Panache: A Parallel File System Cache for Global File Access

Marc Eshel Roger Haskin Dean Hildebrand Manoj Naik Frank Schmuck
Renu Tewari
IBM Almaden Research
{eshel, roger, manoj, schmuck}@almaden.ibm.com, {dhildeb, tewarir}@us.ibm.com

Abstract

Cloud computing promises large-scale and seamless access to vast quantities of data across the globe. Applications will demand the reliability, consistency, and performance of a traditional cluster file system regardless of the physical distance between data centers.

Panache is a scalable, high-performance, clustered file system cache for parallel data-intensive applications that require wide area file access. Panache is the first file system cache to exploit parallelism in every aspect of its design—parallel applications can access and update the cache from multiple nodes while data and metadata is pulled into and pushed out of the cache in parallel. Data is cached and updated using pNFS, which performs parallel I/O between clients and servers, eliminating the single-server bottleneck of vanilla client-server file access protocols. Furthermore, Panache shields applications from fluctuating WAN latencies and outages and is easy to deploy as it relies on open standards for high-performance file serving and does not require any proprietary hardware or software to be installed at the remote cluster.

In this paper, we present the overall design and implementation of Panache and evaluate its key features with multiple workloads across local and wide area networks.

1 Introduction

Next generation data centers, global enterprises, and distributed cloud storage all require sharing of massive amounts of file data in a consistent, efficient, and reliable manner across a wide-area network. The two emerging trends of offloading data to a distributed storage cloud and using the MapReduce [11] framework for building highly parallel data-intensive applications, have highlighted the need for an extremely scalable infrastructure for moving, storing, and accessing massive amounts of data across geographically distributed sites. While large cluster file systems, e.g., GPFS [26], Lustre [3], PanFS [29] and Internet-scale file systems, e.g., GFS [14], HDFS [6] can scale in capacity and access bandwidth to support a large number of clients and petabytes of data, they cannot mask the latency and fluctuating performance of accessing data across a WAN.

Traditionally, NFS (for Unix) and CIFS (for Windows) have been the protocols of choice for remote file serving. Originally designed for local area access, both are rather "chatty" and therefore unsuited for wide-area access. NFSv4 has numerous optimizations for wide-area use, but its scalability continues to suffer from the "single server" design. NFSv4.1, which includes pNFS, improves I/O performance by enabling parallel data transfers between clients and servers. Unfortunately, while NFSv4 and pNFS can improve network and I/O performance, they cannot completely mask WAN latencies nor operate during intermittent network outages.

As "storage cloud" architectures evolve from a single high bandwidth data-center towards a larger multi-tiered storage delivery architecture, e.g., Nirvanix SDN [7], file data needs to be efficiently moved across locations and be accessible using standard file system APIs. Moreover, for data-intensive applications to function seamlessly in "compute clouds", the data needs to be cached closer to or at the site of the computation. Consider a typical multi-site compute cloud architecture that presents a virtualized environment to customer applications running at multiple sites within the cloud. Applications run inside a virtual machine (VM) and access data from a virtual LUN, which is typically stored as a file, e.g., VMware's .vmdk file, in one of the data centers. Today, whenever a new virtual machine is configured, migrated, or restarted on failure, the OS image and its virtual LUN (greater than 80 GB of data) must be transferred between sites causing long delays before the application is ready to be online. A better solution would store all files at a central core site and then dynamically cache the OS image and its virtual LUN at an edge site closer to the physical machine. The machine hosting the VMs (e.g., the ESX server) would connect to the edge site to access the virtual LUNs over NFS while the data would move transparently between the core and edge sites on demand. This enormously simplifies both the time and complexity of configuring new VMs and dynamically moving them across a WAN.

Research efforts on caching file system data have mostly been limited to improving the performance of a single client machine [18, 25, 22]. Moreover, most available solutions are NFS client based caches [15, 18]

and cannot function as a standalone file system (without network connectivity) that can be used by a POSIX-dependent application. What is needed is the ability to pull and push data in parallel, across a wide-area network, store it in a scalable underlying infrastructure while guaranteeing file system consistency semantics.

In this paper we describe Panache, a read-write, multi-node file system cache built for scalability and performance. The distributed and parallel nature of the system completely changes the design space and requires re-architecting the entire stack to eliminate bottlenecks. The key contribution of Panache is a *fully parallelizable* design that allows every aspect of the file system cache to operate in parallel. These include:

- *parallel ingest* wherein, on a miss, multiple files and multiple chunks of a file are pulled into the cache in parallel from multiple nodes,
- *parallel access* wherein a cached file is accessible immediately from all the nodes of the cache,
- *parallel update* where all nodes of the cache can write and queue, for remote execution, updates to the same file in parallel or update the data and metadata of multiple files in parallel,
- *parallel delayed data write-back* wherein the written file data is asynchronously flushed in parallel from multiple nodes of the cache to the remote cluster, and
- *parallel delayed metadata write-back* where all metadata updates (file creates, removes etc.) can be made from any node of the cache and asynchronously flushed back in parallel from multiple nodes of the cache. The multi-node flush preserves the order in which dependent operations occurred to maintain correctness.

There is, by design, no single metadata server and no single network end point to limit scalability as is the case in typical NAS systems. In addition, all *data and metadata updates* made to the cache are *asynchronous*. This is essential to support WAN latencies and outages as high performance applications cannot function if every update operation requires a WAN round-trip (with latencies running from 30ms to more than 200ms).

While the focus in this paper is on the parallel aspects of the design, Panache is a fully functioning POSIX-compliant caching file system with additional features including disconnected operations, persistence across failures, and consistency management, that are all needed for a commercial deployment. Panache also borrows from Coda [25] the basic premise of conflict handling and conflict resolution when supporting disconnected mode operations and manages them in a clustered setting. However, these are beyond the scope of this paper. In this paper, we present the overall design and implementation of Panache and evaluate its key features with multiple workloads across local and wide area networks.

The rest of the paper is organized as follows. In the next two sections we provide a brief background of pNFS and GPFS, the two essential components of Panache. Section 4 provides an overview of the Panache architecture. The details of how synchronous and asynchronous operations are handled are described in Section 5 and Section 6. Section 7 presents the evaluation of Panache using different workloads. Finally, Section 8 discusses the related work and Section 9 presents our conclusions.

2 Background

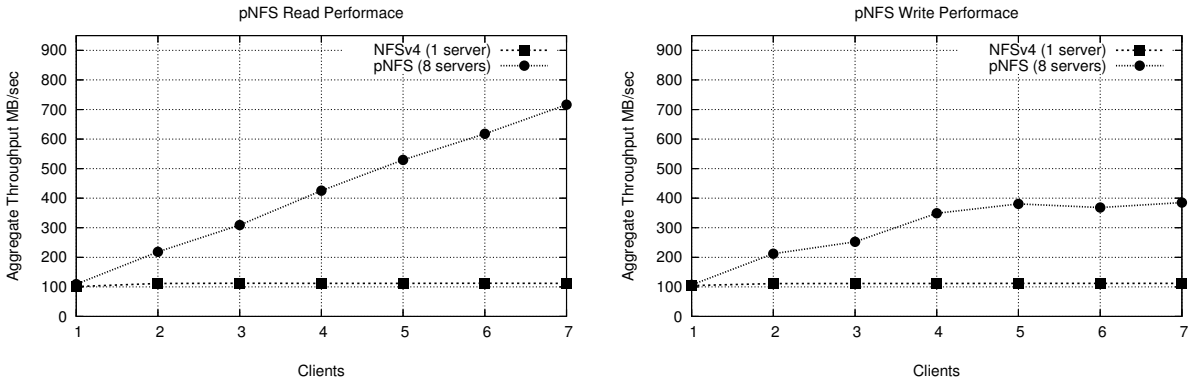
In order to better understand the design of Panache let us review its two basic components: GPFS, the parallel cluster file system used to store the cached data, and pNFS, the nascent industry-standard protocol for transferring data between the cache and the remote site.

GPFS: General Parallel File System [26] is IBM's high-performance shared-disk cluster file system. GPFS achieves its extreme scalability through a shared-disk architecture. Files are wide-striped across all disks in the file system where the number of disks can range from tens to several thousand disks in the largest GPFS installations. In addition to balancing the load on the disks, striping achieves the full throughput that the disk subsystem is capable of by reading and writing data blocks in parallel.

The switching fabric that connects file system nodes to disks may consist of a storage area network (SAN), e.g., Fibre Channel, iSCSI, or, a general-purpose network by using I/O server nodes. GPFS uses distributed locking to synchronize access to shared disks where all nodes share responsibility for data and metadata consistency. GPFS distributed locking protocols ensure file system consistency is maintained regardless of the number of nodes simultaneously reading from and writing to the file system, while at the same time allowing the parallelism necessary to achieve maximum throughput.

pNFS: The pNFS protocol, now an integral part of NFSv4.1, enables clients for direct and parallel access to storage while preserving operating system, hardware platform, and file system independence [16]. pNFS clients and servers are responsible for control and file management operations, but delegate I/O functionality to a storage-specific layout driver on the client.

To perform direct and parallel I/O, a pNFS client first requests layout information from a pNFS server. A layout contains the information required to access any byte of a file. The layout driver uses the information to translate I/O requests from the pNFS client into I/O requests



(a) pNFS Reads (b) pNFS Writes

Figure 1: **pNFS Read and Write performance.** *pNFS performance scales with available hardware and network bandwidth while NFSv4 performance remains constant due to the single server bottleneck.*

directed to the data servers. For example, the NFSv4.1 file-based storage protocol stripes files across NFSv4.1 data servers, with only READ, WRITE, COMMIT, and session operations sent on the data path. The pNFS metadata server can generate layout information itself or request assistance from the underlying file system.

3 pNFS for Scalable Data Transfers

Panache leverages pNFS to increase the scalability and performance of data transfers between the cache and remote site. This section describes how pNFS performs in comparison to vanilla NFSv4.

NFS and CIFS have become the de-facto file serving protocols and follow the traditional multiple client–single server model. With the single-server design, which binds one network endpoint to all files in a file system, the back-end cluster file system is exported by a single NFS server or multiple independent NFS servers.

In contrast, pNFS removes the single server bottleneck by using the storage protocol of the underlying cluster file system to distribute I/O across the bisectional bandwidth of the storage network between clients and data servers. In combination, the elimination of the single server bottleneck and direct storage access by clients yields superior remote file access performance and scalability [16].

Figure 2 displays the pNFS-GPFS architecture. The nodes in the cluster exporting data for pNFS access are divided into (possibly overlapping) groups of state and data servers. pNFS client metadata requests are partitioned among the available state servers while I/O is distributed across all of the data servers. The pNFS client requests the data layout from the state server using a LAYOUTGET operation. It then accesses data in parallel by using the layout information to send NFSv4 READ and WRITE operations to the correct data servers. For writes, once the I/O is complete, the client

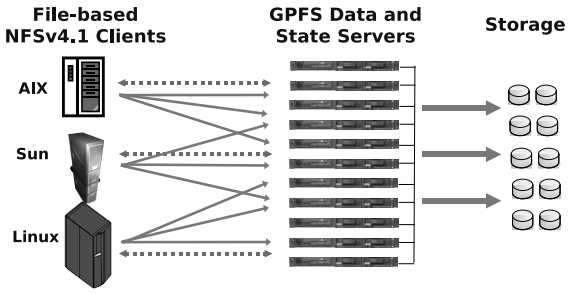
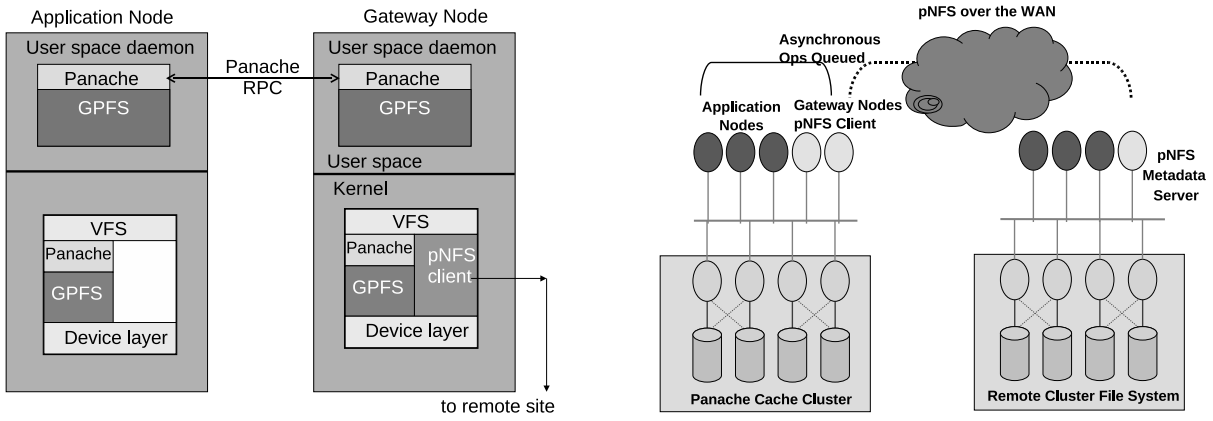


Figure 2: **pNFS-GPFS Architecture.** *Servers are divided into (possibly overlapping) groups of state and data servers. pNFS/NFSv4.1 clients use the state servers for metadata operations and use the file-based layout to perform parallel I/O to the data servers.*

sends an NFSv4 COMMIT operation to the state server. This single COMMIT operation flushes data to stable storage on every data server. The underlying cluster file system management protocol maintains the freshness of NFSv4 state information among servers.

To demonstrate the effectiveness of pNFS for scalable file access, Figures 1(a) and 1(b) compare the aggregate I/O performance of pNFS and standard NFSv4 exporting a seven server GPFS file system. GPFS returns a file layout to the pNFS client that stripes files across all data servers using a round-robin order and continually alternates the first data server of the stripe. Experiments use the IOR micro-benchmark [2] to increase the number of clients accessing individual large files. As the number of NFSv4 clients accessing a single NFSv4 server is increased, performance remains constant. On the other hand, pNFS can better utilize the available bandwidth. With reads, pNFS clients completely saturate the local network bandwidth. Write throughput ascends to 3.8x of standard NFSv4 performance with five clients before reaching the limitations of the storage controller.



(a) Node Block Diagram

(b) Cache Cluster Architecture

Figure 3: **Panache Caching Architecture.** *(a) Block diagram of an application and gateway node. On tje gateway node, Panache communicates with the pNFS client kernel module through the VFS layer. The application and gateway nodes communicate via custom RPCs through the user-space daemon. (b) The cache cluster architecture. The gateway nodes of the cache cluster act as pNFS/NFS clients to access the data from the remote cluster. The application nodes access data from the cache cluster.*

4 Panache Architecture Overview

The design of the Panache architecture is guided by the following performance and operational requirements:

- Data and metadata read performance, on a cache hit, matches that of a cluster file system. Thus, reads should be limited only by the aggregate disk bandwidth of the local cache site and not by the WAN.
- Read performance, on a cache miss, is limited only by the network bandwidth between the sites.
- Data and metadata update performance matches that of a cluster file system update.
- The cache can operate as a standalone fileserver (in the presence of intermittent or no network connectivity), ensuring that applications continue to see a POSIX compliant file system.

Panache is implemented as a multi-node caching layer, integrated within the GPFS, that can persistently and consistently cache *data and metadata* from a remote cluster. Every node in the Panache cache cluster has direct access to cached data and metadata. Thus, once data is cached, applications running on the Panache cluster achieve the same performance as if they were running directly on the remote cluster. If the data is not in the cache, Panache acts as a caching proxy to fetch the data in parallel both by using a parallel read across multiple cache cluster nodes to drive the ingest, and from multiple remote cluster nodes using pNFS. Panache allows updates to be made to the cache cluster at local cluster performance by asynchronously pushing all updates of data and metadata to the remote cluster.

More importantly, Panache, compared to other single-node file caching solutions, can function both as a standalone clustered file system and as a clustered caching proxy. Thus applications can run on the cache cluster using POSIX semantics and access, update, and traverse the directory tree even when the remote cluster is offline. As the cache mimics the same namespace as the remote cluster, browsing through the cache cluster (say with ls -R) shows the same listing of directories and files, as well as most of their remote attributes. Furthermore, NFS/pNFS clients can access the cache and see the same view of the data (as defined by NFS consistency semantics) as NFS clients accessing the data directly from the remote cluster. In essence, both in terms of consistency and performance, applications can operate as if the WAN did not exist.

Figure 3(b) shows the schematic of the Panache architecture with the cache cluster and the remote cluster. The remote cluster can be any file system or NAS filer exporting data over NFS/pNFS. Panache can operate on a multi-node cluster (henceforth called the cache cluster) where all nodes need not be identical in terms of hardware, OS, or support for remote network connectivity. Only a set of designated nodes, called *Gateway nodes*, need to have the hardware and software support for remote access. These nodes internally act as NFS/pNFS client proxies to fetch the data in parallel from the remote cluster. The remaining nodes of the cluster, called *Application nodes*, service the application data requests from the Panache cluster. The split between application and gateway nodes is conceptual and any node in the cache cluster can function both as a gateway node or a application node based on its configuration. The gate-

way nodes can be viewed as the edge of the cache cluster that can communicate with the remote cluster while the application nodes interface with the application. Figure 3(a) illustrates the internal components of a Panache node. Gateway nodes communicate with the pNFS kernel module via the VFS layer, which in turn communicates with the remote cluster. Gateway and application nodes communicate with each other via 26 different internal RPC requests from the user space daemon.

When an application request cannot be satisfied by the cache, due to a cache miss or to invalid cached data, the application node sends a read request to one of the gateway nodes. The gateway node then accesses the data from the remote cluster and returns it to the application node. Panache supports different mechanisms for gateway nodes to share the data with application nodes. One option is for the gateway nodes to write the remote data to the shared storage, which the application nodes can then read and return the data to the application. Another option is for gateway nodes to transfer the data directly to the application nodes using the cluster interconnect. Our current Panache prototype shares data through the storage subsystem, which can generally give higher performance than a typical network link.

All updates to the cache cause an application node to send and queue a command message on one or more gateway nodes. Note that this message includes no file data or metadata. At a later time, the gateway node(s) will read the data in parallel from the storage system and push it to the remote cluster over pNFS.

The selection of a gateway node to service a request needs to ensure that dependent requests are executed in the intended order. The application node selects a gateway node using a hash function based on a unique identifier of the object on which a file system operation is requested. Sections 5 and 6 describe how this identifier is chosen and how Panache executes read and update operations in more detail.

4.1 Consistency

Consistency in Panache can be controlled across various dimensions and can be defined relative to the cache cluster, the remote cluster and the network connectivity.

Definition 1 Locally consistent: *The cached data is considered locally consistent if a read from a node of the cache cluster returns the last write from any node of the cache cluster.*

Definition 2 Validity Lag: *The time delay between a read at the cache cluster reflecting the last write at the remote cluster.*

Definition 3 Synchronization Lag: *The time delay between a read at the remote cluster reflecting the last write at the cache cluster.*

Definition 4 Eventually Consistent: *After recovering from a node or network failure, in the absence of further failures, the cache and remote cluster data will eventually become consistent within the bounds of the lags.*

Panache, by virtue of relying on the cluster-wide distributed locking mechanism of the underlying clustered file system, is always locally consistent for the updates made at the cache cluster. Accesses are serialized by electing one of the nodes to be the token manager and issuing read and write tokens [26]. Local consistency within the cache cluster basically translates to the traditional definition of strong consistency [17].

For cross-cluster consistency across the WAN, Panache allows both the validity lag and the synchronization (synch) lag to be tunable based on the workload. For example, setting the validity lag to zero ensures that data is always validated with the remote cluster on an open and setting the synch lag to zero ensures that updates are flushed to the remote cluster immediately.

NFS uses a attribute timeout value (typically 30s) to recheck with the server if the file attributes have changed. Dependence on NFS consistency semantics can be removed via the O_DIRECT parameter (which disables NFS client data caching) and/or by disabling attribute caching (effectively setting the attribute timeout value to 0). NFSv4 file delegations can reduce the overhead of consistency management by having the remote cluster's NFS/pNFS server transfer ownership of a file to the cache cluster. This allows the cache cluster to avoid periodically checking the remote file's attributes and safely assume that the data is valid.

When the synch lag is greater than zero, all updates made to the cache are asynchronously committed at the remote cluster. In fact, the semantics will no longer be close-to-open as updates will ignore the file close and will be time delayed. Asynchronous updates can result in conflicts which, in Panache, are resolved using policies as discussed in Section 6.3.

When there is a network or remote cluster failure both the validation lag and synch lag become indeterminate. When connectivity is restored, the cache and remote clusters are eventually synchronized.

5 Synchronous Operations

Synchronous operations block until the remote operation completes, either because an object does not exist in the cache, i.e., a cache miss, or the object exists in the cache but needs to be revalidated. In either case, the object or its attributes need to be fetched or validated from the remote cluster on an application request. All file system data and metadata "read" operations, e.g., *lookup, open, read, readdir, getattr*, are synchronous. Unlike typical caching systems, Panache ingests the data and metadata

in parallel from multiple gateway nodes so that the cache miss or pre-populate time is limited only by the network bandwidth between the caching and remote clusters.

5.1 Metadata Reads

The first time an application node accesses an object via the VFS *lookup* or *open* operations, the object is created in the cache cluster as an empty object with no data. The mapping with the remote object is through the NFS filehandle that is stored with the inode as an extended attribute. The flow of messages proceeds as follows: i) the application node sends a request to the designated gateway node based on a hash of the inode number or its parent inode number if the object doesn't exist ii) the gateway node sends a request to the remote cluster's NFS/pNFS server(s), iii) on success at the remote cluster, the filehandle and attributes of the object are returned back to the gateway node which then creates the object in the cache, marks it as empty, and stores the remote filehandle mapping, iv) the gateway node then returns success back to the application node. On a later read or prefetch request the data in the empty object will be populated.

5.2 Parallel Data Reads

On an application *read* request, the application node first checks if the object exists in the local cache cluster. If the object exists but is empty or incomplete, the application node, as before, requests the designated gateway node to read in the requested offset and size. The gateway node, based on the prefetch policy, fetches the requested bytes or the entire file and writes it to the cache cluster. With prefetching, the whole file is asynchronously read after the byte-range requested by the application is ingested. Panache supports both whole file and partial file (segments consisting of a set of contiguous blocks) caching. Once the data is ingested, the application node reads the requested bytes from the local cache and returns them to the application as if they were present locally all along. Recall that the application and gateway nodes exchange only request and response messages while the actual data is accessed locally via the shared storage subsystem. On a later cache hit, the application node(s) can directly service the file read request from the local cache cluster. The cache miss performance is, therefore, limited by the network bandwidth to the remote cluster, while the cache hit performance is limited only by the local storage subsystem bandwidth (as shown in Table 1).

Panache scales I/O performance by using multiple gateway nodes to read chunks of a single file in parallel from the multiple nodes over NFS/pNFS. One of the gateway nodes (based on the hash function) becomes the coordinator for a file. It, in turn, divides the requests among the other gateway nodes which can proceed to read the data in parallel. Once a node is finished with its chunk it requests the coordinator for more chunks to read. When all the requested chunks have been read the gateway node responds to the application node that the requested blocks of the object are now in cache. If the remote cluster file system does not support pNFS but does support NFS access to multiple servers, data can still be read in parallel. Given N gateway nodes at the cache cluster and M nodes exporting data at the remote cluster, a file can be read either in 1xM (pNFS case) parallel streams, or min{N,M} 1x1 parallel streams (multiple gateway parallel reads with NFS) or NxM parallel streams (multiple gateway parallel reads with pNFS) as shown in Figure 4.

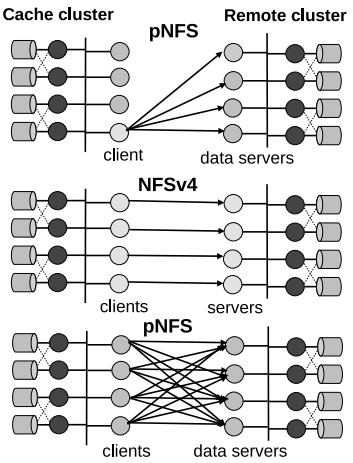


Figure 4: **Multiple gateway node configurations.** *The top setup is a single pNFS client reading a file from multiple data servers in parallel. The middle setup is multiple gateway nodes acting as NFS clients reading parts of the file from the remote cluster's NFS servers. The bottom setup has multiple gateway nodes acting as pNFS clients reading parts of the file in parallel from multiple data servers.*

File Read	2 gateway nodes	3 gateway nodes
Miss	1.456 Gb/s	1.952 Gb/s
Hit	8.24 Gb/s	8.24 Gb/s
Direct over pNFS	1.776 Gb/s	2.552 Gb/s

Table 1: **Panache (with pNFS) and pNFS read performance using the IOR benchmark.** *Clients read 20 files of 5GB each using 2 and 3 gateway nodes with gigabit ethernet connecting to a 6-node remote cluster. Panache scales on both cache miss and cache hit. On cache miss, Panache incurs the overhead of passing data through the SAN, while on a cache hit it saturates the SAN.*

5.3 Namespace Caching

Panache provides a standard POSIX file system interface for applications. When an application tra-

verses the namespace directory tree, Panache reflects the view of the corresponding tree at the remote cluster. For example, an "ls -R" done at the cache cluster presents the same list of entries as one done at the remote cluster. Note that Panache does not simply return the directory listing with $dirents$ containing the $< name, inode\_num >$ pairs from the remote cluster (as an NFS client would). Instead, Panache first creates the directory entries in the local cluster and then returns the cached name and inode number to the application. This is done to ensure application nodes can continue to traverse the directory tree if a network or server outage occurs. In addition, if the cache simply returns the remote inode numbers to the application, and later a file is created in the cache with that inode number, the application may observe different inode numbers for the same file.

One approach to returning consistent inode numbers to the application on a *readdir* (directory listing) or *lookup* and *getattr*, e.g., file stat, is by mandating that the remote cluster and the cache cluster mirror the same inode space. This can be impossible to implement where remote inode numbers conflict with inode numbers of reserved files and clearly limits the choice of the remote cluster file systems. A simple approach is to fetch the attributes of all the directory entries, i.e., an extra lookup across the network and create the files locally on a *readdir* request. This approach of creating files on a directory access has an obvious performance penalty for directories with a large number of files.

To solve the performance problems with creates on a *readdir* and allow for the cache cluster to operate with a separate inode space, we create only the directory entries in the local cluster and create placeholders for the actual files and directories. This is done by allocating but not creating or using inodes for the new entries. This allows us to satisfy the readdir request with locally allocated inode numbers without incurring the overhead of creating all the entries. These allocated, but not yet created, entries are termed *orphans*. On a subsequent *lookup*, the allocated inode is "filled" with the correct attributes and created on disk. Orphan inodes cause interesting problems on *fsck*, file deletes, and cache eviction and have to be handled separately in each case. Table 2 shows the performance (in secs) of reading a directory for 3 cases: i) where the files are created on a *readdir*, ii) when only orphan inodes are created, and iii) when the readdir is returned locally from the cache.

Files per dir	readdir & creates	readdir & orphan inodes	readdir from cache
100	1.952 (s)	0.77 (s)	0.032 (s)
1,000	3.122	1.26	0.097
10,000	7.588	2.825	0.15
100,000	451.76	25.45	1.212

Table 2: **Cache traversal with a readdir.** *Performance (in secs.) of a readdir on a cache miss where the individual files are created vs. the orphan inodes. The last column shows the performance of readdir on a cache hit.*

5.4 Data and Attribute Revalidation

The data validity in the cache cluster is controlled by a revalidation timeout, in a manner similar to the NFS attribute timeout, whose value is determined by the desired *validity lag* of the workload. The cache cluster's inode stores both the local modification time $mtime_{local}$ and inode change time $ctime_{local}$ along with the remote $mtime_{remote}, ctime_{remote}$. When the object is accessed after the revalidation timeout has expired the gateway node gets the remote object's time attributes and compares them with the stored values. A change in $mtime_{remote}$ indicates that the object's data was modified and a change in $ctime_{remote}$, indicates that the object's inode was changed as the attributes or data was modified [1]. In case the remote cluster supports NFSv4 with delegations, some of this overhead can be removed by assuming the data is valid when there is an active delegation. However, every time the delegation is recalled, the cache falls back to timeout based revalidation.

During a network outage or remote server failure, the revalidation lag becomes indeterminate. By policy, either the requests are made blocking where they wait till connectivity is restored or all synchronous operations are handled locally by the cache cluster and no request is sent to the gateway node for remote execution.

6 Asynchronous Operations

One important design decision in Panache was to mask the WAN latencies by ensuring applications see the cache cluster's performance on all data writes and metadata updates. Towards that end, all data writes and metadata updates are done asynchronously—the application proceeds after the update is "committed" to the cache cluster with the update being pushed to the remote cluster at a later time governed by the synch lag. Moreover, executing updates to the remote cluster is done in parallel across *multiple* gateway nodes. Most caching systems delay only data writes and perform all the metadata and namespace updates synchronously, preventing disconnected operation. By allowing asynchronous metadata updates, Panache allows data and metadata updates at local speeds and also masks remote cluster failures and network outages.

In Panache asynchronous operations consist of operations that encapsulate modifications to the cached file

[1] Currently we ignore the possibility that the mtime may not change on update. This may require content based signatures or a kernel supported change_info to verify.

system. These include relatively simple modify requests that involve a single file or directory, e.g., write, truncate, and modification of attributes such as ownership, times, and more complex requests that involve changes to the name space through updates of one or more directories, e.g., creation, deletion or renaming of a file and directory or symbolic links.

6.1 Dependent Metadata Operations

In contrast to synchronous operations, asynchronous operations modify the data and metadata at the cache cluster and then are simply queued at the gateway nodes for delayed execution at the remote cluster. Each gateway node maintains an in-memory queue of asynchronous requests that were sent by the application nodes. Each message contains the unique object identifier *fileId:* $<$ *inode_num, gen_num, fsid* $>$ of one or more objects being operated upon and the parameters of the command.

If there is a single gateway node and all the requests are queued in FIFO order, then operations will execute remotely in the same order as they did in the cache cluster. When multiple gateway nodes can push commands to the remote cluster, the distributed multi-node queue has to be controlled to maintain the desired ordering. To better understand this, let's first define some terms.

Definition 5 *A pair of update commands $C_i(X), C_j(X)$, on an object X, executed at the cache cluster at time $t_i < t_j$ are said to be* **time ordered** *, denoted by $C_i \rightarrow C_j$, if they need to be executed in the same relative order at the remote cluster.*

For example, commands CREATE(File_X) and WRITE(File_X, offset, length) are time ordered as the data writes cannot be pushed to the remote cluster until the file gets created.

Observation 1 *If commands C_i, C_j, C_k are pair-wise time ordered, i.e., $C_i \rightarrow C_j$ and $C_j \rightarrow C_k$ then the three commands form a time ordered sequence $C_i \rightarrow C_j \rightarrow C_k$*

Definition 6 *A pair of objects O_x, O_y, are said to be* **dependent objects** *if there exists queued commands C_i and C_j such that $C_i(O_x)$ and $C_j(O_y)$ are time ordered.*

For example, creating a file $File_X$ and its parent directory Dir_Y make X and Y dependent objects as the parent directory create has to be pushed before the file create.

Observation 2 *If objects O_x, O_y, and O_y, O_z are pair-wise dependent, then O_x, O_z are also dependent objects.*

Observe that the creation of a file depends on the creation of its parent directory, which in turn depends on the creation of its parent directory, and so on. Thus, a create of a directory tree creates a chain of dependent objects. The removes follow the reverse order where the rmdir depends on the directory being empty so that the removes of the children need to execute earlier.

Definition 7 *A set of commands over a set of objects, $C_1(O_x), C_2(O_y)...C_n(O_z)$, are said to be* **permutable** *if they are neither time ordered nor contain dependent objects.*

Thus permutable commands can be pushed out in parallel from multiple gateway nodes without affecting correctness. For example, *create file A, create file B* are permutable among themselves.

Based on these definitions, if all commands on a given object are queued and pushed in FIFO order at the same gateway node we trivially get the time order requirements satisfied for all commands on that object. Thus, Panache hashes on the object's unique identifier, e.g., inode number and generation number, to select a gateway node on which to queue an object. It is dependent objects queued on different gateway nodes that make distributed queue ordering a challenge. To further complicate the issue, some commands such as rename and link involve multiple objects.

To maintain the distributed time ordering among dependent objects across multiple gateway node queues, we build upon the GPFS distributed token management infrastructure. This infrastructure currently coordinates access to shared objects such as inodes and byte-range locks and is explained in detail elsewhere [26]. Panache extends this distributed token infrastructure to coordinate execution of queued commands among multiple gateway nodes. The key idea is that an enqueued command acquires a shared token on objects on which it operates. Prior to the execution of a command to the remote cluster, it upgrades these tokens to exclusive, which in turn forces a token revoke on the shared tokens that are currently held by other commands on dependent objects on other gateway nodes. When a command receives a token revoke, it then also upgrades its tokens to exclusive, which results in a chain reaction of token revokes. Once a command acquires an exclusive token on its objects, it is executed and dequeued. This process results in all commands being pushed out of the distributed queues in dependent order.

The link and rename commands operate on multiple objects. Panache uses the hash function to queue these commands on multiple gateway nodes. When a multi-object request is executed, only one of the queued commands will execute to the remote cluster, with the others simply acting as placeholders to ensure intra-gateway node ordering.

6.2 Data Write Operations

On a write request, the application node first writes the data locally to the cache cluster and then sends a message to the designated gateway node to perform the write operation at the remote cluster. At a later time, the gateway node reads the data from the cache cluster and completes the remote write over pNFS.

The delayed nature of the queued write requests allow some optimizations that would not otherwise be possible if the requests had been synchronously serviced. One such optimization is write coalescing that groups the write request to match the optimal GPFS and NFS buffer sizes. The queue is also evaluated before requests are serviced to eliminate transient data updates, e.g., the creation and deletion of temporary files. All such "canceling" operations are purged without affecting the behavior of the remote cluster.

In case of remote cluster failures and network outages, all asynchronous operations can still update the cache cluster and return successfully to the application. The requests simply remain queued at the gateway nodes pending execution at the remote cluster. Any such failure, however, will affect the synchronization lag making the consistency semantics fall back to a looser eventual consistency guarantee.

6.3 Discussion

Conflict Handling Clearly, asynchronous updates can result in non-serializable executions and conflicting updates. For example, the same file may be created or updated by both the cache cluster and the remote cluster. Panache cannot prevent such conflicts, but it will detect them and resolve them based on simple policies. For example, one policy could have the cache cluster always override any conflict; another policy could move a copy of the conflicting file to a special ".conflicts" directory for manual inspection and intervention, similar to the lost+found directory generated on a normal file system check (fsck) scan. Further, it is possible to merge some types of conflicts without intervention. For example, a directory with two new files, one created by the cache and another by the remote system can be merged to form the directory containing both files. Earlier research on conflict handling of disconnected operations in Coda [25] and Intermezzo have inspired some of the techniques used in Panache after being suitably modified to handle a cluster setting.

Access control and authentication: One aspect of the caching system is that data is no more vulnerable to wrongful access as it was at the remote cluster. Panache requires userid mappings to make sure that file access permissions and ACLs setup at the remote cluster are enforced at the cache. Similarly, authentication via NFSv4's RPCSEC_GSS mechanism can be forwarded to the remote cluster to make sure end-to-end authentication can be enforced.

Recovery on Failure: The queue of pending updates can be lost due to memory pressures or a cache cluster node reboot. To avoid losing track of application updates, Panache stores sufficient persistent state to recreate the updates and synchronize the data with the remote cluster. The persistent state is stored in the inode on disk and relies on the GPFS fast inode scan to determine which inodes have been updated. Inode scans are very efficient as they can be done in parallel across multiple nodes and are basically a sequential read of the inode file. For example, in our test environment, a simple inode scan (with file attributes) on a single application node of 300K files took 2.24 seconds.

7 Evaluation

In this section we assess the performance of Panache as a scalable cache. We first use the *IOR* micro-benchmark [2] to analyze the amount of overhead Panache incurs along the data path to the remote cluster. We then use the *mdtest* micro-benchmark [4] to measure the overhead Panache incurs to queue and flush metadata operations on the gateway nodes. Finally, we run a parallel visualization application and a Hadoop application to analyze Panache with an HPC access pattern.

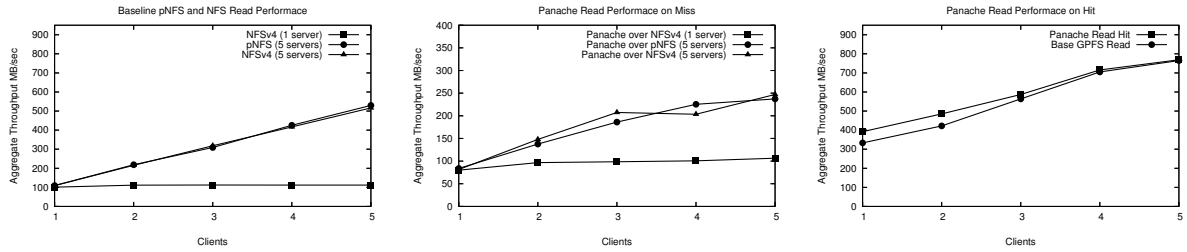
7.1 Experimental Setup

All experiments use a sixteen-node cluster connected via gigabit Ethernet, with each node assigned a different role depending on the experiment. Each node is equipped with dual 3 GHz Xeon processors, 4 GB memory and runs an experimental version of Linux 2.6.27 with pNFS. GPFS uses a 1 MB stripe size. All NFS experiments use 32 server threads and 512 KB wsize and rsize. All nodes have access to the SAN, which is comprised of a 16-port FC switch connected to a DS4800 storage controller with 12 LUNs configured for the cache cluster.

7.2 I/O Performance

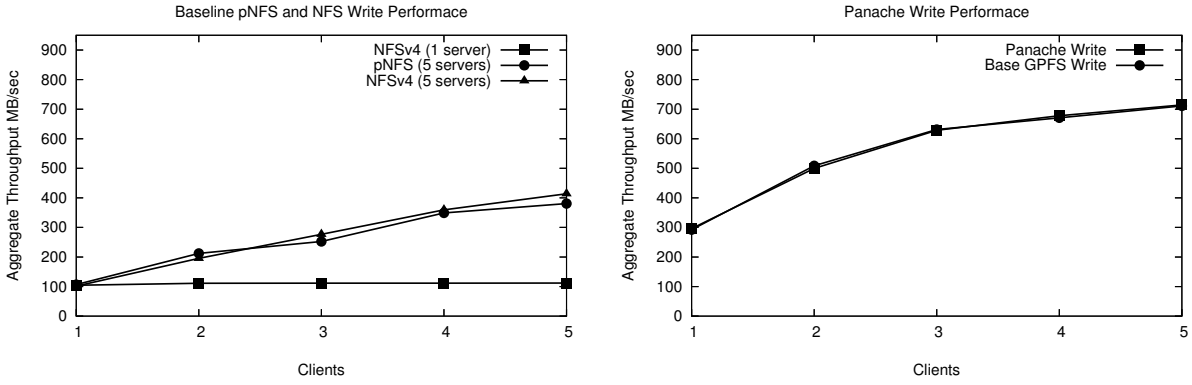
Ideally, the design of Panache is such that it should match the storage subsystem throughput on a cache hit and saturate the network bandwidth on a cache miss (assuming that the network bandwidth is less than the disk bandwidth of the cache cluster).

In the first experiment, we measure the performance reading separate 8 GB files in parallel from the remote cluster. Our local Panache cluster uses up to 5 application and gateway nodes, while the remote 5 node GPFS cluster has all nodes configured to be pNFS data servers. As we increase the number of application (client) nodes,



(a) pNFS and NFSv4 (b) Panache Cache Miss (c) Panache Cache Hit vs. Standard GPFS

Figure 5: **Aggregate Read Throughput.** *(a) pNFS and NFSv4 scale with available remote bandwidth. (b) Panache using pNFS and NFSv4 scales with available local bandwidth. (c) Panache local read performance matches standard GPFS.*



(a) pNFS and NFSv4 (b) Panache vs. Standard GPFS

Figure 6: **Aggregate Write Throughput.** *(a) pNFS and NFSv4 scale with available disk bandwidth. (b) Panache local write performance matches standard GPFS, demonstrating the negligible overhead of queuing write messages on the gateway nodes.*

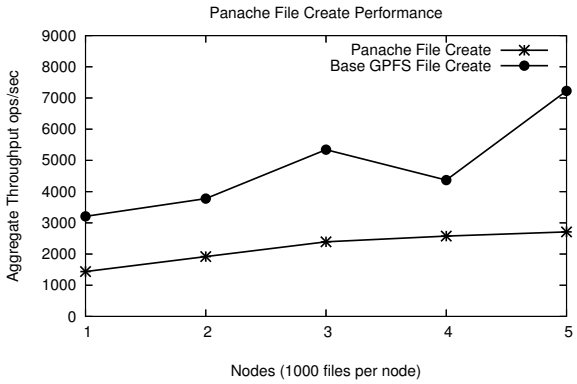
the number of gateway nodes increases as well since the miss requests are evenly dispatched. Figure 5(a) displays how the underlying data transfer mechanisms used by Panache can scale with the available bandwidth. NFSv4 with a single server is limited to the bandwidth of the single remote server while NFSv4 with multiple servers and pNFS can take advantage of all 5 available remote servers. With each NFSv4 client mounting a separate server, aggregate read throughput reaches a maximum of 516.49 MB/s with 5 clients. pNFS scales in a similar manner, reaching a maximum aggregate read throughput of 529.37 with 5 clients.

Figure 5(b) displays the aggregate read throughput of Panache utilizing pNFS and NFSv4 as its underlying transfer mechanism. The performance of Panache using NFSv4 with a single server is 5-10% less than standard NFSv4 performance. This performance hit comes from our Panache prototype, which does not fully pipeline the data between the application and gateway nodes. When Panache uses pNFS and NFSv4 using multiple servers, increasing the number of clients gives a maximum aggregate throughput of 247.16 MB/s due to a saturation of the storage network. A more robust SAN would shift the bottleneck back on the network between the local and remote clusters.

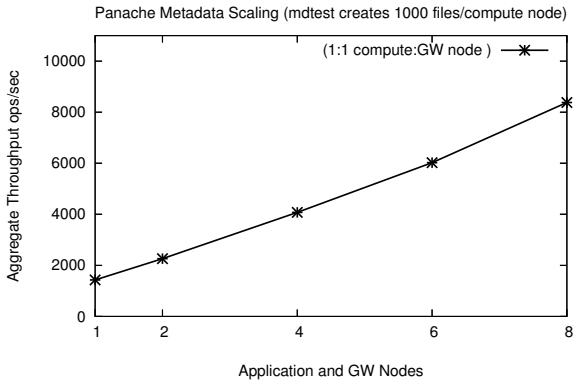
Finally, Figure 5(c) demonstrates that once a file is cached, Panache stays out of the I/O path, allowing the aggregate read throughput of Panache to match the aggregate read throughput of standard GPFS.

In the second experiment we increase the number of clients writing to a separate 8 GB files. As shown in Figure 6(b), the aggregate write throughput of Panache matches the aggregate write throughput of standard GPFS. For Panache, writes are done locally to GPFS while a write request is queued on a gateway node for asynchronous execution to the remote cluster. This experiment demonstrates that the extra step of queuing the write request on the gateway node does not impact write performance. Therefore, application write throughput is not constrained by the network bandwidth or the number of pNFS data servers, but rather by the same constraints as standard GPFS.

Eventually, data written to the cache must be synchronized to the remote cluster. Depending on the capabilities of the remote cluster, Panache can use three I/O methods: standard NFSv4 to a single server, standard NFSv4 with each client mounting a separate remote server, and pNFS. Figure 6(a) displays the ag-



(a) File metadata ops.



(b) Gateway metadata ops.

Figure 7: **Metadata performance.** *Performance of mdtest benchmark for file creates. Each node creates 1000 files in parallel. In (a), we use a single gateway node. In (b), the number of application and gateway nodes are increased in unison, with each cluster node playing both application and gateway roles.*

gregate write performance of writing separate 8 GB files to the remote cluster using these three I/O methods. Unsurprisingly, aggregate write throughput for standard NFSv4 with a single server remains flat. With each NFSv4 client mounting a separate server, aggregate write throughput reaches a maximum of 413.77 MB/s with 5 clients. pNFS scales in a similar manner, reaching a maximum aggregate write throughput of 380.78 MB/s with 5 clients. Neither NFSv4 with multiple servers nor pNFS saturate the available network bandwidth due to limitations in the disk subsystem.

It is important to note that although the performance of pNFS and NFSv4 with multiple servers appears on the surface to be similar, the lack of coordinated access in NFSv4 creates several performance hurdles. For instance, if there are a greater number of gateway nodes than remote servers, NFSv4 clients will not be evenly load balanced among the servers, creating possible hot spots. pNFS avoids this by always balancing I/O requests among the remote servers evenly. In addition, NFSv4 unaligned file writes across multiple servers can create false sharing of data blocks, causing the cluster file system to lock and flush data unnecessarily.

7.3 Metadata Performance

To measure the metadata update performance in the cache cluster we use the *mdtest* benchmark, which performs file creates from multiple nodes in the cluster. Figure 7(a) shows the aggregate throughput of 1000 file create operations per cluster node. With 4 application nodes simultaneously creating a total of 4000 files, the Panache throughput (2574 ops/s) is roughly half that of the local GPFS (4370 ops/s) performance. The Panache code path has the added overhead of first creating the file locally and then sending a RPC to queue the operation on a gateway node. As the graph shows, as the number of nodes increases, we can saturate the single gateway node. To see the impact of increasing the number of gateway nodes, Figure 7(b) demonstrates the scale up when the number of application nodes and gateway nodes increase in tandem, up to a maximum of 8 cache and remote nodes.

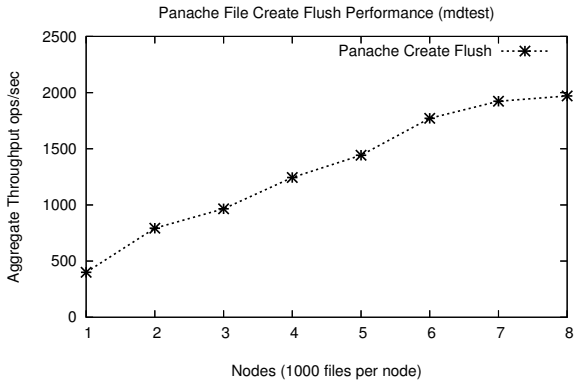


Figure 8: **Metadata flush performance.** *Performance of mdtest benchmark for file creates with flush. Each node flushes 1000 files in parallel back to the home cluster.*

As all updates are asynchronous, we also demonstrate the performance of flushing file creates to the remote cluster in Figure 8. By increasing the number of gateway and remote nodes in tandem, we can scale the number of creates per second from 400 to 2000, a five fold increase for 7 additional nodes. The lack of linear increase is due to our prototype's inefficient use of the GPFS token management service.

7.4 WAN Performance

To validate the effectiveness of Panache over a WAN we used the IOR parallel file read benchmark and the Linux tc command. The WAN represented the 30ms la-

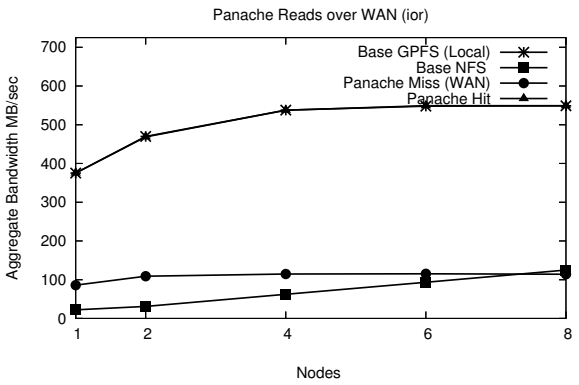


Figure 9: **IOR file Reads over a WAN.** *The 8 node cache cluster and 8 node remote cluster are separated by a 30ms latency link. Each file is 5GB in size.*

tency link between the IBM San Jose and Tucson facilities. The cache and remote clusters both contain 8 nodes, keeping the gateway and remote nodes in tandem. Figure 9 shows the aggregate bandwidth on both a hit and a miss for an increasing number of nodes in the cluster. The hit bandwidth matches that of a local GPFS read. For cache miss, while Panache can utilize parallel ingest to increase performance initially, both Panache and NFS eventually suffer from slow network bandwidth.

7.5 Visualization for Cognitive Models

This section evaluates Panache with a real supercomputing application that visualizes the $8x10^6$ neural firings of a large scale cognitive model of a mouse brain [23]. The cognitive model runs at a remote cluster (a BlueGene/L system with 4096 nodes) and the visualization application runs at the cache cluster and creates a "movie" as output. In the experiment in Table 3, we copied a fraction of the data (64 files of 200MB each) generated by the cognitive model to our 5 node remote cluster and ran the visualization application on the Panache cluster. The application reads in the data and creates a movie file of 250MB. Visualization is a CPU-bound operation, but asynchronous writes helped Panache reduce runtime over pNFS by 14 percent. Once the data is cached, time to regenerate the visualization files is reduced by an additional 17.6 percent.

pNFS	Panache (miss)	Panache (hit)
46.74 (s)	40.2 (s)	31.96 (s)

Table 3: **Supercomputing application.** *pNFS includes remote cluster reads and writes. Panache Miss reads from the remote and asynchronous write back. Panache Hit reads from the cache and asynchronous write back.*

7.6 MapReduce Application

The MapReduce framework provides a programmable infrastructure to build highly parallel applications that operate on large data sets [11]. Using this framework, applications define a map function that defines a key and operates on a chunk of the data. The reduce function aggregates the results for a given key. Developers may write several MapReduce programs to extract different properties from a single data set, building a use case for remote caching. We use the MapReduce framework from Hadoop 0.20.1 [6] and configured it to use Panache as the underlying distributed store (instead of the HDFS file system it uses by default).

Table 4 presents the performance of Distributed Grep, a canonical MapReduce example application, over a data set of 16 files, 500MB each, running in parallel across 8 nodes with the remote cluster also consisting of 8 nodes. The GPFS result was the baseline result where the data was already available in the local GPFS cluster. In the Panache miss case, as the distributed grep application accessed the input files, the gateway nodes dynamically ingested the data in parallel from the remote cluster. In the hit case, Panache revalidated the data every 15 secs with the remote cluster. This experiment validates our assertion that data can be dynamically cached and immediately available for parallel access from multiple nodes within the cluster.

Hadoop+GPFS	Hadoop+Panache		
Local	Miss LAN	Miss WAN	Hit
81.6 (s)	113.1 (s)	140.6 (s)	86.5 (s)

Table 4: **MapReduce application.** *Distributed Grep using the Hadoop framework over GPFS and Panache. The WAN results are over a 30ms latency link.*

8 Related Work

Distributed file systems have been an active area of research for almost two decades. NFS is among the most widely-used distributed networked file systems. Other variants of NFS, Spritely NFS [28] and NQNFS [20] added stronger consistency semantics to NFS by adding server callbacks and leases. NFSv4 greatly enhances wide-area access support, optimizes consistency support via delegations, and improves compatibility with Windows. The latest revision, NFSv4.1, also adds parallel data access across a variety of clustered file and storage systems. In the non-Unix world, the Common Internet File System (CIFS) protocol is used to allow MS-Windows hosts to share data over the Internet. While these distributed file systems provide remote file access and some limited in-memory client caching they cannot operate across multiple nodes and in the presence of net-

work and server failures.

Apart from NFS, another widely studied globally distributed file system is AFS [17]. It provides close-to-open consistency, supports client-side persistent caching, and relies on client callbacks as the primary mechanism for cache revalidation. Later, Coda [25] and Ficus [24] dealt with replication for better scalability while focusing on disconnected operations for greater data availability in the event of a network partition.

More recently, the work on TierStore applies some of the same principles for the development and deployment of applications in bandwidth challenged networks [13]. It defines Delay Tolerant Networking with a store-and-forward network overlay and a publish/subscribe-based multicast replication protocol. In limited bandwidth environments, LBFS takes a different approach by focusing on reducing bandwidth usage by eliminating cross-file similarities [22]. Panache can easily absorb some of its similarity techniques to reduce the data transfer to and from the cache.

A plethora of commercial WAFS and WAN acceleration products provide caching for NFS and CIFS using custom devices and proprietary protocols [1]. Panache differs from WAFS solutions as it relies on standard protocols between the remote and cache sites. Muntz and Honeyman [21] looked at multi-level caching to solve scaling problems in distributed file systems but questioned its effectiveness. However, their observations may not hold today as the advances in network bandwidth, web-based applications, and the emerging trends of cloud stores have substantially increased remote collaboration. Furthermore, cooperative caching, both in the web and file system space, has been extensively studied [10]. The primary focus, however, has been to expand the cache space available by sharing data across sites to improve hit rates.

Lustre [3] and PanFS [29] are highly-scalable object based cluster file systems. These efforts have focused on improving file-serving performance and are not designed for remotely accessing data from existing file servers and NAS appliances over a WAN.

FS-Cache is a single-node caching file system layer for Linux that can be used to enhance the performance of a distributed file system such as NFS [18]. FS-Cache is not a standalone file system; instead it is meant to work with the front and back file systems. Unlike Panache, it does not mimic the namespace of the remote file system and does not provide direct POSIX access to the cache. Moreover, FS-Cache is a single node system and is not designed for multiple nodes of a cluster accessing the cache concurrently. Similar implementations such as CacheFS are available on other platforms such as Solaris and as a stackable file system with improved cache policies [27].

A number of research efforts have focused on building large scale distributed storage facilities using customized protocols and replication. The Bayou [12] project introduced eventual consistency across replicas, an idea that we borrowed in Panache for converging to a consistent state after failure. The Oceanstore project [19] used Byzantine agreement techniques to coordinate access between the primary replica and the secondaries. The PRACTI replication framework [9] separated the flow of cache invalidation traffic from that of data itself. Others like Farsite [8] enabled unreliable servers to combine their resources into a highly-available and reliable file storage facility.

Recently the success of file sharing on the Web, especially BitTorrent [5] which has been widely studied, has triggered renewed effort for applying similar ideas to build peer-to-peer storage systems. BitTorrent's chunk-based data retrieval method that enables clients to fetch data in parallel from multiple remote sources is similar to the implementation of parallel reads in Panache.

9 Conclusions

This paper introduced Panache, a scalable, high-performance, clustered file system cache that promises seamless access to massive and remote datasets. Panache supports a POSIX interface and employs a fully parallelizable design, enabling applications to saturate available network and compute hardware. Panache can also mask fluctuating WAN latencies and outages by acting as a standalone file system under adverse conditions.

We evaluated Panache using several data and metadata micro-benchmarks in local and wide area networks, demonstrating the scalability of using multiple gateway nodes to flush and ingest data from a remote cluster. We also demonstrated the benefits for both a visualization and analytics application. As Panache achieves the performance of a clustered file system on a cache hit, large scale applications can leverage a clustered caching solution without paying the performance penalty of accessing remote data using out-of-band techniques.

References

[1] Blue Coat Systems, Inc. `www.bluecoat.com`.

[2] IOR Benchmark. `sourceforge.net/projects/ior-sio`.

[3] Lustre file system. `www.lustre.org`.

[4] Mdtest benchmark. `sourceforge.net/projects/mdtest`.

[5] Bittorrent. `www.bittorrent.com`.

[6] Hadoop Distribued Filesystem. `hadoop.apache.org`.

[7] Nirvanix Storage Delivery Network. `www.nirvanix.com`.

[8] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch,

M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation*, 2002.

[9] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc. of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2006.

[10] M. Dahlin, R. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. of the 1st Symposium on Operating Systems Design and Implementation*, 1994.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. of the 6th Symposium on Operating System Design and Implementation*, 2004.

[12] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proc. of the IEEE Workshop on Mobile Computing Systems & Applications*, 1994.

[13] M. Demmer, B. Du, and E. Brewer. Tierstore: a distributed filesystem for challenged networks in developing regions. In *Proc. of the 6th USENIX Conference on File and Storage Technologies*, 2008.

[14] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *Proc. of the 19th ACM symposium on operating systems principles*, 2003.

[15] A. Gulati, M. Naik, and R. Tewari. Nache: Design and Implementation of a Caching Proxy for NFSv4. In *Proc. of the Fifth Conference on File and Storage Technologies*, 2007.

[16] D. Hildebrand and P. Honeyman. Exporting storage systems in a scalable manner with pNFS. In *Proc. of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2005.

[17] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[18] D. Howells. FS-Cache: A Network Filesystem Caching Facility. In *Proc. of the Linux Symposium*, 2006.

[19] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[20] R. Macklem. Not Quite NFS, soft cache consistency for NFS. In *Proc. of the USENIX Winter Technical Conference*, 1994.

[21] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems. In *Proc. of the USENIX Winter Technical Conference*, 1992.

[22] A. Muthitacharoen, B. Chen, and D. Mazi. A low-bandwidth network file system. In *Proc. of the 18th ACM symposium on operating systems principles*, 2001.

[23] A. Rajagopal and D. Modha. Anatomy of a cortical simulator. In *Proc. of Supercomputing '07*, 2007.

[24] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving file conflicts in the Ficus file system. In *Proc. of the USENIX Summer Technical Conference*, 1994.

[25] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[26] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proc. of the First Conference on File and Storage Technologies*, 2002.

[27] G. Sivathanu and E. Zadok. A Versatile Persistent Caching Framework for File Systems. *Technical Report FSL-05-05, Stony Brook University*, 2005.

[28] V. Srinivasan and J. Mogul. Spritely NFS: experiments with cache-consistency protocols. In *Proc. of the 12th Symposium on Operating Systems Principles*, 1989.

[29] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *Proc. of the 6th Conference on File and Storage Technologies*, 2008.