Multi-Node NVRAM in a Virtualized Environment

by
Herman M. Mutiso

Submitted to the
Department of Electrical Engineering and Computer Science
May 20, 2011
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The demand for increased data availability and reliability of storage systems has contributed to the
design and deployment of multi-node data storage clusters. This paper presents a simulator of one such
multi-node, multi-machine cluster. The simulator is architected by extending the NetApp, Inc. 2-node
cluster architecture to an N-node design. Data availability is provided by mirroring client requests to a
subset of peers in the multi-node cluster.

Using this simulator, this thesis explores the relationship between the number of peers that each node
mirrors to and the overall mirroring latency. This thesis also explores the performance cost incurred
when, in response to a mirroring request from a peer node, a node stores the mirrored data in non-
volatile storage before acknowledgement.

Using a workload consisting of multiple write requests to different nodes in the simulator, this thesis
finds that there exists a linear relationship between the number of mirroring peers in a cluster and the
resulting mirroring latency. Experiments using this workload also reveal a 40% increase in mirroring
latency when the mirroring requests are stored on peer nodes persistent storage as opposed to volatile
memory.
1 Introduction

The significant increase in data used by institutions in the world today has led to increased demand for highly reliable and highly available storage systems. These increased availability and reliability requirements contend with demand for high performance: typically, availability is provided using replication of data across two or more nodes, thereby adding network latencies and remote disk writes into an operation's critical path.

Partly because of these availability and performance requirements, data storage and management companies have provided multi-node cluster solutions, which potentially offer better performance and availability guarantees when compared to two-node cluster solutions.

This thesis describes a multi-node, multi-machine simulator, which models a data storage cluster in which data is stored on multiple nodes. Data availability is primarily achieved by mirroring client write requests to one or more peer nodes in the cluster, thereby guarding against data loss in the event of node failures. In particular, this thesis explores the effect of increasing the number of nodes in the cluster on mirroring performance. As part of this project, various experiments are conducted to measure the effect of increasing the number of mirroring peers each node has on mirroring performance.

The multi-machine multi-node simulator models storage solutions provided by NetApp, Inc. These storage solutions can broadly be categorized into two:

1. the 2-node cluster, and
2. the multi-node cluster (> 2 nodes)

1.1 2-node High Availability Cluster

NetApp, Inc.'s High Availability (HA) solution consists of two nodes in an active-active controller configuration. These two nodes both depend on Non-Volatile Random Access Memory (NVRAM) to cache data which have not yet been written to disk. Therefore, disk I/O need not run to completion before an acknowledgement is sent to a client, improving system performance. NVRAM also prevents data loss in the event of a node failure, since data that had not been written to disk before the failure can be retrieved by replaying NVRAM logs, thereby maintaining consistency of on-disk data. In the event
that node A in an HA pair fails, the surviving node B reads node A’s NVRAM logs and replays them, and assumes ownership of node A’s disks. This allows node B to continue serving clients whose requests were previously handled by node A. To ensure that node B has access to node A’s logs even after node A’s failure, node A’s NVRAM is mirrored to node B, and vice versa.

1.2 Multi-Node High Availability Cluster

There are significant benefits to be obtained from extending NetApp’s 2-node HA solution to a multi-node cluster solution. Compared to 2-node HA clusters, multi-node clusters offer increased resource utilization, improved performance and better availability guarantees.

As is the case with 2-node HA clusters, multi-node HA clusters require an infrastructure for sharing NVRAM across multiple nodes, to provide equal or better availability guarantees as a 2-node cluster.

In a multi-node cluster, the NVRAM mirroring model is crucial to the overall performance of the cluster. One approach involves use of a shared NVRAM appliance, in which all nodes in the multi-node cluster have read-write access to a shared NVRAM file. Alternatively, the distributed-NVRAM approach can be employed. When the NVRAM is distributed, each node can write only to its individual NVRAM file, and mirrors its NVRAM’s contents to a subset of its peer nodes in the cluster.

1.3 Contributions: Multi-Node Simulator

As part of this thesis project, we first designed and implemented a shared-NVRAM, single-machine, multi-node simulator, in which all nodes were implemented as individual processes running on a single machine.

We then implemented a distributed-NVRAM multi-machine simulator: each node ran as a process on a different machine. We opted for the distributed-NVRAM multi-machine implementation as it more accurately simulates performance of a physical multi-node cluster (the shared-NVRAM single-machine simulator does not account for network latencies and socket contention, for instance).

Using the distributed-NVRAM multi-machine simulator, we measured time required for each node to mirror its NVRAM data, the NVLOG, to another node in the cluster. We also measured mirroring performance as a function of the number of nodes in the simulated cluster. In addition,
we measured mirroring performance as a function of the number of peers mirrored to by each node. In summary, as part of this project, we set out to:

1. Investigate potential multi-node v-nvram architectures
2. Implement a multi-node cluster simulator
3. Examine performance of the simulated cluster by measuring:
   (a) remote write latencies as a function of the total number of nodes in the cluster.
   (b) remote write latencies as a function of the number of peers each node mirrors to.
   (c) cost of remote persistence.

Our experiments show that mirroring overhead can be kept constant despite the size of the mirroring request. In addition, preliminary results suggests that mirroring latency grows linearly in the number of peers that each node mirrors to. We also find that writing to non-volatile storage before acknowledging a mirroring request results in a 40% increase in mirroring latency.

2 Background Work

Prashanth Radhakrishnan, a member of NetApp, Inc’s Advanced Technology Group (ATG), conducted experiments measuring NVRAM and WAFL performance in a virtualized environment. The experiments revealed that:

1. Varying the virtual NVRAM (v-nvram) size has minimal impact on performance. Typically, disk utilization reaches 100% and disk I/O becomes the bottleneck.

2. WAFL’s logging of data in the NVRAM, nvlogging, is the chief determiner of performance. There are a number of v-nvram partitions that are persisted to disk: the TOC (V-NVRAM Table of Contents), RAID, SANOWN (Software Disk Ownership Information) and WAFL. Prashanth’s experiments found that nvlogging performance remains unaffected when nvlogging of other partitions of the V-NVRAM (RAID,
SANOWN and TOC) is (dis)enabled. Therefore, we concluded that a prototype which mirrored only WAFL logs was considered a good approximation of an actual VSIM or VSA.

2.1 2-node High Availability Cluster

NetApp's current HA solution consists of a 2-node cluster in an active-active configuration: during normal operation, both nodes process client requests, as opposed to a master-slave configuration in which only one of the nodes can be serving clients at any point in time. NetApp's HA solution is shown in Figure 1.

As can be seen in the diagram above, the cluster consists of two controllers (nodes), A and B, and a set of disks. These disks are divided into two groups: each controller owns 3 disks. A controller that owns a disk has...
primary read and write privileges over the disk. This is true of all disks in the cluster, except for special disks referred to as mailbox disks, which are used to communicate between the two controllers.

In order to provide high availability, when node A fails, node B should be able to continue serving (reading and writing) data stored on disks which were previously owned by node A. This availability requirement therefore implies that NetApp’s HA solution should:

1. ensure consistency of all data written to disk, and
2. prevent data loss during failover.

Data consistency is maintained by NetApp’s underlying Write Anywhere File Layout (WAFL) File system. Data loss is prevented in various ways, one of which is by use of NVRAM.

### 2.2 Non-Volatile Random Access Memory

Non-Volatile Random Access Memory, or NVRAM, is, as the name suggests, RAM that can hold data across power cycles. NVRAM has three primary uses on NetApp’s HA systems:

1. Cache data that has yet to be written to disk,
2. Log certain operations in selected subsystems, and
3. Prevent data loss during failover.

Using NVRAM as a data cache enables faster acknowledgment of clients’ write requests, thereby improving performance. The NVRAM hardware is also used to log operations in subsystems such as RAID, and can therefore be used for recovery in case of a subsystem failure. We refer to these logs, which are stored in the NVRAM, and mirrored between nodes during normal operation, as NVLOGs.

In addition to recovery, NVRAM guards against data loss during takeover and giveback, as is described in the next section.
2.3 Takeover and Giveback

During normal operation, both controllers in the active-active controller configuration periodically communicate via time-stamped heartbeat messages. These messages, which are written to special disks called mailbox disks, enable each controller to determine that its partner is still live and serving data to clients.

When these heartbeat messages are no longer detected, then a controller can assume its partner is no longer live. At this point, the live controller initiates takeover, a process during which the live controller assumes ownership of the failed controller’s disks and continues serving data to clients. After the failed controller has been brought back on-line, it can regain ownership of its data from its partner via a process referred to as giveback. For instance, if node A has not sent a heartbeat message to node B’s mailbox disks within a timeout period \( t \), then node B assumes that node A is no longer live, and initiates takeover of node A’s disks. After node A is back on-line, then it regains ownership of its disks from B during giveback. If the two nodes are partitioned by the network such that they cannot communicate with each other but can communicate with the disks, then the first node to access the disks will update disk-ownership information on the disk, causing the other node’s takeover process to fail. This ensures that only one of A and B has primary ownership- write access- of the disks at any point in time, thereby maintaining data consistency.

During takeover, controller B replays node A’s NVRAM and, if there are any pending disk writes, commits these writes to disk, thereby ensuring that no data is lost. Each node can access its partner’s NVRAM, even after a partner’s failure, because the NVRAM is mirrored between the two nodes: during normal operation, the NVRAM of each of the two controllers is evenly split into local and partner regions. The partner region is used to store a copy of the partner node’s NVRAM. Therefore, if a node A fails, then the live node B reads its copy of node A’s NVRAM, and commits the pending writes to disk as part of takeover.

2.4 Virtualized versus Physical Clusters

A strong motivation for implementing virtualized clusters is that they do not require as much specialized hardware as physical clusters; therefore, virtualized clusters invariably cost significantly less. In addition to lower cost, the
fact that the virtualized cluster is tailored to fit different platforms and virtualized environments means that the virtualized cluster can be configured so as to appropriately meet a customer's particular needs.

The ability to run multiple virtualized clusters on a smaller hardware set also means that cloud computing vendors stand to make economic savings by incorporating more virtualized clusters into their infrastructure.

However, the cost and specialized nature of NVRAM cards precludes its use in virtualized clusters, necessitating alternative implementations in virtualized environments.

2.5 Virtual NVRAM Layout

In a virtualized environment, physical NVRAM cards are emulated using dirty-first-flush-later semantics. In the dirtying phase, data are initially written to system memory. Upon reaching a commit point, then the appropriate data are flushed to the backing storage during the flushing phase. Therefore, in a virtualized environment, Virtual NVRAM, or V-NVRAM, consists of a combination of:

1. V-NVRAM memory and
2. V-NVRAM persistent storage.

3 Approach

We first constructed a set of hypotheses. Based on these hypotheses, we designed and implemented a multi-node mirrored v-nvram prototype that was capable of N-Way mirroring. By N-Way mirroring we refer to the ability of each node to mirror its v-nvram data to every other node in the simulated cluster. In the following subsections, we describe our hypotheses and objectives, and simulator architecture.

3.1 Hypotheses

We aimed to test the validity of the following initial hypotheses in the course of this project:
Remote write latency is linear in the size of the remote mirroring request: mirroring overhead should exhibit $O(1)$ time complexity with respect to the size of the data to be mirrored.

Remote write latency exhibits sub-linear time complexity with respect to the number of nodes mirrored to. We aimed to test the extent to which mirroring to multiple nodes can be parallelized. If this hypothesis holds, then it is possible to increase number of peers a node mirrors its data to without suffering a linear decrease in the node’s performance.

Remote write latency exhibits $O(1)$ time complexity with respect to the total number of nodes in the cluster, if the number of peers each node mirrors to is kept constant. We tested this hypothesis by running the simulator with each node mirroring to only one peer, and gradually increased the number of nodes in the cluster from 2 to 7. If this hypothesis holds, then one can add an arbitrary amount of nodes to a cluster, and not expect a decrease in performance, as long as the number of mirroring peers is kept constant.

3.2 Objectives

A consequence of the two-component $v$-$nvram$ layout is that NVLOG data can be stored either on a peer node’s $v$-$nvram$ persistent storage, or in the peer node’s $v$-$nvram$ memory. In other words, a multi-node cluster can be architected such that when node A sends a mirroring request to node B, node B can either directly write the data to its persistent storage, or store the data in its $v$-$nvram$ memory, and later flush the data to disk. This presents a trade-off between performance (writing NVLOG data to a peer node’s non-volatile storage adds disk I/O to the critical path) and availability (writing to a peer node’s volatile memory but not persistent storage introduces the risk of losing the $v$-$nvram$ data if the peer node crashes).

To quantify this trade-off, we investigated the cost of persistently mirroring $v$-$nvram$ data, as compared to storing the mirrored data only in memory before acknowledging the write request. We also investigated the trade-off between the availability offered by mirroring to a greater number of nodes, and the performance gained by reducing the number of nodes that each node mirrors to.

The above relationships would be captured by the following graphs:
1. latency of remote mirroring as a function of number of nodes mirrored to, with remote persistence.

2. latency of remote mirroring as a function of number of nodes mirrored to, without remote persistence.

4 Simulator Design

This section describes the design decisions and architecture of the multi-node, multi-machine, distributed-NVRAM simulator. Figure 2 represents a stylized diagram of a multi-machine, 4-node simulator:

![Figure 2: High level simulator design.](image)

The simulator was designed as a Linux user space application, with each node in the cluster implemented as an individual Linux process. The simulator can operate in two modes: single-machine and multi-machine mode.
In single-machine mode, all the nodes are started on a single machine; in the multi-machine version of the simulator, each process is started on an individual machine.

The simulator mimics virtualized environments’ active peer model, in which a peer node’s CPU is actively involved in receiving a mirroring request on a socket and writing it to the appropriate v-nvram location. This is as opposed to the unaware peer model that is afforded by Remote Direct Memory Access (RDMA) technology. RDMA allows a node to write directly to a peer node’s NVRAM, without having to signal an interrupt to the peer node’s CPU. We opted against the unaware peer approach as it required specialized hardware that is currently not supported on NetApp’s virtual environments.

4.1 Inter-Node Communication

We considered three approaches for implementing sockets[1]:

1. a forking server model, in which a new thread/process is created to handle each new connection,

2. an iterating server model, in which only one socket is processed at a time, and

3. a concurrent server model, in which the select() API is used to poll a set of socket file descriptors to establish which is ready for processing

We adopted the concurrent server model because of its better performance compared to the iterating server model, and because it required fewer context switches and did not result in as much thread complexity as the forking server model [2].

In the concurrent server model, each node has one server socket and as many client sockets as there are nodes being mirrored to. The server socket is used to listen to incoming connections, and to receive any data sent from remote nodes. The client sockets are used to initiate connection requests to remote nodes, and to send remote v-nvram write requests (during v-nvram mirroring).

Before accepting client requests, each node spawns a server thread that creates a server socket. This socket uses the select() API to listen to incoming connection requests. Upon accepting a connection, the newly created socket
file descriptor is added to a set of sockets to be listened to. This set is then continuously polled, and when incoming data is received on any of the set's sockets, proc.rcvd.data() is called. proc.rcvd.data() checks that the incoming data is correctly composed, acts according to the incoming data (dirties and/or flushes the v-nvram region of the requesting node) and returns an ack to the requesting node. The ack which is sent back is the same xfer.id sent as part of the incoming request.

4.1.1 Communication Protocol

This subsection describes the means by which any two nodes in the simulator communicate. Three types of messages are used for Inter-Node communication:

1. An ack message from node A's server socket to node B's client socket. This ack message informs node B's client socket that node A's socket has received and processed node B's connection request to completion.

2. A done.msg message from node A's server socket to node B's client socket. This message informs node B that node A has run to completion, and is ready to shutdown. The done.msg therefore ensures that nodes initiate a shutdown sequence only after every other node has run to completion, thereby precluding unacknowledged mirroring requests due to clean shutdowns.

3. The send.buff, which is used to send remote mirroring data to a peer node. The send.buff is constructed as follows:

   (a) xfer.id is the transfer ID of the remote mirroring request. If the request is successfully received and processed by the node receiving the request, then the receiving node sends back the xfer.id as an acknowledgement of successful processing and logging.

   (b) Start.offset and End.offset indicate where in the remote node's region the payload.data should be written.

   (c) Region indicates which region on the remote node is to be written to. Each node's v-nvram is split into regions, with each region containing the NVLOGs of one node in the cluster. Therefore, if a node is mirroring to m peers, its v-nvram file is split into m+1
regions, one for the node itself, and the other $m$ regions for the mirroring peers.

(d) *Payload.data* is the data that is to be written to the remote node’s *v-nvram* memory and/or the *v-nvram* persistent storage.

Figure 3 shows the layout of the *send_buff*. The aforementioned *send_buff* design implies that the size of *payload.data* is predetermined and known to all nodes. If varying size remote writes are required, the structure of the *send_buff* can be modified to incorporate an additional field, *bytes.sent*, so that the recipient node is aware of how many bytes to expect. The write size was varied when running the first experiment (which was intended to determine the relationship between the size of a mirroring request and mirroring latency). The remaining experiments were run with a constant mirroring request size.

### 4.2 Virtual NVRAM

We considered implementing *v-nvram* as a shared appliance, with the following design: the *v-nvram* file lives on an NFS server. Each node in the cluster NFS mounts this *v-nvram* file, such that all nodes in the cluster have read-write access to this shared *v-nvram* file. A shared *v-nvram* model promises certain benefits:

- Minimizes inter-node communication, eliminating the inter-node network from several critical paths.
• All NVLOGs are co-located on the same NFS server, making the system easier to design and implement.

However, a shared $v$-$nvram$ appliance presents certain difficulties:

• A locking scheme is now required to prevent concurrent modifications to the $v$-$nvram$ file by multiple nodes

• The serialization of modifications to the $v$-$nvram$ file results in additional latency that does not scale well with the number of nodes in the cluster.

• The shared $v$-$nvram$ module is now a single point of failure, and so requires redundant $v$-$nvram$ files living on redundant NFS servers.

These disadvantages dissuaded us from the shared $v$-$nvram$ model; our multi-node simulator therefore implements distributed NVRAM.

We opted for mmap-ed files instead of explicit disk I/O due to the atomicty guarantees provided by the mmap API, and for ease of implementation.

The simulator’s virtual NVRAM (V-NVRAM) is implemented as a combination of:

1. V-NVRAM memory and

2. V-NVRAM persistent storage.

The $v$-$nvram$ memory is a dedicated region of system memory that is used to temporarily hold NVLOG data before the data are flushed to on-disk $v$-$nvram$ persistent storage, which is implemented as an exclusive on-disk mmap-ed file. The $v$-$nvram$ persistent storage must be at least as large as $v$-$nvram$ memory. Each simulated node is associated with both a $v$-$nvram$ memory and $v$-$nvram$ persistent storage. Each node can read from and write to its $v$-$nvram$ memory and $v$-$nvram$ persistent storage, but cannot directly access any other node’s $v$-$nvram$ memory or backing file. This is the distributed-NVRAM model.

4.2.1 Virtual NVRAM Partitioning

Each node’s $v$-$nvram$ memory is partitioned into node_count regions, with all regions having equal size; there are as many regions as there are nodes
in the simulation. Therefore \textit{nv.region.count} necessarily equals \textit{node.count}. The \textit{v-nvram}'s first region is dedicated to holding NVLOG data of the local node. All other regions of the \textit{v-nvram} are dedicated to holding mirrored NVLOG data received from peer nodes in the cluster. A node is prevented from accessing the wrong region by a combination of region-specific locks and memory access checks. When processing a mirroring request from a peer node, the local node writes only to non-local \textit{v-nvram} regions. The \textit{vnvram.region.t} structure is used to hold timing statistics associated with all regions of a node's \textit{v-nvram} memory.

4.3 Workload Generator

The workload generator uses a \textit{client.file} as a source of NVLOG data: data read from the \textit{client.file} are written to \textit{v-nvram} memory via a call to the function \textit{vnvram.dirty()}, and later flushed to the \textit{nv.backing.file} by calling the function \textit{vnvram.flush()}. Writes to \textit{v-nvram} memory have a fixed size throughout each simulation. Therefore, obtaining performance measurements with varying write sizes requires multiple runs of the simulation.

An alternative to the \textit{client.file} approach would be to use a file system workload as a source of writes, as this automates more of the simulation, and involves a wider variety of tests. However, we did not come across a performance workload whose writes can be diverted to memory. Writing directly to an on-disk file would break the \textit{dirty-first-flush-later} semantics of \textit{v-nvram}; we therefore opted against this approach.

Another approach that was considered was the creation of a RAM-based file system such as tempfs, combined with using a standard workload on these memory-based files. This approach was not pursued because of the complexity involved in instrumenting writes to tempfs so as to get the desired measurements and statistics, especially when flush sizes were not known \textit{a priori}.

5 Implementation

5.1 Simulator Setup

Before initializing the simulator, one should set \textit{node.count} to the desired value (default value is 2). Note that the array variables \textit{nv.backing.files},
client_files and servnames might have to be updated in order to cater to the desired number of nodes. The client application is started with two arguments: node_id and rmtPersist. node_id is a value between 0 and node_count, and is used by the node in setting up inter-node communication. rmtPersist is set to 0 or 1, and determines whether acknowledging a remote node’s mirroring request must be preceded by a flush to disk or not. After these variables have been set, the simulator is initialized using the following steps:

1. Node initializes its v-nvram. Node initialization involves opening a client_file from which to read NVLOG data, an nv_backing_file to serve as the vnvram persistent storage, and a stat_file to write statistics gathered during the simulation. In addition to creation of these files, the nv_backing_file is mapped to system memory using the mmap() system call, and partitioned into regions, for both the local and remote nodes’ NVLOGs. The v-nvram is partitioned such that there is a region dedicated to the local node, and another region for every node that is mirrored to. Therefore, the following are invariants of the system:

   (a) \( \text{nv.region.count} = \text{nodes.mirrored.to} + 1 \)
   (b) \( \text{nv.region.count} \leq \text{node.count} > \text{nodes.mirrored.to} \)

   Node initialization also involves initializing multiple data structures which are used for logging simulation statistics.

2. Node initializes a server socket thread. Within this server socket thread, the node creates a server socket, and uses this socket to listen to incoming connections from remote nodes. Once a connection request is received, the node sends back an acknowledgment of the connection.

3. Node spawns a client request thread. This thread reads data from a client_file, and sends this data to the v-nvram for logging. The client request thread therefore simulates client write requests to the v-nvram.

5.2 Dirtying V-NVRAM Memory

Upon receiving a write request from the workload generator, a lock for the region of v-nvram memory that is being dirtied is obtained. The write’s start and end offsets are then checked to ensure that they fall within the bounds of the region. If this check passes, the vnvram_log, dirty_low and dirty_high
values for the region are updated. The contents of v-nvram memory are then
dirtied, and, finally, the lock is released. Pseudo-code for the \texttt{vnvram\_dirty}
function follows.

\begin{algorithm}
\caption{vnvram\_dirty}
\begin{algorithmic}
\State acquire \texttt{region.lock}
\If{$\texttt{region.low} \leq \texttt{dirty.low} \leq \texttt{dirty.high} \leq \texttt{region.high}$}
\State add log entry to \texttt{vnvram.log}
\State \texttt{region.dirty.low} $\leftarrow \min(\texttt{write.low}, \texttt{region.dirty.low})$
\State \texttt{region.dirty.high} $\leftarrow \max(\texttt{write.high}, \texttt{region.dirty.high})$
\State write \texttt{data} to v-nvram memory
\State record dirty.size; dirty.duration
\State release \texttt{region.lock}
\EndIf
\end{algorithmic}
\end{algorithm}

5.3 Flushing to Local and Remote Persistent Storage

After several client write requests, which are logged to the serving node’s
v-nvram, the v-nvram memory is flushed to the on-disk \texttt{nv.backing.file}. In
order to flush its data to disk, a node first spawns a single thread to flush
to local disk. Another \texttt{nodes.mirrored.to} threads are spawned, with each
thread sending a mirroring request to one of the peer nodes. The generation
of the \texttt{local.flush.thread} and the \texttt{rmt.write.threads} is done
by the function \texttt{copy.to.vnvram}. First, we present the local flushing implementation, and
then the remote mirroring implementation.

5.4 Flushing to Local Persistent Storage

The local flush thread executes the \texttt{vnvram.flush} function and then exits.
The \texttt{vnvram.flush} function firsts checks that the region to be flushed has
been initialized, then acquires the region’s lock, and updates the \texttt{dirty.low}
and \texttt{dirty.high} pointers to their default values (\texttt{nv.high.limit} and \texttt{nv.low.limit},
respectively). This update serves as an indicator that the region is clean.

After acquisition of the region’s lock, the \texttt{vnvram.log} is updated, and the
flush effected using an \texttt{msync} system call. The duration and size of the write
is then recorded in a separate file. Finally, the region’s lock is released.
Note that each node periodically flushes its entire v-nvram memory to disk. This is accomplished by having vnvram_flush called with the region argument as flush_all. In this case, all the regions’ locks need to be taken before the flush, and instead of the msync() system call, the fsync() system call is used, thereby ensuring the entire on-disk file is synchronized with the in-memory mapping. The pseudo-code for the vnvram_flush function is provided below:

Algorithm 2 vnvram_flush

```python
if region_to_flush = flush_all then
    for region = 0 to nv_region_count do
        acquire region.lock
    end for
    fsync(nv_low_limit, nv_high_limit) // flush all regions to disk
    record flush.size; flush.duration
    for all region in regions do
        region.low <- nv_high_limit
        region.high <- nv_low_limit
        release region.lock
    end for
else
    acquire region.lock
    msync(region.low, region.high) // flush single region to disk
    record flush.size; flush.duration
    region.low <- nv_high_limit
    region.high <- nv_low_limit
    release region.lock
end if
```

5.5 Mirroring To Remote Nodes

The rmt_write_threads run the function nv_rmt_write() and exit once the function completes. The nv_rmt_write() function first creates a char array of data to send over to the remote node. This array is marshaled as described in section 4.1.1. The data are then sent to the node’s mirroring peers.

Upon receipt of the remote mirroring request, the remote node calls proc_rcvd_data(), which unmarshals the array, checks whether the data is
as expected, and then either dirties the partner’s v-nvram region or flushes it to disk, depending on whether remote mirroring should persist the data to disk or not (rmt Persist). If proc_revd_data() completes without incident, then the recipient node sends an ack back to the requesting node, with the same xfer_id as the initial receive request. After the requesting node sends a remote mirroring request, it waits for an ack from the recipient node. Upon receipt of this ack, the node checks if the ack is valid. If the check passes, the node logs the time taken to complete the remote mirroring request, and then exits.

6 Experiments

We measured the latency of the following operations with the distributed-NVRAM, multi-node, multi-machine prototype:

1. Remote write latency as a function of remote write size.

2. Remote write latency as a function of nodes_mirrored_to, with remote persistence.

3. Remote write latency as a function of nodes_mirrored_to, without remote persistence.

4. Remote write latency as a function of the number of nodes in the cluster, with remote persistence.

5. Remote write latency as a function of the number of nodes in the cluster, without remote persistence. The number of nodes in the prototype was varied from 2 to 8. Each node in this experiment mirrored to one peer node, so as to keep mirroring overhead constant despite increasing the number of nodes in the prototype.

In all these experiments, each node used an mmap-ed file for its v-nvram. Mirroring latency is computed as the time taken for a mirroring node to send out all its mirroring requests and receive the corresponding acknowledgements from each of its mirroring peers. The last two experiments were conducted to determine whether the total number of nodes in the cluster, node_count, impacted the latency of a remote write, even when the total
number of nodes was larger than the number of peers each node was mirroring to.

The above experiments were performed on a subset of NetApp’s Linux servers. In the multi-node experiments therefore, a node was created on a unique server, and the experiments run once all the nodes had been initialized and setup communication sockets with all other mirroring nodes.

Because the NetApp servers were running other jobs concurrently, the experiments were susceptible to increased latencies due to contention with other processes running on these machines. For instance disk accesses on remote nodes might have been delayed if other disk I/O-intensive processes were running concurrently. We mitigated this by reporting minimum mirroring latencies observed during these experiments.

7 Results

7.1 Mirroring Latency vs. Remote Write Size

This experiment measured the effect of different remote write sizes (size of a single mirroring request) on mirroring times, thereby characterizing the overhead of remote writes. We used a two node simulator (each node mirroring to one peer node) to ensure no additional overhead was incurred due to contention for the same server socket. Each of the two nodes completed 1,000 remote mirroring requests for every remote write size. We used 11 different flush sizes, ranging from 4 KB to 4 MB, resulting in 11,000 total mirroring requests for each node. The size of the v-nvram file was 256 MB.

Figure 4 is a plot of average remote write latencies as a function of the remote write size, for both of the nodes in the experiment.

The results for the two nodes were plotted, and their corresponding correlation coefficient ($R^2$) value obtained. The data suggested a linear relationship with $R^2$ values of 0.981 and 0.930 for each of the two nodes, respectively. This experiment was repeated with three machines which were not running any additional processes; the results also suggested a linear relationship, although with fewer data points. This therefore implies that mirroring overhead is constant with respect to the size of the actual mirroring request, confirming our first hypothesis.
7.2 Mirroring Latency vs. Number of Nodes Mirrored To, With Remote Persistence

This experiment measured the effect of increasing nodes_mirrored_to on remote write latencies, with remote persistence. We used 8 nodes in this experiment, and each node made use of an mmap-ed v-nvram file. We varied the number of peers that each node mirrored to from 1 to 7. For each value of nodes_mirrored_to, each node sent 8,000 mirroring requests of size 512KB to all its mirroring peers.

Figure 5 graphs the minimum mirroring latency across all the nodes for each run of the experiment. The best fit line shown has an $R^2$ value of 0.9751, suggesting that increasing the number of peers that each nodes mirrors to results in a linear increase in the mirroring latency.

These preliminary results suggest a linear relationship but are not conclusive enough to verify our second hypothesis, which postulated a sub-linear
relationship between the number of peers a node mirrors to and the resulting mirroring latency. We expected that the relationship would be sub-linear because each of the mirroring requests occurred simultaneously (each request was handled by a unique thread) and each request culminated in a write to a unique disk. We suspect that the contention described above might have been responsible for the increased latency, but due to time and resource constraints, we were not able to conclusively verify or disprove our initial hypothesis.

7.3 Mirroring Latency vs. Number of Nodes Mirrored To, Without Remote Persistence

This experiment was identical to the previous experiment, except peers did not write data to persistent storage before acknowledging a mirroring request.

Figure 6 graphs the minimum mirroring latency across all the nodes in a cluster for a specific value of \( nodes\_mirrored\_to \). The results show a conclusive
linear relationship \((R^2 = 0.9961)\) between mirroring latency and the number of nodes mirrored to. This suggests that, contrary to our expectation, the potential for parallelizing mirroring requests to different nodes is not as high as was anticipated.

As was the case with experiment 2, the experimental environment might have confounded the results. We speculate that the linear relationship observed in experiment 3 was due primarily to the increased latency observed in the shared network and due to resource contention on the shared servers which were used for the experiment. We were, however, unable to pursue these secondary hypotheses further due to time constraints.

### 7.4 Mirroring Latency vs. Total Number of Nodes, With Remote Persistence

This experiment measured the relationship between the number of nodes in a cluster, and the latency of remote writes.
We hypothesized that the latency of mirroring requests should only depend on the number of peers each node mirrors to (nodes\_mirrored\_to), but not on the total number of nodes in the cluster (node\_count). We therefore kept the value of nodes\_mirrored\_to constant (1), and varied node\_count from 2 to 8.

Each of the 8 nodes used an mmap-ed file of size 256 MB for its v-nvram, and made 8,000 mirroring requests to its mirroring peer. The size of each mirroring request was kept constant at 512 KB. The minimum mirroring latencies for each value of node\_count are captured in figure 7.

![Mirroring Latency vs. Node Count, with remote persistence](image)

Figure 7: Relationship between mirroring latency and node\_count, with persistence on peer nodes.

The results indicate a linear relationship between the number of nodes in the cluster and the overall mirroring latency.

This provides evidence against our third hypothesis, which postulated that in the steady state, the number of nodes in the cluster should not impact mirroring latencies if the number of peers that each node was mirroring to is kept constant.

We believe that this linear relationship was observed because nodes in the
simulation communicated with two nodes, even though each node mirrored to one peer. This was because a node's mirroring peer was elected in a round-robin fashion. For instance, node B received remote mirroring requests from node A, while sending its own mirroring requests to node C. Given that the nodes were running identical workloads, it is likely that this caused contention for node B's server socket, thereby introducing additional delays that produced the above results. In addition, we suspect that the shared network and resource contention on the servers on which these experiments were run contributed to the above results. We were not able to tests these secondary hypotheses due to time constraints.

7.5 Mirroring Latency vs. Total Number of Nodes, Without Remote Persistence

This experiment was identical to experiment 4, except mirroring happened without persistence on the remote node.

The value of nodes.mirrored.to was kept constant at 1, but the node_count was varied from 2 to 8. Each of the 8 nodes in the simulation used an mmap-ed file of size 256 MB for its v-nvram, and made 8,000 mirroring requests to its mirroring peer. The size of each mirroring request was kept constant at 512 KB. Results for each node are captured in figure 8.

The results do not provide conclusive evidence as to the relationship between mirroring latency and node count when mirroring requests are not persisted remotely.

We believe this experiment was particularly sensitive to latencies due to resource contention and shared networks. This is because the mirroring request’s critical path did not include any disk writes when NVLOG data was not persisted on peer nodes. This meant that any additional unintended network latencies significantly increased the mirroring latencies observed in the experiment. This was not the case with the previous experiment, in which mirroring latencies dominated arbitrary network overhead in a shared network.

Access to an isolated network of machines would allow a design of an experiment to test this renewed hypothesis, and potentially provide conclusive evidence to verify or disprove the correlation between the number of nodes in a cluster and the resulting mirroring latencies.
7.6 Cost of Remote Persistence

We consider the cost of remote persistence as the percentage increase in mirroring latencies when remote persistence is added into a simulation.

In particular, we compare experiments 2 and 3, in which the total number of nodes in each simulation was kept constant, but the number of nodes each peer was mirroring to was varied from 2 to 8.

The table below shows the difference in mirroring latencies, and the percentage change between mirroring with remote persistence, and mirroring without remote persistence.

The mean, median and maximum percentage increases were 39.187%, 38.105% and 66.876%, respectively. This suggests an average increase of mirroring latency by around 40% when mirroring requests are persisted remotely.
7.7 High Availability Considerations

Availability guarantees are significantly higher when remote persistence is integrated into the system. Specifically, if we assume that:

- node failures are independent,
- each node has a probability \( p \) of failure,
- mirroring requests to each node are stored in non-volatile storage,
- the nodes are in an N-Way cluster of \( n \) nodes and
- each node mirrors its data to \( m \) other nodes,

then the probability of data loss if mirroring requests are stored on permanent storage is \( \left( \frac{1}{p} \right)^m \), since all of the \( m \) nodes need to fail in order for the data to be lost.

Without remote persistence, however, there are numerous corner cases that result in weaker availability guarantees when compared to mirroring with remote persistence. An example of this is when a node A sends a mirroring
request to a peer node B, and then node A crashes, closely followed by a failure of node B. If node B had not written node A’s mirroring request to disk before node B crashed, then data loss results.

8 Future Work

The following extensions would improve the accuracy with which the current prototype can predict N-Way mirroring behavior:

Remote Direct Memory Access Currently, mirroring NVLOGs requires the active peer model previously described. An RDMA engine allows mirroring requests to be processed without involving the CPU of the receiving node: the unaware peer model. This unaware peer model, if correctly implemented, affords lower latency mirroring. We considered emulating the RDMA unaware peer model using a mounted NFS file, but the resulting NFS latencies and contention for a single file would result in even poorer performance, thereby inaccurately modeling the unaware peer approach.

Use of DATA ONTAP’s stack Implementing an N-Way mirroring prototype using DATA ONTAP’s stack (DATA ONTAP is a Unix based operating system used on NetApp’s data management and storage solutions) will provide a more accurate prediction of N-Way mirroring performance on NetApp’s HA storage solutions.

Variable number of mirroring peers Also of interest is the overall performance of a cluster when nodes in the cluster do not mirror to the same number of peers, and the resulting availability implications of this approach.

9 Conclusion

The linear increase in mirroring latency with increase in the number of mirroring nodes suggests that a real trade-off must be made between data availability and desired performance. This suggests that applications running on top of an N-Way mirroring-based storage system must tolerate the reduced performance that accompanies increased data reliability.
However, use of hardware that allows RDMA in multi-node clusters promises to significantly mitigate the cost of additional mirroring.

10 Acknowledgements

Without the support, advice and guidance of the following NetApp engineers, this project would not have been possible. I am sincerely grateful to have had the opportunity to work with and learn from each of them over the duration of this project.

- David Robinson
- Sue Coatney
- Narayan Venkat
- Prashanth Radhakrishnan
- Don Cleland
- Poonam Dhavale
- Steve Watanabe
- Kalaivani Arumugham
- Deepa Ramesh
- Steve Miller

I would also like to express my gratitude to Professor Frans Kaashoek, my thesis supervisor, and Professor Samuel Madden, the VI-A NetApp Professor, for their support, mentoring and continuous feedback, as a result of which this thesis project was realized.
11 Bibliography

References
