# Providing Tunable Consistency for a Parallel File Store*

Murali Vilayannur [1] Partho Nath Anand Sivasubramaniam
*Department of Computer Science and Engineering*
*Pennsylvania State University*
{vilayann,nath,anand}@cse.psu.edu

## Abstract

Consistency, throughput, and scalability form the backbone of a cluster-based parallel file system. With little or no information about the workloads to be supported, a file system designer has to often make a one-glove-fits-all decision regarding the consistency policies. Taking a hard stance on consistency demotes throughput and scalability to second-class status, having to make do with whatever leeway is available. Leaving the choice and granularity of consistency policies to the user at open/mount time provides an attractive way of providing the best of all worlds. We present the design and implementation of such a file-store, CAPFS (Content Addressable Parallel File System), that allows the user to define consistency semantic policies at runtime. A client-side plug-in architecture based on user-written plug-ins leaves the choice of consistency policies to the end-user. The parallelism exploited by use of multiple data stores provides for bandwidth and scalability. We provide extensive evaluations of our prototype file system on a concurrent read/write workload and a parallel tiled visualization code.

## 1 Introduction

High-bandwidth I/O continues to play a critical role in the performance of numerous scientific applications that manipulate large data sets. Parallelism in disks and servers provides cost-effective solutions at the hardware level for enhancing I/O bandwidth. However, several components in the system software stack, particularly in the file system layer, fail to meet the demands of applications. This is primarily due to tradeoffs that parallel file system designers need to make between performance and scalability goals at one end, and transparency and ease-of-use goals at the other.

Compared to network file systems (such as NFS [26], AFS [14], and Coda [16]), which despite allowing multiple file servers still allocate all portions of a file to a server, parallel file systems (such as PVFS [7], GPFS [27], and Lustre [6]) distribute portions of a file across different servers. With the files typically being quite large and different processes of the same application sharing a file, such striping can amplify the overall bandwidth. With multiple clients reading and writing a file, coordination between the activities becomes essential to enforce a consistent view of the file system state.

The level of sharing when viewed at a file granularity in parallel computing environments is much higher than that observed in network file systems [4, 21], making consistency more important. Enforcement of such consistency can, however, conflict with performance and scalability goals. Contemporary parallel file system design lacks a consensus on which path to take. For instance, PVFS provides high-bandwidth access to I/O servers without enforcing overlapping-write atomicity, leaving it entirely to the applications or runtime libraries (such as MPI-I/O [9]) to handle such consistency requirements. On the other hand, GPFS and Lustre enforce byte-range POSIX [28] consistency. Locking is used to enforce serialization, which in turn may reduce performance and scalability (more scalable strategies are used in GPFS for fine-grained sharing, but the architecture is fundamentally based on distributed locking).

Serialization is not an evil but a necessity for certain applications. Instead of avoiding consistency issues and using an external mechanism (e.g., DLM [15]) to deal with serialization when required, incorporating consistency enforcement in the design might reduce the overheads. Hence the skill lies in being able to make an informed decision regarding the consistency needs of an application. A key insight here is that applications, not the system, know best to deal with their concurrency needs. In fact, partial attempts at such optimizations already exist — many parallel applications partition the data space to minimize read-write and write-write sharing. Since different applications can have different sharing behavior, designing for performance *and* consistency would force the design to cater to *all* their needs — si-

*First published in Fourth USENIX Conference on File and Storage Technologies, 2005

multaneously! Provisioning a single (and strict) consistency mechanism may not only make such fine-grained customization hard but may also constrain the suitability of running diverse sets of applications on the same parallel file system.

Addressing some of these deficiencies, this paper presents the design and implementation of a novel parallel file system called CAPFS that provides the following notable features:

- To the best of our knowledge, CAPFS is the first file system to provide a tunable consistency framework that can be customized for an application. A set of plug-in libraries is provided with clearly defined entry points, to implement different consistency models, including POSIX, Session, and Immutable-files. Though a user could build a model for each application, we envision a set of predefined libraries that an application can pick before execution for each file and/or file system.
- The data store in CAPFS is content-addressable. Consequently, blocks are not modified in place, allowing more concurrency in certain situations. In addition, content addressability can make *write propagation* (which is needed to enforce coherence) more efficient. For instance, update-based coherence mechanisms are usually avoided because of the large volume of data that needs to be sent. In our system however, we allow update messages that are just a sequence of (cryptographic) hashes of the new content being generated. Further, content addressability can exploit commonality of content within and across files, thereby lowering caching and network bandwidth requirements.
- Rather than locking when enforcing serialization for read-write sharing or write-write sharing (write atomicity), CAPFS uses optimistic concurrency control mechanisms [17, 19] with the presumption that these are rare events. Avoidance of distributed locking enhances the scalability and fault-tolerance of the system.

The rest of this paper is organized as follows. The next section outlines the design issues guiding our system architecture, following which the system architecture and the operational details of our system are presented in Section 3. An experimental evaluation of our system is presented in Section 4 on a concurrent read/write workload and on a parallel tiled visualization code. Section 5 summarizes related work and Section 6 concludes with the contributions of this paper and discusses directions for further improvements.

# 2 Design Issues

The guiding rails of the CAPFS design is based on the following goals: 1) user should be able to define the consistency policy at a chosen granularity, and 2) implementation of consistency policies should be as lightweight and concurrent as possible. The CAPFS design explores these directions simultaneously — providing easily expressible, tunable, robust, lightweight and scalable consistency without losing focus of the primary goal of providing high bandwidth.

## 2.1 Tunable Consistency

If performance is a criterion, consistency requirements for applications might be best decided by applications themselves. Forcing an application that has little or no sharing to use a strong or strict consistency model may lead to unnecessarily reduced I/O performance. Traditional techniques to provide strong file system consistency guarantees for both meta-data and data use variants of locking techniques. In this paper, we are interested in providing tunable semantic guarantees for *file data alone*.

The choice of a system wide consistency policy may not be easy. NFS [26] offers poorly defined consistency guarantees that are not suitable for parallel workloads. On the other hand, Sprite [20] requires the central server to keep track of all concurrent sessions and disable caching at clients when write-sharing is detected. Such an approach forces *all* write-traffic to be network bound from thereon until one or more processes close the shared file. Although such a policy enforces correctness, it penalizes performance of applications when writers update spatially disjoint portions of the same file which is quite common in parallel workloads. For example, an application may choose to have a few temporary files (store locally, no consistency), a few files that it knows no one else will be using (no consistency), a few files that will be extensively shared (strong consistency), and a few files that might have sharing in the rare case (weaker user-defined consistency). A single consistency policy for a cluster-based file system cannot cater to the performance of different workloads such as those described above.

As shown in Figure 1, CAPFS provides a client-side plug-in architecture to enable users to define their own consistency policies. The users write plug-ins that define what actions should be taken before and after the client-side daemon services the corresponding system call. (The details of the above mechanism are deferred to Section 3.6).

The choice of a plug-in architecture to implement this functionality has several benefits. Using this architecture, a user can define not just standard consistency policies like POSIX, session and NFS, but also custom policies, at a chosen granularity (sub-file, file, partition-wide). First and foremost, the client keeps track of its files; servers do not need to manage copy-sets unless explicitly requested by client. Furthermore, a client can be using several different consistency policies for different files or even changing the consistency policy for a given file *at runtime*, without having to recompile or restart the file system or even the client-side daemon (Figure 1). All that is needed is that a desired policy be compiled as a

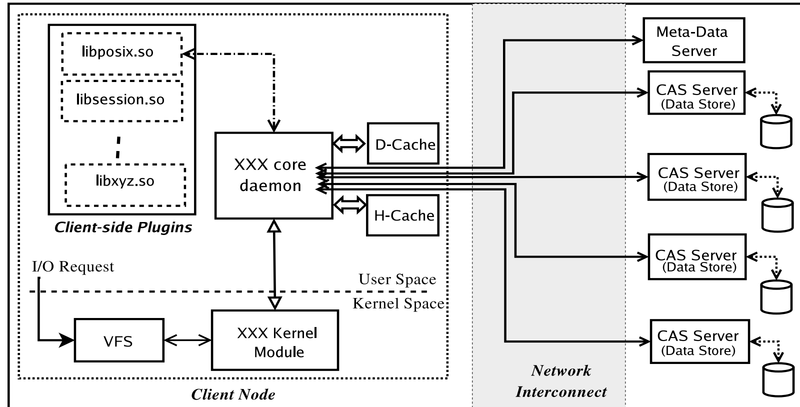



Figure 1: Design of the CAPFS parallel file system: Clients feature a user-space plug-in architecture, a content-addressable data cache(D-Cache), and an H-Cache(hash cache) to cache hashes of data blocks encountered.

plug-in and be installed in a special directory, after which the daemon is sent a signal to indicate the availability of a new policy. Leaving the choice of the consistency policy *and* allowing the user to change it at runtime enable tuning performance at a very fine granularity. However, one major underlying assumption in our system design is that we anticipate that the file system administrator sets the same policy on all the nodes of the cluster that accesses the file system. Handling conflicting consistency policies for the same file system or files could lead to incorrect execution of applications.

## 2.2 Lightweight Synchronization

Any distributed file system needs to provide a consistency protocol to arbitrate accesses to data and meta-data blocks. The consistency protocol needs to expose primitives both for atomic read/modify/write operations and for notification of updates to regions that are being managed. The former primitive is necessary to ensure that the state of the system is consistent in the presence of multiple updates, while the latter is necessary to incorporate client caching and prevent stale data from being read. Traditional approaches use locking to address both these issues.

### 2.2.1 To Lock or Not to Lock?

Some parallel cluster file systems (such as Lustre [6] and GPFS [27]) enforce data consistency by using file locks to prevent simultaneous file access from multiple clients. In a networked file system, this strategy usually involves acquiring a lock from a central lock manager on a file before proceeding with the write/read operation. Such a coarse-grained file locks-based approach ensures that only one process at a time can write data to a file. As the number of processes writing to the same file increases, performance (from lock contention) degrades rapidly. On the other hand, fine-grained file-locking schemes, such as byte-range locking, allow multiple processes to simultaneously write to different regions of a shared file. However, they also restrict scalability because of the overhead associated with maintaining state for a large number of locks, eventually leading to performance degradation. Furthermore, any networked locking system introduces a bottleneck for data access: the lock server.

The recent explosion in the scale of clusters, coupled with the emphasis on fault tolerance, has made traditional locking less suitable. GPFS [27], for instance, uses a variant of a distributed lock manager algorithm that essentially runs at two levels: one at a central server and the other on every client node. For efficiency reasons, clients can cache lock tokens on their files until they are explicitly revoked.

Such optimizations usually have hidden costs. For example, in order to handle situations where clients terminate while holding locks, complex lock recovery/release mechanisms are used. Typically, these involve some combination of a distributed crash recovery algorithm or a lease system [11]. Timeouts guarantee that lost locks can be reclaimed within a bounded time. Any lease-based system that wishes to guarantee a sequentially consistent execution must handle a race condition, where clients must finish their operation after acquiring the lock before the lease terminates. Additionally, the choice of the lease timeout is a tradeoff between performance and reliability concerns and further exacerbates the problem of reliably implementing such a system.

The pitfalls of using locks to solve the consistency problems in parallel file systems motivated us to investigate different approaches to providing the same functionality. We use a lockless approach for providing atomic file system data accesses. The approach to providing lockless, sequentially consistent data in the presence of concurrent conflicting accesses presented here has roots in three other transactional systems: store conditional operations in modern microprocessors [18], optimistic concurrency algorithms in databases [17], and optimistic concurrency approach in the Amoeba distributed file service [19].

Herlihy [13] proposed a methodology for constructing lock-free and wait-free implementations for highly

concurrent objects using the load-linked and store-conditional instructions. Our lockless approach, similar in spirit, does not imply the absence of any synchronization primitives (such as barriers) but, rather, implies the *absence of a distributed byte-range file locking service*. By taking an optimistic approach to consistency, we hope to gain on concurrency and scalability, while pinning our bets on the fact that conflicting updates (write-sharing) will be rare [4, 8, 21]. In general, it is well understood that optimistic concurrency control works best when updates are small or when the probability of simultaneous updates to the same item is small [19]. Consequently, we expect our approach to be ideal for parallel scientific applications. Parallel applications are likely to have each process write to distinct regions in a single shared file. For these types of applications, there is no need for locking, and we would like for all writes to proceed in parallel without the delay introduced by such an approach.

#### 2.2.2 Invalidates or Updates?

Given that client-side caching is a proven technique with apparent benefits for a distributed file system, a natural question that arises in the context of parallel file systems is whether the cost of keeping the caches coherent outweighs the benefits of caching. However, as outlined earlier, we believe that deciding to use caches and whether to keep them coherent should be the prerogative of the consistency policy and should not be imposed by the system. Thus, only those applications that require strict policies and cache coherence are penalized, instead of the whole file system. A natural consequence of opting to cache is the mechanism used to synchronize stale caches; that is, should consistency mechanisms for keeping caches coherent be based on expensive update-based protocols or on cheaper invalidation-based protocols or hybrid protocols?

Although update-based protocols reduce lookup latencies, they are not considered a suitable choice for workloads that exhibit a high degree of read-write sharing [3]. Furthermore, an update-based protocol is inefficient in its use of network bandwidth for keeping file system caches coherent, thus leading to a common adoption of invalidation-based protocols.

As stated before, parallel workloads do not exhibit much block-level sharing [8] . Even when sharing does occur, the number of consumers that actually read the modified data blocks is typically low. In Figure 2 we compute the number of consumers that read a block between two successive writes to the same block (we assume a block size of 4 KB). Upon normalizing against the number of times sharing occurs, we get the values plotted in Figure 2. This figure was computed from the traces of four parallel applications that were obtained from [31]. In other words, Figure 2 attempts to convey the amount of read-write sharing exhibited by typical parallel applications. It indicates that the number of consumers of a newly written block is very small (with the exception of LU, where a newly written block is read by all the remaining processes before the next write to the same block). Thus, an update-based protocol may be viable as long as the update mechanism does not consume too much network bandwidth. This result motivated us to consider content-addressable cryptographic hashes (such as SHA-1 [12]) for maintaining consistency because they allow for a bandwidth-efficient update-based protocol by transferring just the hash in place of the actual data. We defer the description of the actual mechanism to Section 3.5.

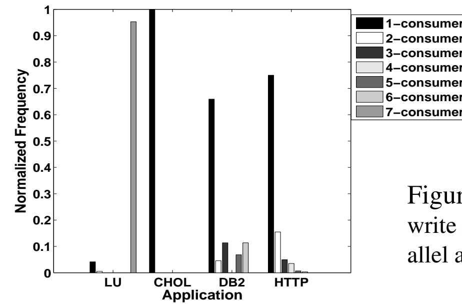



Figure 2: Read-write sharing for parallel applications.

#### 2.2.3 Content Addressability

Content addressability provides an elegant way to summarize the contents of a file. It provides the following advantages:

- The contents of a file can be listed as a concatenation of the hashes of its blocks. Such a representation was referred to as *recipes* in a previous study [30]. This approach provides a lightweight method of updating or invalidating sections of a file and so forth.
- It increases system concurrency, by not requiring synchronization at the content-addressable data servers (Figure 1). In comparison to versioning file systems that require a central version/time-stamp servers [19] or a distributed protocol for obtaining unique timestamps [10], a content-addressable system provides an independent, autonomous technique for clients to generate new version numbers for a block. Since newly written blocks will have new cryptographic checksums (assuming no hash collisions), a content-addressable data server also achieves the "no-overwrite" property that is essential for guaranteeing any sort of consistency.
- Using cryptographic hashes also allows for a bandwidth-efficient update-based protocol for maintaining cache coherence. This forms the basis for adopting a content-addressable storage server design in place of a traditional versioning mechanism. Additionally, it is foreseeable that the content-addressable nature of data may lead to easy replication schemes.
- Depending on the workload, content addressability might be able to reduce network traffic and storage demands. Blocks with the same content, if in the cache (because of commonality of data across files or within

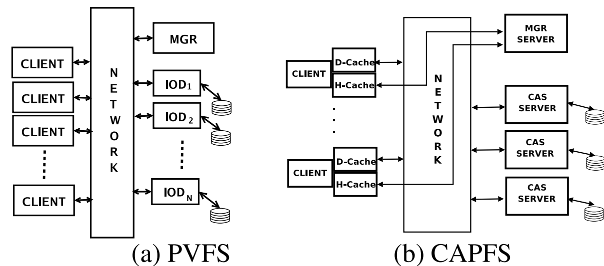
a file) do not need to be fetched or written. Only a single instance of the common block needs to be stored, leading to space savings.

As shown in Figure 1, the client employs two caches for performance. The H-Cache, or hash cache, stores all or a portion of a file's *recipe* [30]. A file in the CAPFS file system is composed of content-addressable chunks. Thus, a chunk is the unit of computation of cryptographic hashes and is also the smallest unit of accessibility from the CAS servers. The chunk size is crucial because it can impact the performance of the applications. Choosing a very small value of chunk size increases the CPU computation costs on the clients and the overheads associated with maintaining a large recipe file, while a very large value of chunk size may increase the chances of false sharing and hence coherence traffic. Thus, we leave this as a tunable knob that can be set by the plug-ins at the time of creation of a file and is a part of the file's meta-data. For our experiments, unless otherwise mentioned, we chose a default chunk size of 16 KB. The recipe holds the mapping between the chunk number and the hash value of the chunk holding that data. Using the H-Cache provides a lightweight method of providing updates when sharing occurs. An update to the hashes of a file ensures that the next request for that chunk will fetch the new content.

The D-Cache, or the data cache, is a content addressable cache. The basic object stored in the D-Cache is a chunk of data addressed by its SHA1-hash value. One can think of a D-cache as being a local replica of the CAS server's data store. When a section of a file is requested by the client, the corresponding data chunks are brought into the D-Cache. Alternatively, when the client creates new content, it is also cached locally in the D-Cache. The D-Cache serves as a simple cache with *no consistency requirements*. Since the H-caches are kept coherent (whenever the policy dictates), there is no need to keep the D-caches coherent. Additionally, given a suitable workload, it could also exploit commonality across data chunks and possibly across temporal runs of the same benchmark/application, thus potentially reducing latency and network traffic.

# 3 System Architecture

The goal of our system is to provide a robust parallel file system with good concurrency, high throughput and tunable consistency. The design of CAPFS resembles that of PVFS [7] in many aspects — central meta-data server, multiple data servers, RAID-0-style striping of data across the I/O servers, and so forth . The RAID-0 striping scheme also enables a client to easily calculate which data server has which data blocks of a file. In this section, we first take a quick look at the PVFS architecture and its limitations from the perspective of consistency semantics and then detail our system's design.



(a) PVFS (b) CAPFS

Figure 3: System architectures: CAPFS design incorporates two client-side caches that are absent in PVFS.

Figure 3 depicts a simplified diagram of the PVFS and CAPFS system architectures.

## 3.1 PVFS Architecture

The primary goal of PVFS as a parallel file system is to provide high-speed access to file data for parallel applications. PVFS is designed as a client-server system, as shown in Figure 3 (a).

PVFS uses two server components, both of which run as user-level daemons on one or more nodes of the cluster. One of these is a meta-data server (called MGR) to which requests for meta-data management (access rights, directories, file attributes, and physical distribution of file data) are sent. In addition, there are several instances of a data server daemon (called IOD), one on each node of the cluster whose disk is being used to store data as part of the PVFS name space. There are well-defined protocol structures for exchanging information between the clients and the servers. For instance, when a client wishes to open a file, it communicates with the MGR daemon, which provides it the necessary meta-data information (such as the location of IOD servers for this file, or stripe information) to do subsequent operations on the file. Subsequent reads and writes to this file do not interact with the MGR daemon and are handled directly by the IOD servers.

This strategy is key to achieving scalable performance under concurrent read and write requests from many clients and has been adopted by more recent parallel file system efforts. However, a flip-side to this strategy is that the file system does not guarantee any data consistency semantics in the face of conflicting operations or sessions. Fundamental problems that need to be addressed to offer sequential/ POSIX [28] style semantics are the *write atomicity* and *write propagation* requirements. Since file data is striped across different nodes and since the data is always overwritten, the I/O servers cannot guarantee write atomicity, and hence reads issued by clients could contain mixed data that is disallowed by POSIX semantics. Therefore, any application that requires sequential semantics must rely on external tools or higher-level locking solutions to enforce access restrictions. For instance, any application that relies on UNIX/POSIX semantics needs to use a distributed

cluster-wide lock manager such as the DLM [15] infrastructure, so that all `read/write` accesses acquire the appropriate file/byte-range locks before proceeding.

## 3.2 CAPFS: Servers

The underlying foundation for our system is the content-addressable storage model, wherein file blocks are *addressed and located* based on the cryptographic hashes of their contents. A file is logically split into fixed-size data chunks, and the hashes for these chunks are stored in the *hash server daemon*. The hash server daemon, analogous to the meta-data server (MGR) daemon of the PVFS system design, is responsible for mapping and storing the hashes of file blocks (termed recipes [30]) for all files. In essence, this daemon translates the logical block-based addressing mode to the content addressable scheme, that is, given a logical block $i$ of a particular file $F$, the daemon returns the hashes for that particular block. Even though in the current implementation there is a central server, work is under way to use multiple meta-data servers to serve a file's hashes for load-balancing purposes. Throughout the rest of the paper, we will use the term MGR server synonymously with hash server or meta-data server to refer to this daemon.

Analogous to the PVFS I/O server daemon is a content-addressable server (CAS) daemon, which supports a simple *get/put* interface to retrieve/store data blocks based on their cryptographic hashes. However, this differs significantly both in terms of functionality and exposed interfaces from the I/O servers of PVFS. Throughout the rest of this paper, we will use the term CAS server synonymously with data server to refer to this daemon.

## 3.3 CAPFS: Clients

The design of the VFS glue in CAPFS is akin to the upcall/downcall mechanism that was initially prototyped in the Coda [16] file system (and later adapted in many other file systems including PVFS). In this design, file system requests obtained from the VFS are queued in a device file and serviced by a user-level daemon. If an error is generated or if the operation completes successfully, the response is queued back into the device file, and the kernel signals the process that was waiting for completion of the operation. The client-side code intercepts these upcalls and funnels meta-data operations to the meta-data server. The data operations are striped to the appropriate CAS servers. Prototype implementations of the VFS glue are available at [1] for both Linux 2.4 and 2.6 kernels.

## 3.4 System Calls

The CAPFS system uses optimistic concurrency mechanisms to handle write atomicity on a central meta-data server, while striping writes in parallel over multiple content-addressable servers (CAS servers). The system has a lockless design: the only form of locking used is mutual-exclusion locks on the meta-data server to serialize the multiple threads (whenever necessary), as opposed to distributed locking schemes (such as DLM [15]).

### 3.4.1 Steps for the `open` and `close` System Call

- When a client wishes to open a file, a request is sent to the hash-server to query the hashes for the file if any.
- The server returns the list of hashes for the file (if the file is small). Hashes can also be obtained on demand from the server subsequently. The server also adds H-cache callbacks to this node for this file if requested.
- After the hashes are obtained, the client caches them locally (if specified by the policy) in the H-cache to minimize server load. H-cache coherence is achieved by having the server keep track of when commits are successful, and issuing callbacks to clients that may have cached the hashes. This step is described in greater detail in the subsequent discussions.
- On the last close of the file, all the entries in the H-cache for this file are invalidated for subsequent opens to reacquire, and if necessary the server is notified to terminate any callbacks for this node.

### 3.4.2 Steps for the `read` System Call

- The client tries to obtain the appropriate hashes for the relevant blocks either from the H-cache or from the hash server. An implicit agreement here is that the server promises to keep the client's H-cache coherent. This goal may be achieved by using either an update-based mechanism or an invalidation-based mechanism depending on the number of sharers. Note that the update callbacks contain merely the hashes and not the actual data.
- Using these hashes, it tries to locate the blocks in the D-cache. Note that keeping the H-cache coherent is enough to guarantee sequential consistency; nothing needs to be done for the D-cache because it is content addressable.
- If the D-cache has the requested blocks, the read returns and the process continues. On a miss, the client issues a *get* request to the appropriate CAS servers, which is cached subsequently. Consequently, reads in our system do not suffer any slowdowns and should be able to exploit the available bandwidth to the CAS servers by accessing data in parallel.

### 3.4.3 Steps for the `write` System Call

Writes from clients need to be handled a little differently because consistency guarantees may have to be met (depending on the policy). Since writes change the contents of the block, the cryptographic hashes for the block changes, and hence this is a new block in the system

altogether. We emphasize that we need mechanisms to ensure write atomicity not only across blocks but also across copies that may be cached on the different nodes. On a write to a block, the client does the following sequence of steps,

- Hashes for all the relevant blocks are obtained either from the H-cache or from the hash server.
- If the write spans an entire block, then the new hash can be computed locally by the client. Otherwise, it must read the block and compute the new hash based on the block's locally modified contents.
- After the old and new hashes for all relevant blocks are fetched or computed, the client does an *optimistic put* of the new blocks to the CAS servers, which store the new blocks. Note that by virtue of using content-addressable storage, the servers do not overwrite older blocks. This is an example of an optimistic update, because we assume that the majority of writes will be race-free and uncontested.
- If the policy requires that the writer's updates be made immediately visible, the next step is the *commit* operation. Depending on the policy, the client informs the server whether the commit should be forced or whether it can fail. Upon a successful commit, the return values are propagated back.
- A failed commit raises the possibility of *orphaned* blocks that have been stored in the I/O servers but are not part of any file. Consequently, we need a distributed cleaner process that is invoked when necessary to remove blocks that do not belong to any file. We refer readers to [1] for a detailed description of the cleaner protocol.

#### 3.4.4 Commit Step

- In the commit step, the client contacts the hash server with the list of blocks that have been updated, the set of old hashes, and the set of new hashes. In the next section, we illustrate the need for sending the old hashes, but in short they are used for detecting concurrent write-sharing scenarios similar to store-conditional operations [18].
- The meta-data server atomically compares the set of old hashes that it maintains with the set of old hashes provided by the client. In the uncontested case, all these hashes would match, and hence the commit is deemed race free and successful. The hash server can now update its recipe list with the new hashes. In the rare case of a concurrent conflicting updates, the server detects a mismatch in the old hashes reported for one or more of the client's commits and asks them to retry the entire operation. However, clients can override this by requesting the server to force the commit despite conflicts.
- Although such a mechanism has guaranteed write-atomicity across blocks, we still need to provide mechanisms to ensure that client's caches are also updated or invalidated to guarantee write atomicity across all copies of blocks that may be required by the consistency policy (sequential consistency/UNIX semantics require this). Since the server keeps track of clients that may have cached file hashes, a successful commit also entails updating or invalidating any client's H-cache with the latest hashes.
- Our system guarantees that updates to all locations are made visible in the same order to all clients (this mechanism is not exposed to the policies yet). Therefore, care must be exercised in the previous step to ensure that updates to all clients' H-caches are atomic. In other words, if multiple clients may have cached the hashes for a particular chunk and if the hash-server decides to update the hashes for the same chunk, the update-based protocol must use a two-phase commit protocol (such as those used in relational databases), so that all clients see the updates in the same order. This is not needed in an invalidation-based protocol however. Hence, we use an invalidation-based protocol in the cases of multiple readers/writers and an update-based protocol for single reader/writer scenarios.

### 3.5 Conflict Resolution

Figure 4 depicts a possible sequence of actions and messages that are exchanged in the case of multiple-readers and a single-writer client to the same file. We do not show the steps involved in opening the file and caching the hashes. In step 1, the writer optimistically writes to the CAS servers after computing the hashes locally. Step 2 is the request for committing the write sent to the hash server. Step 3 is an example of the invalidation-based protocol that is used in the multiple reader scenario from the point of view of correctness as well as performance. Our system resorts to an update-based protocol in the single sharer case. Sequential consistency requires that any update-based protocol has to be two-phased for ensuring the write-ordering requirements, and hence we opted to dynamically switch to using invalidation-based protocol in this scenario to alleviate performance concerns. Steps 5 and 6 depict the case where the readers look up the hashes and the local cache. Since the hashes could be invalidated by the writer, this step may also incur an additional network transaction to fetch the latest hashes for the appropriate blocks. After the hashes are fetched, the reader looks up its local data cache or sends requests to the appropriate data servers to fetch the data blocks. Steps 5 and 6 are shown in dotted lines to indicate the possibility that a network transaction may not be necessary if the requested hash and data are cached locally (which happens if both the `read`'s occurred before the `write` in the total ordering).

Figure 5 depicts a possible sequence of actions and messages that are exchanged in the case of multiple-readers and multiple-writers to the same file. As before, we do not show the steps involved in opening the file and caching the hashes. In step 1, writer client II optimistically writes to the CAS servers after computing

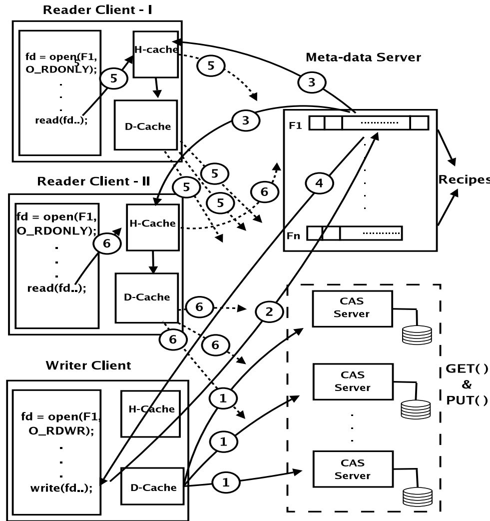



Figure 4: Action sequence: multiple-readers single-writer

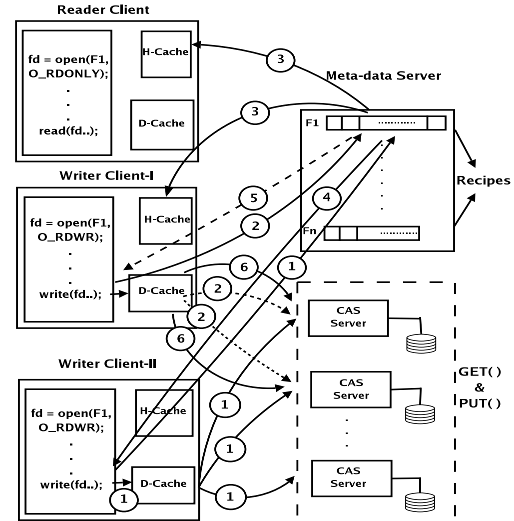



Figure 5: Action sequence: multiple-readers multiple-writers

hashes locally. In step 2, writer client I does the same after computing hashes locally. Both these writers have atleast one overlapping byte in the file to which they are writing (*true-sharing*) or are updating different portions of the same chunk (*false-sharing*). In other words this is an instance of concurrent-write sharing. Since neither writer is aware of the other's updates, one of them is asked to retry. The hash server acts as a serializing agent. Since it processes requests from client II before client I, the write from client II is successfully committed, and step 3 shows the invalidation messages sent to the reader and the writer client. Step 4 is the acknowledgment for the successful write commit. Step 5 is shown dashed to indicate that the hash server requests writer client I to retry its operation. The write done by this client in step 2 is shown dotted to indicate that this created orphaned blocks on the data server and needs to be cleaned. After receiving a reply from the hash server that the write needs to be retried, the writer client I obtains the latest hashes or data blocks to recompute its hashes and reissues the write as shown in step 6.

In summary, our system provides mechanisms to achieve serializability that can be used by the consistency policies if they desire. In our system, *read-write serializability* and *write atomicity across copies* are achieved by having the server update or invalidate the client's H-cache when a write successfully commits. *Write-write serializability across blocks* is achieved by having the clients send in the older hash values at the time of the commit to detect concurrent write-sharing and having one or more of the writers to restart or redo the entire operation.

We emphasize here that, since *client state is mostly eliminated*, there is no need for a complicated recovery process or lease-based timeouts that are an inherent part of distributed locking-based approaches. Thus, our proposed scheme is inherently more robust and fault tolerant from this perspective when H-caches are disabled. If H-caches are enabled however, temporary failures such as network disconnects can cause clients to read/write stale data. Further, the centralized meta-data server with no built-in support for replication is still a deterrent from the point of view of fault-tolerance and availability. We hope to address both these issues as future extensions.

## 3.6 Client-side Plug-in Architecture

The CAPFS design incorporates a client-side plug-in architecture that allows users to specify their own consistency policy to fine tune their application's performance. Figure 6 shows the hooks exported by the client-side and what callbacks a plug-in can register with the client-side daemon. Each plug-in is also associated with a "unique" name and identifier. The plug-in policy's name is used as a command-line option to the mount utility to indicate the desired consistency policy. The CAPFS client-side daemon loads default values based on the command-line specified policy name at mount time. The user is free to define any of the callbacks in the plug-ins (setting the remainder to NULL), and hence choosing the best trade-off between throughput and consistency for the application. The plug-in API/callbacks to be defined by the user provide a flexible and extensible way of defining a large range of (possibly non-standard) consistency policies. Additionally, other optimizations such as prefetching of data or hashes, delayed commits, periodic commits(e.g., commit after "t" units of time, or commit after every "n" requests), and others can be accommodated by the set of callbacks shown in Figure 6). For standard cases, we envision that the callbacks be used as follows.

**Setting Parameters at Open:** On mounting the CAPFS file system, the client-side daemon loads

```
struct plugin_policy_ops {
handle (*pre_open)(force_commit, use_hcache,
    hcache_coherence, delay_commit, num_hashes);
int (*post_open)(void *handle);
int (*pre_close)(void *handle);
int (*post_close)(void *handle);
int (*pre_read)(void *handle, size, offset);
int (*post_read)(void *handle, size, offset);
int (*pre_write)(void *handle, size, offset,
    int *delay_wc);
int (*post_write)(void *handle, sha1_hashes *old,
    sha_hashes *new);
int (*pre_sync)(const char *);
int (*post_sync)(void *handle);
};
```

Client-Side Plug-in API

```
int hcache_get(void *handle, begin_chunk, nchunks,
      void *buf);
int hcache_put(void *handle, begin_chunk, nchunks,
      const void *buf);
int hcache_clear(void *handle);
int hcache_clear_range(void *handle, begin_chunk,
      nchunks);
void hcache_invalidate(void);

int dcache_get(char *hash, void *buf, size);
int dcache_put(char *hash, const void *buf, size);
int commit(void *handle, sha1_hashes *old_hashes,
      sha1_hashes *new_hashes,
      sha1_hashes *current_hashes);
```

CAPFS Client-Daemon: Core API

Figure 6: The client-side plug-in API and the CAPFS client-daemon core API. On receiving a system call, the CAPFS client-daemon calls the corresponding user-defined pre- and post- functions, respectively, before servicing the system call.

default values for force_commit, use_hcache, hcache_coherence, delay_commit, and num_hashes parameters. However, these values can be overridden on a per-file basis as well by providing a non-NULL pre_open callback. Section 3.4.4 indicates that in a commit operation, a client tells the server what it thinks the old hashes for the data are and then asks the server to replace them with new, locally calculated hashes. Hence a commit operation fails if the old hashes supplied by the client do not match the ones currently on the server (because of intervening commits by other clients). On setting the force_commit parameter, the client forces the server into accepting the locally computed hashes, overwriting whatever hashes the server currently has. The use_hcache parameter indicates whether the policy desires to use the H-Cache. The hcache_coherence parameter is a flag that indicates to the server the need for maintaining a coherent H-cache on all the clients that may have stale entries. The delay_commit indicates whether the commits due to writes should be delayed (buffered) at the client. The num_hashes parameter specifies how many hashes to fetch from the meta-data server at a time. These parameters can be changed by the user by defining a pre_open callback in the plug-in (Figure 6). This function returns a handle, which is cached by the client and is used as an identifier for the file. This handle is passed back to the user plug-in in post_open and other subsequent callbacks until the last reference to the file is closed. For instance, a plug-in implementing an AFS session like semantics [14] would fetch all hashes at the time of open, delay the commits till the time of a close, set the force_commit flag and commit all the hashes of a file at the end of the session.

**Prefetching and Caching:** Prior to a read, the client daemon invokes the pre_read callback (if registered). We envision that the user might desire to check H-Cache and D-Cache and fill them using the appropriate hcache_get/dcache_get API (Figure 6) exported by the client daemon. This callback might also be used to implement prefetching data, hashes, and the like.

**Delayed commits:** A user might overload the pre_write callback routine to implement delayed commits over specific byte ranges. One possible way of doing this is to have the pre_write callback routine set a timer (in case a policy wishes to commit every "t" units of time) that would invoke the post_write on expiration. But for the moment, pre_write returns a value for delay_wc (Figure 6) to indicate to the core daemon that the write commits need to be delayed or committed immediately. Hence, on getting triggered, the post_write checks for pending commits and then initiates them by calling the appropriate core daemon API (commit). The post_write could also handle operations such as flushing or clearing the caches.

**Summary:** The callbacks provide enough flexibility to let the user choose when and how to implement most known optimizations (delayed writes, prefetching, caching, etc.) in addition to specifying any customized consistency policies. By passing in the offsets and sizes of the operations to the callback functions such as pre_read, pre_write, plug-in writers can also use more specialized policies at a very fine granularity (such as optimizations making use of MPI derived datatypes [9]). This description details just one possible way of doing things. Users can use the API in a way that suits their workload, or fall back on standard predefined policies. Note that guaranteeing correctness of execution is the prerogative of the plug-in writer. Implementation of a few standard policies (Sequential, SESSION-like, NFS-like) and others (Table 1 in Section 4) indicate that this step does not place an undue burden on the user. The above plug-ins were implemented in less than 150 lines of C code.

# 4 Experimental Results

Our experimental evaluation of CAPFS was carried out on an IBM pSeries cluster. with the following configura-

tion. There are 20 compute nodes each of which is a dual hyper-threaded Xeon clocked at 2.8 GHz, equipped with 1.5 GB of RAM, a 36 GB SCSI disk and a 32-bit Myrinet card (LANai9.0 clocked at 134 MHz). The nodes run Redhat 9.0 with Linux 2.4.20-8 kernel compiled for SMP use and GM 1.6.5 used to drive the Myrinet cards. Our I/O configuration includes 16 CAS servers with one server doubling as both a meta-data server and a CAS server. All newly created files are striped with a stripe size of 16 KB and use the entire set of servers to store the file data. A modified version of MPICH 1.2.6 distributed by Myricom for GM was used in our experimental evaluations.

| Policy Name | Use Hcache | Force Commit | Hcache Coherence |
|---|---|---|---|
| SEQ-1 | 0 | 0 | X |
| SEQ-2 | 1 | 0 | 1 |
| FOR-1 | 0 | 1 | X |
| FOR-2 | 1 | 1 | 1 |
| REL-1 | 1 | 1 | 0 |

Table 1: Design space constituting a sample set of consistency policies: SEQ-1, SEQ-2 implement sequential consistency; FOR-1, FOR-2 implement a slightly relaxed mechanism where commits are forced; REL-1 implements an even more relaxed mechanism. The X in rows 1 and 3 denotes a don't care for the variable's value.

## 4.1 Aggregate Bandwidth Tests

Since the primary focus of parallel file systems is aggregate throughput, our first workload is a parallel MPI program(*pvfs_test.c* from the PVFS distribution), that determines the aggregate read/write bandwidths and verifies correctness of the run. The block sizes, iteration counts, and number of clients are varied in different runs. Consequently, this workload demonstrates concurrent-write sharing and sequential-write sharing patterns, albeit not simultaneously. Times for the read/write operations on each node are recorded over ten iterations and the maximum averaged time over all the tasks is used to compute the bandwidth achieved. The graphs for the above workload plot the aggregate bandwidth (in MB/s) on the y-axis against the total data transferred to or from the file system (measured in MB). The total data transferred is the product of the number of clients, block size and the number of iterations.

We compare the performance of CAPFS against a representative parallel file system – PVFS (Version 1.6.4). To evaluate the flexibility and fine-grained performance tuning made possible by CAPFS' plug-in infrastructure, we divide our experimental evaluation of into categories summarized in Table 1. Five simple plug-ins have been implemented to demonstrate the performance spectrum.

The values of the parameters in Table 1 — (*force_commit, hcache_coherence* and *use_hcache*) dictate the consistency policies of the file system. The *force_commit* parameter indicates to the meta-data server that the commit operation needs to be carried out without checking for conflicts and being asked to retry. Consequently, this parameter influences write performance. Likewise, the *hcache_coherence* parameter indicates to the meta-data server that a commit operation needs to be carried out in strict accordance with the H-cache coherence protocol. Since the commit operation is not deemed complete until the H-cache coherence protocol finishes, any consistency policy that relaxes this requirement is also going to show performance improvements for writes. Note that neither of these two parameters is expected to have any significant effect on the read performance of this workload. On the other hand, using the H-cache on the client-side (*use_hcache* parameter) has the potential to improving the read performance because the number of RPC calls required to reach the data is effectively halved.

The first two rows of Table 1 illustrate two possible ways of implementing a sequentially consistent file system. The first approach denoted as SEQ-1, does not use the H-cache (and therefore H-caches need not be kept coherent) and does not force commits. The second approach denoted as SEQ-2, uses the H-cache, does not force commits, and requires that H-caches be kept coherent. Both approaches implement a sequentially consistent file system image and are expected to have different performance ramifications depending on the workload and the degree of sharing.

The third and fourth rows of Table 1 illustrate a slightly relaxed consistency policy where the commits are forced by clients instead of retrying on conflicts. The approach denoted as FOR-1, does not use the H-cache (no coherence required). The approach denoted as FOR-2, uses the H-cache and requires that they be kept coherent. One can envisage that such policies could be used in mixed-mode-environments where files are possibly accessed or modified by nonoverlapping MPI jobs as well as unrelated processes.

The fifth row of Table 1 illustrates an even more relaxed consistency policy denoted as REL-1, that forces commits, uses the H-cache, and does not require that the H-caches be kept coherent. Such a policy is expected to be used in environments where files are assumed to be non-shared among unrelated process or MPI-based applications or in scenarios where consistency is not desired. Note that it is the prerogative of the application-writer or plug-in developers to determine whether the usage of a consistency policy would violate the correctness of the application's execution.

**Read Bandwidth:** In the case of the aggregate read bandwidth results (Figures 7(a) and 7(b)), the policies using the H-cache (SEQ-2, FOR-2, REL-1) start to perform better in comparison to both PVFS and policies not using the H-cache (SEQ-1, FOR-1). This tipping point occurs when the amount of data being transferred is fairly large (around 3 GB). This is intuitively correct, because the

larger the file, the greater the number of hashes that need to be obtained from the meta-data server. This requirement imposes a higher load on the server and leads to degraded performance for the uncached case. The sharp drop in the read bandwidth for the H-cache based policies (beyond 4 GB) is an implementation artifact caused by capping the maximum number of hashes that can be stored for a particular file in the H-cache.

On the other hand, reading a small file requires proportionately fewer hashes to be retrieved from the server, as well as fewer RPC call invocations to retrieve the entire set of hashes. In this scenario, the overhead of indexing and retrieving hashes from the H-cache is greater than the time it takes to fetch all the hashes from the server in one shot. This is responsible for the poor performance of the H-cache based policies for smaller file sizes. In fact, a consistency policy that utilizes the H-cache allows us to achieve a peak aggregate read bandwidth of about 450 MB/s with 16 clients. This is almost a 55% increase in peak aggregate read bandwidth in comparison to PVFS which achieves a peak aggregate read bandwidth of about 290 MB/s. For smaller numbers of clients, even the policies that do not make use of the H-cache perform better than PVFS.

In summary, for medium to large file transfers, from an aggregate read bandwidth perspective, consistency policies using the H-cache (SEQ-2, FOR-2, REL-1) outperform both PVFS and consistency policies that do not use the H-cache (SEQ-1, FOR-1).

**Write Bandwidth:** As explained in Section 3.3, write bandwidths on our system are expected to be lower than read bandwidths and these can be readily corroborated from Figures 7(c) and 7(d). We also see that PVFS performs better than all of our consistency policies for smaller data transfers (upto around 2 GB). At around the 1.5–2 GB size range, PVFS experiences a sharp drop in the write bandwidth because the data starts to be written out to disk on the I/O servers that are equipped with 1.5 GB physical memory. On the other hand no such drop is seen for CAPFS. The benchmark writes data initialized to a repeated sequence of known patterns. We surmise that CAPFS exploits this commonality in the data blocks, causing the content-addressable CAS servers to utilize the available physical memory more efficiently with fewer writes to the disk itself.

At larger values of data transfers (greater than 2 GB), the relaxed consistency policies that use the H-cache (REL-1, FOR-2) outperform both PVFS and the other consistency policies (SEQ-1, SEQ-2, FOR-1). This result is to be expected, because the relaxed consistency semantics avoid the expenses associated with having to retry commits on a conflict and the H-cache coherence protocol. Note that the REL-1 scheme outperforms the FOR-2 scheme as well, since it does not perform even the H-cache coherence protocol. Using the REL-1 scheme, we obtain a peak write bandwidth of about 320 MB/s with 16 clients, which is about a 12% increase in peak aggregate write bandwidth in comparison to that of PVFS, which achieves a peak aggregate write bandwidth of about 280 MB/s.

These experiments confirm that performance is directly influenced by the choice of consistency policies. Choosing an overly strict consistency policy such as SEQ-1 for a workload that does not require sequential consistency impairs the possible performance benefits. For example, the write bandwidth obtained with SEQ-1 decreased by as much as 50% in comparison to REL-1. We also notice that read bandwidth can be improved by incorporating a client-side H-cache. For example, the read bandwidth obtained with SEQ-2 (FOR-2) increased by as much as 80% in comparison to SEQ-1 (FOR-1). However, this does not come for free, because the policy may require that the H-caches be kept coherent. Therefore, using a client-side H-cache may have a detrimental effect on the write bandwidth. All of these performance ramifications have to be carefully addressed by the application designers and plug-in writers before selecting a consistency policy.

## 4.2 Tiled I/O Benchmark

Tiled visualization codes are used to study the effectiveness of today's commodity-based graphics systems in creating parallel and distributed visualization tools. In this experiment, we use a version of the tiled visualization code [24] that uses multiple compute nodes, where each compute node takes high-resolution display frames and reads only the visualization data necessary for its own display.

We use nine compute nodes for our testing, which mimics the display size of the visualization application. The nine compute nodes are arranged in the 3 x 3 display as shown in Figure 8, each with a resolution of 1024 x 768 pixels with 24-bit color. In order to hide the merging of display edges, there is a 270-pixel horizontal overlap and a 128-pixel vertical overlap. Each frame has a file size of about 118 MB, and our experiment is set up to manipulate a set of 5 frames, for a total of about 600 MB.

This application can be set up to run both in collective I/O mode [9], wherein all the tasks of the application perform I/O collectively, and in noncollective I/O mode. Collective I/O refers to an MPI I/O optimization technique that enables each processor to do I/O on behalf of other processors if doing so improves the overall performance. The premise upon which collective I/O is based is that it is better to make large requests to the file system and cheaper to exchange data over the network than to transfer it over the I/O buses. Once again, we compare CAPFS against PVFS for the policies described earlier in Table 1. All of our results are the average of five runs.

**Read Bandwidth:** The aggregate read bandwidth plots (Figures 9(a) and 9(c)), indicate that CAPFS outperforms PVFS for both the noncollective and the col-

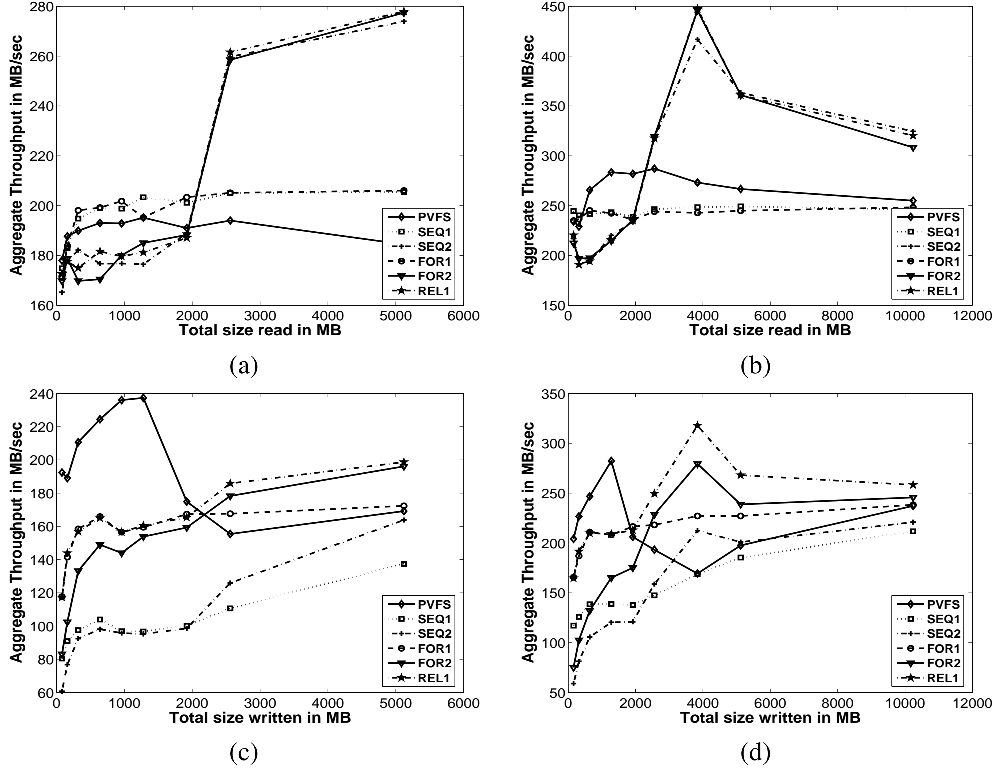



Figure 7: Aggregate Bandwidth in MB/s with varying block sizes: CAPFS vs. PVFS for (a) read-8-clients, (b) read-16-clients, (c) write-8-clients, (d) write-16-clients.

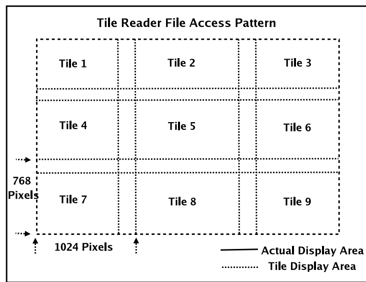



Figure 8: Tile reader file access pattern: Each processor reads data from a display file onto local display (also known as a tile).

lective I/O scenarios, across all the consistency policies. Note that the read phase of this application can benefit only if the policies use the H-caches (if available). As we saw in our previous bandwidth experiments, benefits of using the H-cache start to show up only for larger file sizes. Therefore, read bandwidths for policies that use the H-cache are not significantly different from those that don't in this application. Using our system, we achieve a maximum aggregate read bandwidth of about 90 MB/s without collective I/O and about 120 MB/s with collective I/O. These results translate to a performance improvement of 28% over PVFS read bandwidth for the noncollective scenario and 20% over PVFS read bandwidth for the collective scenario.

**Write Bandwidth:** The aggregate write bandwidths paint a different picture. For noncollective I/O, Figure 9 (b), the write bandwidth is very low for two of our policies (SEQ-2, FOR-2). The reason is that both these policies use an H-cache and also require that the H-caches be kept coherent. Also, the noncollective I/O version of this program makes a number of small write requests. Consequently, the number of H-cache coherence messages (invalidates) also increases, which in turn increases the time it takes for the writes to commit at the server. One must also bear in mind that commits to a file are serialized by the meta-data server and could end up penalizing other writers that are trying to write simultaneously to the same file. Note that the REL-1 policy does not lose out on write performance despite using the H-cache, since commits to the file do not execute the expensive H-cache coherence protocol. In summary, this result indicates that if a parallel workload performs a lot of small updates to a shared file, then any consistency policy that requires H-caches to be kept coherent is not appropriate from a performance perspective.

Figure 9(d) plots the write bandwidth for the collective I/O scenario. As stated earlier, since the collective I/O optimization makes large, well-structured requests to the file system, all the consistency policies (including the ones that require coherent H-caches) show a marked improvement in write bandwidth. Using our system, we achieve a maximum aggregate write bandwidth of about 35 MB/s without collective I/O and about 120 MB/s with

collective I/O. These results translate to a performance improvement of about 6% over PVFS write bandwidth for the noncollective scenario and about 13% improvement over PVFS write bandwidth for the collective scenario.

## 5 Related Work

Providing a plug-in architecture for allowing the user to *define* their own consistency policies for a parallel file system is a contribution unique to CAPFS file system. Tunable consistency models and tradeoffs with availability have been studied in the context of replicated services by Yu et al. [32].

Distributed file systems such as AFS [14], NFS [26] and Sprite [4, 20] have only a single server that doubles in functionality as a meta-data and data server. Because of the centralized nature of the servers, write atomicity is fairly easy to implement. Client-side caches still need to be kept consistent however, and it is with respect to this issue (write propagation) that these approaches differ from the CAPFS architecture. Coda [16] allows for server replication and it solves the write atomicity problem by having modifications propagated in parallel to all available replica servers (volume storages), and eventually to those that missed the updates.

Parallel file systems such as GPFS [27] and Lustre [6] employ distributed locking to synchronize parallel read-write disk accesses from multiple client nodes to its shared disks. The locking protocols are designed to allow maximum throughput, parallelism, and scalability, while simultaneously guaranteeing that file system consistency is maintained. Likewise, the Global File System (GFS) [22, 23] (a shared-disk, cluster file system) uses fine-grained SCSI locking commands, lock-caching and callbacks for performance and synchronization of accesses to shared disk blocks, and leases, journalling for handling node failures and replays. Although such algorithms can be highly tuned and efficient, failures of clients can significantly complicate the recovery process. Hence any locking-based consistency protocol needs additional distributed crash recovery algorithms or lease-based timeout mechanisms to guarantee correctness. The CAPFS file system eliminates much of the client state from the entire process, and hence client failures do not need any special handling.

Sprite-LFS [25] proposed a new technique for disk management, where all modifications to a file system are recorded sequentially in a log, which speeds crash recovery and writes. An important property in such a file system is that no disk block is ever overwritten (except after a disk block is reclaimed by the cleaner). Content-addressability helps the CAPFS file system gain this property, wherein updates from a process do not overwrite any existing disk or file system blocks. Recently, content-addressable storage paradigms have started to evolve that are based on distributed hash tables like Chord [29]. A key property of such a storage system is that blocks are addressed by the cryptographic hashes of their contents, like SHA-1 [12]. Tolia et al. [30] propose a distributed file system CASPER that utilizes such a storage layer to opportunistically fetch blocks in low-bandwidth scenarios. Usage of cryptographic content hashes to represent files in file systems has been explored previously in the context of Single Instance Storage [5], Farsite [2], and many others. Similar to log-structured file systems, these storage systems share a similar no-overwrite property because every write of a file/disk block has a different cryptographic hash (assuming no collisions). CAPFS uses content-addressability in the hope of minimizing network traffic by exploiting commonality between data block, and to reduce synchronization overheads, by using hashes for cheap update based synchronization. The no-overwrite property that comes for free with content addressability has been exploited to provide extra concurrency at the data servers.

## 6 Concluding Remarks

In this paper, we have presented the design and implementation of a robust, high-performance parallel file system that offers user-defined consistency at a user-defined granularity using a client-side plug-in architecture. To the best of our knowledge CAPFS is the only file system that offers tunable consistency that is also user-defined and user-selectable at runtime. Rather than resorting to locking for enforcing serialization for read-write sharing or write-write sharing, CAPFS uses an optimistic concurrency control mechanism. Unlike previous network/parallel file system designs that impose a consistency policy on the users, our approach provides the mechanisms and defers the policy to application developers and plug-in writers.

## 7 Acknowledgments

We thank Robert Ross, Rajeev Thakur, and all the anonymous reviewers for their suggestions, which have greatly improved the content of this paper. This work was supported in part by Pittsburgh Digital Greenhouse, NSF ITR-0325056 and CCR-0130143.

## Notes

[1] Work done while at Penn State University. Current Address: Mathematics and Computer Science Division, Argonne National Laboratory, IL 60439; vilayann@mcs.anl.gov.

## References

[1] Technical Report and CAPFS Source code. *http://www.cse.psu.edu/∼vilayann/capfs/*.

[2] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer.

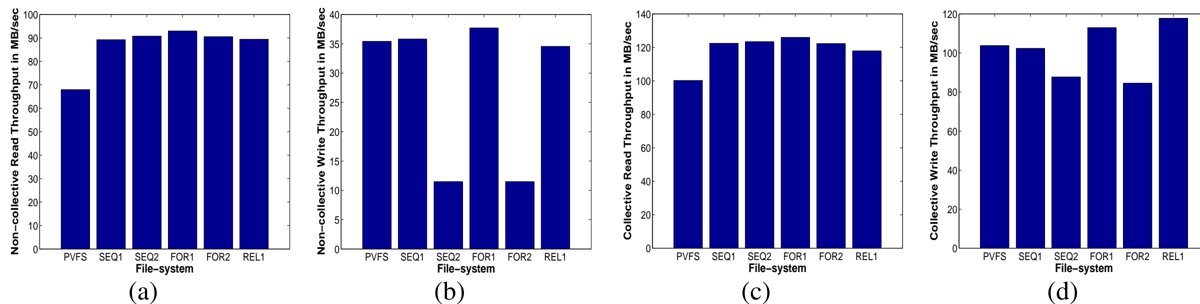



Figure 9: Tile I/O benchmark bandwidth in MB/s: (a) noncollective read, (b) noncollective write, (c) collective read, (d) collective write.


FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, 2002.

[3] M. B. Alexander. *Assessment of Cache Coherence Protocols in Shared-Memory*. PhD thesis, University of Toronto, 2003.

[4] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery SIGOPS, 1991.

[5] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, August 2000.

[6] P. J. Braam. The Lustre Storage Architecture, http://www.lustre.org/documentation.html, August 2004.

[7] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[8] P. F. Corbett and D. G. Feitelson. The Vesta Parallel File System. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. IEEE Computer Society Press and Wiley, 2001.

[9] M. P. I. Forum. MPI-2: Extensions to the Message-Passing Interface, 1997. http://www.mpi-forum.org/docs.

[10] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-Tolerant Erasure-Coded Storage. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, 2004.

[11] C. Gray and D. Cheriton. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 1989.

[12] N. W. Group. RFC 3174 - US Secure Hash Algorithm - I (SHA1), 2001.

[13] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Language Systems*, 15(5), 1993.

[14] 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 Transactions on Computer Systems*, 6(1), 1988.

[15] IBM. Distributed Lock Manager for Linux, 2001. http://oss.software.ibm.com/dlm.

[16] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 1991.

[17] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2), 1981.

[18] C. May, E. Silha, R. Simpson, and H. Warren. The PowerPC Architecture: A Specification for a New Family of RISC Processors, 1994. Morgan Kaufman, San Francisco, CA, Second Edition.

[19] S. J. Mullender and A. S. Tanenbaum. A Distributed File Service based on Optimistic Concurrency Control. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1985.

[20] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*. ACM Press, 1987.

[21] B. D. Noble and M. Satyanarayanan. An Empirical Study of a Highly Available File System. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and Modeling of Computer Systems*, 1994.

[22] K. W. Preslan and A. P. Barry. A 64-bit, Shared Disk File System for Linux. Technical report, Sistina Software, Inc, 1999.

[23] K. W. Preslan, A. P. Barry, J. Brassow, R. Cattelan, A. Manthei, E. Nygaard, S. V. Oort, D. Teigland, M. Tilstra, M. O'Keefe, G. Erickson, and M. Agarwal. Implementing Journaling in a Linux Shared Disk File System. In *IEEE Symposium on Mass Storage Systems*, 2000.

[24] R. B. Ross. Parallel I/O Benchmarking Consortium, http://www-unix.mcs.anl.gov/ rross/pio-benchmark.

[25] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1), 1992.

[26] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, 1985.

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

[28] I. Standard. 1003.1 Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API) [C Language]., 1996.

[29] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Trans. Netw.*, 11(1), 2003.

[30] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, A. Perrig, and T. Bressoud. Opportunistic Use of Content Addressable Storage for Distributed File Systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, 2003.

[31] M. Uysal, A. Acharya, and J. Saltz. Requirements of I/O Systems for Parallel Machines: An Application-driven Study. Technical Report CS-TR-3802, University of Maryland, College Park, MD, 1997.

[32] H. Yu. TACT: Tunable Availability and Consistency Tradeoffs for Replicated Internet Services (poster session). *SIGOPS Operating Systems Review*, 2000.