4) because of the additional I/O cost. Therefore, an optimal configuration can be found by comparing the performance of increasing v_m values.

Case 2: Replication on Hybrid Resources. On volunteering system with supplemental dedicated nodes, the replication degree is specified as a tuple. To avoid overcomplicating the model, we assume a basic replication strategy where all the data in the system is stored as reliable data, where a write operation will always be guaranteed on the dedicated nodes. Let $\{d_m, v'_m\}$ be the replication degree of the intermediate data. The unavailability rate of the intermediate data would be $\lambda^{v'_m}\lambda_d^{d_m}$, where λ_d is the unavailability rate of a dedicated node. Let B_d is the I/O bandwidth of a dedicated node, and N_d is the number of dedicated nodes. The output time of an original Map task is:

$$t'_{mo} = \max\left(\frac{S_m v'_m}{B}, \frac{S_m N(1-\lambda)}{B_d N_d}\right) \tag{6}$$

With similar derivation for the output time of re-executed Map tasks and reduce tasks, we have the execution time of a MapReduce job on hybrid resources:

$$T_{total}(H) = T'_{Org\_Map} + T'_{Re\_Map} + T'_{Reduce} \tag{7}$$

where

$$T'_{Org\_Map} = \frac{M}{N(1-\lambda)} \times \left(t_{mi} + t_{mc} + \max\left(\frac{S_m v'_m}{B}, \frac{S_m N(1-\lambda)}{B_d N_d}\right)\right) \tag{8}$$

$$T'_{Re\_Map} = \frac{RM\lambda^{v'_m}\lambda_d^{d_m}}{N(1-\lambda)}\Bigg(t_{mi} + t_{mc} + \max\left(\frac{S_m v'_m M\lambda^{v'_m}\lambda_d^{d_m}}{BN(1-\lambda)}, \frac{S_m d_m M\lambda^{v'_m}\lambda_d^{d_m}}{B_d N_d}\right)\Bigg) \tag{9}$$

$$T'_{Reduce} = \frac{R}{N(1-\lambda)} \times \left(t_{ri} + t_{rc} + \max\left(\frac{S_r v'_r}{B}, \frac{S_r N(1-\lambda)}{B_d N_d}\right)\right) \tag{10}$$

As can be seen in (9), the number of re-executed Map tasks is $RM\lambda^{v'_m}\lambda_d^{d_m}$ under the hybrid replication design, as compared to $RM\lambda^{v_m}$ as shown in (4). Since λ_d is typically much smaller than λ, replicating data on dedicated nodes can effectively reduce the number of re-executed Map tasks compared to relying on volatile nodes alone. One the other hand, comparing T_{Org_Map}, T_{Re_Map}, T_{Reduce} and T'_{Org_Map}, T'_{Re_Map}, T'_{Reduce}, when the I/O bandwidth of the dedicated nodes is not sufficient, there can be an extra delay in writing data. As such, the hybrid replication will perform better when the saving in Map tasks re-execution outweighs the extra writing delay caused by the limited I/O bandwidth on dedicated nodes. An optimal configuration of the number of dedicated nodes depends on a myriad of factors including I/O bandwidth of both types of nodes, the compute time of MapReduce tasks, and the node unavailability rates, etc. While automatically configuring such an optimal number is out of the scope of this paper, this number can be found via performance tuning in practice. More specifically, the more I/O intensive is a MapReduce job, the more dedicated resources are required for the hybrid architecture to beat the optimal performance achieved with volatile nodes alone. The sorting benchmark will be a good candidate for performance tuning as it is one of the most I/O intensive applications.

8 Related work

Several storage systems have been designed to aggregate idle disk spaces on desktop computers within local area network environments [3, 11, 17, 24]. Farsite [3] aims at building a secure file system service equivalent to centralized file systems 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 [17] is a storage system that can deliver high data availability under large-scale correlated failures. It does not assume any knowledge of machine failure patterns and uses erasure code to reduce 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 [24] 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 [12]. 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 [15]. 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 [22]. However, based on information gleaned from [22], it appears that this technology is intended for use on large dedicated resources, similarly to Hadoop.

When executing Hadoop in heterogeneous environments, Zaharia et al. discovered several limitations of the Hadoop speculative scheduling algorithm and developed the LATE (Longest Approximate Time to End) scheduling algorithm [25]. 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 for a Hadoop job even under dedicated environments [18]. 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 well complement the MOON design.

9 Conclusion and future work

In this paper, we propose and evaluate an approach to provide a MapReduce computing service on opportunistic resources. Specifically, we find that existing MapReduce frameworks are designed for dedicated compute resources, and thus perform poorly on highly volatile resources. To address this shortcoming, we then present MOON, a novel framework designed to efficiently and reliably execute MapReduce jobs on volatile resources. MOON adopts a hybrid resource architecture that uses dedicated and volatile resources to complement each other, and extends Hadoop to take advantage of such a hybrid architecture. Extensive experiments show that MOON can significantly outperform Hadoop on opportunistic compute environments.

Due to testbed limitations in our experiments, we use homogeneous configurations across the nodes used. Although node unavailability creates natural heterogeneity, it does 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.

Acknowledgements 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 the System X cluster in support of this research.

References

1. Hadoop. http://hadoop.apache.org/core/
2. Spot Instances on Amazon EC2. http://aws.amazon.com/ec2/spot-instances/
3. Adya, A., Bolosky, W., Castro, M., Chaiken, R., Cermak, G., Douceur, J., Howell, J., Lorch, J., Theimer, M., Wattenhofer, R.: FARSITE: federated, available, and reliable storage for an incompletely trusted environment. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation (2002)
4. Anderson, D.: Boinc: a system for public-resource computing and storage. In: IEEE/ACM International Workshop on Grid Computing (2004)
5. Apple Inc. Xgrid. http://www.apple.com/server/macosx/technology/xgrid.html
6. Averitt, S., Bugaev, M., Peeler, A., Shaffer, H., Sills, E., Stein, S., Thompson, J., Vouk, M.: Virtual computing laboratory (VCL). In: International of the International Conference on Virtual Computing Initiative (2007)
7. Chen, S., Schlosser, S.: Map-reduce meets wider varieties of applications meets wider varieties of applications. Technical report IRP-TR-08-05, Intel research (2008)
8. Chien, A., Calder, B., Elbert, S., Bhatia, K.: Entropia: Architecture and performance of an enterprise desktop grid system. J. Parallel Distrib. Comput. **63**, 597–610 (2003)
9. Chun, B.-G., Dabek, F., Haeberlen, A., Sit, E., Weatherspoon, H., Kaashoek, M.F., Kubiatowicz, J., Morris, R.: Efficient replica maintenance for distributed storage systems. In: NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation, Berkeley, CA, USA, pp. 4–4. USENIX Association, Berkeley (2006)
10. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
11. Fedak, G., He, H., Cappello, F.: Bitdew: a programmable environment for large-scale data management and distribution. In: SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Piscataway, NJ, USA, pp. 1–12. IEEE Press, New York (2008)
12. Gharaibeh, A., Ripeanu, M.: Exploring data reliability tradeoffs in replicated storage systems. In: HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing, New York, NY, USA, pp. 217–226. ACM, New York (2009)
13. Ghemawat, S., Gobioff, H., Leung, S.: The Google file system. In: Proceedings of the 19th Symposium on Operating Systems Principles (2003)
14. Grant, M., Sehrish, S., Bent, J., Wang, J.: Introducing map-reduce to high end computing. In: 3rd Petascale Data Storage Workshop, Nov (2008)
15. GridGain Systems, LLC. Gridgain. http://www.gridgain.com/
16. Gupta, A., Lin, B., Dinda, P.A.: Measuring and understanding user comfort with resource borrowing. In: HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing, Washington, DC, USA, pp. 214–224. IEEE Computer Society, Los Alamitos (2004)
17. Haeberlen, A., Mislove, A., Druschel, P.: 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)
18. Ko, S., Hoque, I., Cho, B., Gupta, I.: On availability of intermediate data in cloud computations. In: 12th Workshop on Hot Topics in Operating Systems (HotOS XII) (2009)
19. Kondo, D., Taufe, M., Brooks, C., Casanova, H., Chien, A.: Characterizing and evaluating desktop grids: an empirical study. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium (2004)
20. Matsunaga, A., Tsugawa, M., Fortes, J.: Cloudblast: combining mapreduce and virtualization on distributed resources for bioinformatics. In: Microsoft eScience Workshop (2008)
21. Strickland, J., Freeh, V., Ma, X., Vazhkudai, S.: Governor: Autonomic throttling for aggressive idle resource scavenging. In: Proceedings of the 2nd IEEE International Conference on Autonomic Computing (2005)
22. Sun Microsystems. Compute server. https://computeserver.dev.java.net/
23. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the condor experience. In: Concurrency and Computation: Practice and Experience (2004)
24. Vazhkudai, S., Ma, X., Freeh, V., Strickland, J., Tammineedi, N., Scott, S.: Freeloader: scavenging desktop storage resources for bulk, transient data. In: Proceedings of Supercomputing (2005)
25. Zaharia, M., Konwinski, A., Joseph, A., Katz, R., Stoica, I.: Improving mapreduce performance in heterogeneous environments. In: OSDI (2008)
26. Zhong, M., Shen, K., Seiferas, J.: Replication degree customization for high availability. SIGOPS Oper. Syst. Rev. **42**(4), 55–68 (2008)

Heshan Lin received the BS degree in applied math from South China University of Technology in 1998, the M.S. degree in computer science from Temple University in 2004, and the Ph.D. degree in computer science from North Carolina State University in 2009. He is a Senior Research Associate in the Department of Computer Science at Virginia Tech. His research interests include data-intensive parallel and distributed computing, bioinformatics, cloud computing, and GPU (graphics processing unit) computing.

Xiaosong Ma is currently an Associate Professor in the Department of Computer Science at North Carolina State University. She is also a Joint Faculty in the Computer Science and Mathematics Division at Oak Ridge National Laboratory. Her research interests are in the areas of storage systems, parallel I/O, high-performance parallel applications, and self-configurable performance optimization. She received the DOE Early Career Principal Investigator Award in 2005, the NSF CAREER Award in 2006, and the IBM Faculty Award in 2009. Prior to joining NCSU, Xiaosong received her Ph.D. in computer science from the University of Illinois at Urbana-Champaign in 2003, and her B.S. in computer science from Peking University, China.

Wu-chun Feng (S'88-M'98-SM'04) received B.S. degrees in Computer Engineering and Music and a M.S. degree in Computer Engineering from Penn State University in 1988 and 1990, respectively. He received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 1996. In January 2006, he joined Virginia Tech as an associate professor of computer science and electrical & computer engineering at Virginia Tech (2006-present). Previous professional stints include Los Alamos National Laboratory, The Ohio State University, Purdue University, Orion Multisystems, Vosaic, NASA Ames Research Center, and IBM T.J. Watson Research Center. His research interests encompass high-performance computing, green supercomputing, accelerator and hybrid-based computing, and bioinformatics, for which he was bestowed a Distinguished Paper Award at the 2008 International Supercomputing Conference, three R&D 100 Awards, as well as recognition on HPCwire's Top People to Watch List. He leads the following large-scale projects: mpiBLAST, Green500, Supercomputing in Small Spaces, and MyVICE for K-8 Computer Science Pedagogy.