Application-Tailored Cache Consistency for Wide-Area File Systems

Ming Zhao Renato J. Figueiredo
Advanced Computing and Information Systems Laboratory (ACIS)
Electrical and Computer Engineering, University of Florida
{ming, renato}@acis.ufl.edu

Abstract

The inability to perform optimizations based on application-specific information presents a hurdle to the deployment of pervasive LAN file systems across WAN environments. This paper proposes a novel approach addressing this problem through application-tailored caching and consistency in wide-area file systems. It leverages widely available Network File System (NFS) deployments without any modifications to kernels nor applications, and employs middleware to dynamically establish Grid-wide Virtual File System (GVFS) sessions with application-tailored cache consistency. Two consistency models are discussed in this paper: a relaxed model based on invalidation polling, and a stronger model based on delegation and callback. Experimental evaluation based on microbenchmarks and scientific applications show that with application-tailored cache consistency, GVFS is able to both improve application runtimes and reduce server load significantly, compared to kernel-level NFS in WAN.

1. Introduction

This paper addresses the lack of support in current Distributed File Systems (DFSs) for application-tailored caching and consistency models. Central to the proposed approach is the use of a virtualization layer based on user-level DFS proxies [12][24], and the role of middleware as the entity that customizes and creates Grid-wide Virtual File System (GVFS) sessions on demand [1][4][12]. The approach is applicable to a wide variety of systems because it leverages the NFS [6] de-facto standard, is transparent to applications, and requires no kernel modifications to be deployed.

The importance of this approach is that it provides an effective way to support high-performance data access and consistency in cross-domain wide area computing environments, e.g. in support of high-throughput scientific and financial workloads [4]. In such environments, statically established DFSs are unable to cater to application-specific needs. In contrast, related work has shown that virtualized file system sessions can be scheduled by middleware on behalf of users [12][22][31]. This paper presents novel techniques that extend user-level virtualized DFS to support application-tailored caching with strong or weak consistency models that overlay native mechanisms used by NFS. The resulting design enables a middleware scheduler to control caching and consistency policies, on a per-session basis.

User-level implementations incur more overhead compared to kernel-level; however, in many environments kernel DFS changes tailored to application needs are not viable. To evaluate the performance of the proposed techniques in wide area environments, a series of experiments are reported in this paper. These consider the performance of several microbenchmarks and scientific applications. The results show that with application-tailored cache consistency, GVFS is able to both improve application runtimes and reduce server load significantly, in comparison to kernel-level NFS implementations.

In the rest of the paper, Section 2 describes background and related work, Section 3 highlights motivating examples, Section 4 discusses cache consistency models, Section 5 presents experimental evaluation, and Section 6 concludes the paper.

2. Background and Related Work

Currently there are no mechanisms that allow a conventional DFS implementation to be customized to support application- and user-tailored enhancements. This presents a hurdle to the deployment of pervasive LAN file systems (e.g. NFS v2/v3) across WAN environments, where round-trip latencies are considerably larger. If DFSs are capable of leveraging application knowledge, the number of client-server interactions can be reduced, thereby reducing server loads and average request latencies. However, typical DFS implementations are not designed to exploit such knowledge, for two important reasons.

First, traditionally DFSs are setup by system administrators with static, long-lived, homogeneous configurations at the granularity of a collection of users, rather than dynamic, short-lived, customized setups at the granularity of an application session. Second, integrating application-tailored features with DFS implementations in commonly available kernels is very difficult in practice. An optimization tailored for one application (e.g. aggressive pre-fetching of file contents) may result in performance degradation for several others (e.g. sparse files, databases). In addition, kernel-level modifications are difficult to port and deploy, notably in shared environments.

The lack of support for application-tailored optimizations has also been recognized as a limitation by BAD-FS [4]. However, it relies on system-call and library based interposition agents, and hence does not support many applications and OSs. Other system-call and library based extensions have been investigated in [2], [22]. However, it is hard to duplicate kernel functionality [17] and present full file system semantics [24]. In contrast, GVFS is mounted in the same way as conventional NFS, and supports a wide range of unmodified applications and OSs.

Several related systems have leveraged user-level techniques based on loop-back server/client proxies to extend file system O/S functionality - in essence, virtualizing DFSs by means of intercepting RPC calls of protocols such as NFS [6], e.g., the automounter [7], CFS [5], SFS [14] and LegionFS [31]. This paper differentiates from these efforts in that Grid middleware is used to setup, create and destroy DFS sessions on a per-application basis.

There are related kernel-level DFS solutions that exploit the advantages of disk caching (AFS [18], CacheFS [30]), or support different consistency models for improved performance (NQ-NFS [23], Spritely NFS [29], NFS v4 [6]) However, these designs require kernel support that is difficult to deploy across shared Grid environments, and they are not able to employ per-user/-application cache policies.

Scalable distributed data storage/delivery has been pursued in related work, e.g. Pangaea [26] and OceanStore [20]. Pangaea supports only one consistency model - eventual consistency; OceanStore allows for application-specific consistency, but it is not application-transparent, requiring the use of its API to achieve this goal. In the context of Web content caching, a related proxy cache invalidation approach has been studied in [16]. These systems differ from this paper in that they are not architected to allow middleware to dynamically instantiate, configure and compose proxies for application-tailored data sessions.

3. Motivating Examples

Distributed systems that allow the provisioning of general-purpose computing as a utility ("Grids" [13]) have the potential to enable on-demand access to unprecedented computing power. A key challenge arising in such systems is data management - how to seamlessly provide data to applications in WAN environments. DFS-based techniques are key to supporting applications without modifications to source code, libraries, or binaries. Examples include commercial, interactive scientific and engineering tools and virtual machine monitors that often operate on large, sparse data sets [19][27][32].

While application transparency is an asset of approaches based on DFSs, it can also become a performance liability. Enhancements that target wide-area DFSs for shared Grid environments are desirable, but need to be considered in a context where modifications tailored to this application domain are unlikely to be implemented in kernels. Nonetheless, recent work has shown the feasibility of applying user-level techniques to improve the performance of wide-area file systems [14][24][32], motivating the pursuit of user level application-tailored extensions. Potential uses of application-tailored cache consistency can be illustrated with three concrete scenarios:

Distributed Virtual Machines: There are growing interests in employing Virtual Machine (VM) in Grid computing [11][9]. Typical VM technologies (e.g. [28][10][3]) encapsulate a VM's state in regular files or filesystems and thus can leverage DFS support [32]. A VM with a non-persistent disk state can be used as a “master” image for the purpose of VM cloning; and under the management of a VM scheduler [21], such “clones” can be dedicated to executions of individual applications. In this scenario, the “master” image can be read-only shared while each clone has its own redo log or copy-on-write state. Hence it is reasonable to enable aggressive caching for both reads and writes.

Software Repositories: Software repositories are popular in enterprises as a means of sharing software among users. Such repositories are often setup on a DFS in an enterprise local network, read-only shared by organization users and centrally managed by system administrators. However, as the resources and users grow, support for wide-area sharing becomes a challenge to traditional DFS technologies. In this context, leveraging the processing and disk storage at

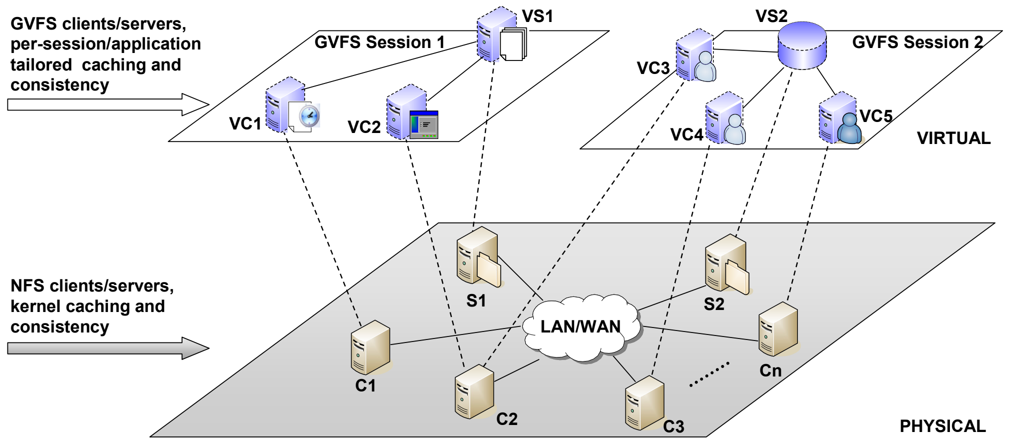
GVFS clients/servers, per-session/application tailored caching and consistency
GVFS Session 1
VS1
VC1
VC2
VS2
GVFS Session 2
VC3
VC4
VC5
VIRTUAL
NFS clients/servers, kernel caching and consistency
S1
S2
LAN/WAN
C1
C2
C3
Cn
PHYSICAL

Figure 1: GVFS sessions consist of virtual clients (VC1-VC5) and servers (VS1-VS2) implemented by user-level proxies. They are dynamically established and managed by middleware and overlay shared physical resources (C1-Cn, S1-S2). Each GVFS session can employ independent application tailored user-level disk caching and consistency model. E.g., Session 1 applies the delegation callback based model (Section 4.3) and support a scenario where real-time data are collected on-site (VC1) and processed off-site (VC2); Session 2 uses the invalidation polling protocol (Section 4.2) to enable read-only sharing of a software repository (VS2) among WAN users (VC3, VC4), and maintenance update by LAN administrator (VC5).

the client side to implement file system data caching is important to improving a repository's performance.

Scientific Data Processing: Scientific data are often collected/generated on-site, and processed and analyzed in an off-site computing center. Using a DFS to service data provisioning helps the analysis to be performed over different data ranges or with different granularities [25]. If temporal locality exists across consecutive runs of analysis, data caching can effectively hide network latency, and because of the producer-consumer model only reads are cached with support for an application acceptable consistency.

4. Cache Consistency Models

4.1. Overview

GVFS employs user-level proxy clients and servers to virtualize distributed file systems [12]. They are placed between native kernel NFS clients and servers to implement extensions and enhancements, including client-side disk caches for file attributes and data blocks [32]. A GVFS session is typically established by middleware through dynamic creation, configuration of a proxy server, and one or more proxy clients and mount points. Multiple sessions share the physical resources, yet each one can apply independent optimizations [33]. Figure 1 illustrates two GVFS sessions that are customized to support data provision for applications described in the motivating examples.

A consistency model specifies constraints on the order in which read and write operations appear to be performed in a distributed system. The choice of a consistency model in a DFS is an important and difficult one, because it has implications in the complexity of developing applications (and the DFS itself) and in the performance of applications. A relaxed model may be acceptable (and desirable) to a simulation application, but may fall short of supporting database applications that rely on locks. A complex consistency protocol may be desirable if it delivers high performance, but undesirable if it is difficult to implement, test and deploy in existing O/Ss or if it requires applications to use a consistency-aware API.

GVFS solves this dilemma by providing flexibly and efficiently customized cache consistency models to applications. Native NFS protocols (v2/v3) mainly rely on client-initiated revalidation requests to check for consistency. GVFS proxies, however, can be configured to support a variety of models, including the underlying NFS consistency itself, and alternative models. This section explains in detail the design and implementation of these application-tailored models.

4.2. Invalidation Polling Consistency

4.2.1. Protocol

This model employs invalidation buffers that reflect potential modifications to many files to reduce the rate at which per-file information is polled. Such an approach proves effective when modifications to the file system are infrequent and need to be quickly propagated to clients. The approach is illustrated in Figure 2: the proxy server of a GVFS session keeps track of logically time-stamped file handles that need to be invalidated in per-client buffers; the proxy clients use a new protocol message - GETINV - to request information related to the invalidation buffer.

Server-side: The proxy server adds file handles into invalidation buffers when it receives file modification requests from a client (e.g. CREATE, WRITE etc.). Multiple invalidations to the same file can be coalesced. Per-client invalidation buffers are of finite size and implemented as circular queues. The timestamps associated with each invalidation entry are generated by the server, and increase monotonically with incoming requests. A proxy client's GETINV request contains the timestamp of the last invalidation it has performed, and the proxy server returns the file handles stored in the buffer which the client needs to invalidate in its cache. The server can handle protocol cases where invalidation information is not fully available by using a flag (force-invalidate) to inform the client to invalidate its entire attributes cache. The proxy server processes a GETINV call as follows:

1) If this is the first GETINV call received from the client: initialize an invalidation buffer for the client, return updated timestamp and force-invalidation flag with value 1. Else,
2) If the timestamp in the GETINV request is earlier than the earliest one in the client's invalidation buffer: flush buffer and return updated timestamp and force-invalidation flag with value 1. Else,
3) Return buffer contents (and clear them), updated timestamp and force-invalidation flag with value 0. If buffer contents do not fit in a single RPC message, then return a poll-again flag with value 1 along with partial buffer contents.

Client-side: The proxy client polls the server with GETINV calls for potential invalidations occurred since its last known timestamp within a short time window. The polling time window can be fixed, or vary within a configurable range using an exponential back-off policy. The received invalidations are performed by invalidating the cached attributes of the concerned files, which will cause the proxy client to revalidate these files when they are accessed again. The proxy client processes the result from the GETINV call as follows:

1) Update a local variable holding the last known server timestamp.
2) If force-invalidation is equal to 1: invalidate its entire attributes cache. Else,
3) Scan the returned buffer and invalidate the attributes of the concerned files in its cache. And,
4) If poll-again is equal to 1: send another GETINV call to the server immediately.

In summary, only the file modifications observed by the proxy server cause invalidations on the proxy clients and they are transferred in a small number of GETINV replies. Only the files that are modified by other clients during the past polling time window need to be revalidated by a proxy client, but all the other per-file consistency checks issued from the kernel NFS client will be filtered out during the next time window.

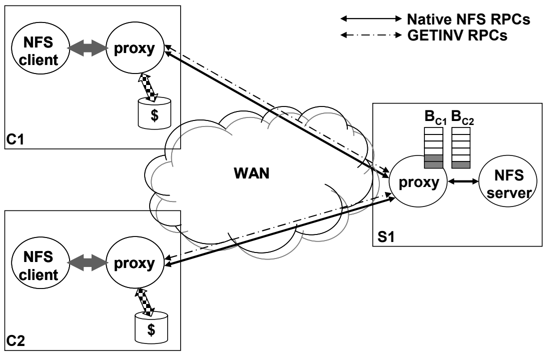

Figure 2: A GVFS session using the invalidation polling consistency protocol between the proxy clients at C1, C2 and the proxy server at S1. RPCs issued from kernel NFS clients can be served from the disk caches, while the proxy clients poll the proxy server for contents of per-client invalidation buffers (B_{C1}, B_{C2}) to maintain consistency.

4.2.2. Bootstrapping

The protocol uses logical timestamps to manage invalidation buffers. These are created by proxy server and used as arguments to GETINV calls by proxy client. The bootstrapping mechanism that provides an initial timestamp to a client uses a GETINV call with a null argument. Another form of bootstrapping takes place if the server fails or restarts and loses timestamp information. In this case, a client has a timestamp which is invalid, and must obtain a new, valid timestamp. The server handles this case by returning a new timestamp and a force-invalidation flag to each client's first GETINV after it restarts.

4.2.3. Failure Handling

The main factor that facilitates failure handling in this protocol is that the state stored on proxy server (invalidation buffers, timestamps) and clients (cached attributes and timestamps) is soft-state which can be safely discarded. If the server crashes, once recovered it can initialize new invalidation buffers from scratch, bootstrap clients with new timestamps as described above, and continue to serve the clients' GETINV calls. If a client crashes and loses its timestamp, then after recovery it issues GETINV with a null argument, and the server returns the latest timestamp with a force-invalidation flag. The same mechanism can be used if the client implements a policy to limit the number of invalidation handles it should process, effectively allowing the client to force a self-invalidation.

If a network partition happens, it is possible that the server's invalidation circular queue for the client has wrapped-around when it receives a GETINV from the client again. The server can detect this case by comparing the client's current timestamp with the earliest timestamp in its buffer. If the former one is earlier, it means that the client has not kept up with the invalidations, so the server should return the force-invalidation flag and an updated timestamp.

The invalidation mechanism is intended to provide relaxed consistency for the benefit of performance, but inconsistency can occur during the polling time window: a client may read a stale data block or file handle. It is appropriate for applications that can tolerate modest inconsistency (with the help from user or middleware). In practice, write sharing happens much less often than read sharing. Therefore this model is capable of providing applications with good performance and acceptable consistency. However, if stronger consistency is required, the delegation callback based model described below is better suited.

4.3. Delegation Callback Based Consistency

4.3.1. Delegation

A strong consistency model is achieved in GVFS via delegation and callback mechanisms. A delegation gives a client the guarantee to perform operations on the cached data without consistency compromises, while callback is used by the server to revoke the delegation in order to avoid potential conflicts. Delegation and callback decisions are made by the proxy server on per-file basis. A GVFS session can realize strong consistency by (1) disabling the kernel NFS client's attributes cache and (2) enabling the GVFS cache's delegation callback protocol. Two types of delegations are provided. Read delegation allows a client to read cached data without revalidation; the periodic consistency checks issued by kernel NFS client can be fully handled at client side. With a write delegation, the proxy client can further delay writes; both read and write requests to the file can be satisfied from the GVFS cache without contacting the server.

In the absence of open and close file operations in NFS (v2/v3), a proxy server speculates about these operations by tracking a client's data access. When a read or write request is received the corresponding file is considered "opened" by the client. In a read sharing scenario, multiple clients can have read delegations on the same file at the same time. But write delegation can be granted only if no other clients have the file opened.

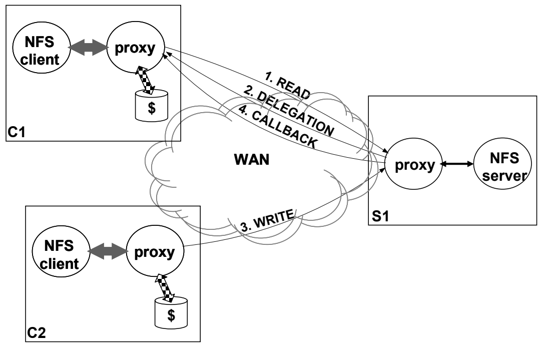

Figure 3: An example showing the sequence of interactions happened during a read delegation and its callback in a GVFS session which consists of two proxy clients and a proxy server.

When there is no sharing conflict a client obtains a delegation automatically with its first read/write request for the file. Otherwise, the conflicting request triggers the proxy server to recall the file's existing delegations and make it temporarily non-cacheable.

On the other hand, when a file has not been accessed by a client for a while, it is speculated closed by the client and the proxy server issues callback if this client has a delegation on the file. To allow a client to automatically renew a delegation, the proxy client periodically let a request for the file bypass the cache. The delegation's expiration and renew periods are both configurable per session, e.g. 10 minutes and 8 minutes respectively. The callback ensures the correctness of consistency even if the clocks of the server and client are badly skewed.

For the above mentioned proxy server-to-client interactions, the delegation and cacheability decisions are either piggybacked on the native NFS reply message, or enclosed in the GVFS callback calls. Figure 3 shows an example of these interactions.

4.3.2. Callback

A callback requires a server-to-client RPC call, which is inherently supported in GVFS because a proxy works as both RPC client and server. A proxy client encapsulates its listening port number along with its identification in regular RPC requests, so the proxy server knows how to connect an authenticated client for callbacks. To avoid deadlocks, the proxies are multithreaded to serve both NFS RPCs and GVFS callbacks. Correspondingly, independent queues are also maintained to buffer these two types of calls.

Callback of a read delegation invalidates the file's attributes in the proxy client's cache, which eventually causes revalidation of the file's cached data, while callback of a write delegation also forces the write back of cached dirty data. In a simple implementation, the callback does not return until all the data have been submitted to the server. However, the volume of dirty data can be very large and thus the callback as well as the other client's request which triggers this callback have to be blocked for a long time and may eventually time out. Note that this is still safe because both NFS and callback requests can be simply retried. But it is not desirable if the application which waits on the write back perceives a substantial response delay.

Since a request to a single block does not have to wait for the entire file being written back, the protocol is optimized as follows. If the number of cached dirty blocks is considerably large (e.g. more than 1k blocks), the proxy client returns a list of these blocks' offsets for the received callback. The block that is requested by the other client is immediately written back (if it is indeed dirty), but the other blocks are submitted afterwards. (To realize this, the requested block's offset is sent along with the file's handle in the callback.) Upon receiving the block list and the first block, the proxy server considers the write delegation revoked. However it needs to monitor the progress of the write-back and update the list accordingly until it completes. Meanwhile, requests from other clients to the blocks that are not written back will still generate callbacks to force the client to submit them promptly.

4.3.3. State Maintenance

The proxy server manages a GVFS session's state using a list for participating clients and a hash table for opened files. Client identification is provided by unique session keys encapsulated by proxy clients in every RPC request [32]. The client list stores their IDs and callback ports. Each opened file has an entry in the hash table to record its current state and sharers' IDs. A timestamp is also kept along with a client ID and updated every time the file is accessed by the client. Once the file is considered closed by a client, the client's information is removed from the file's entry; an entry is deleted from the hash table when the file is not opened by any client any more.

The expiration time determining whether a client has closed a file or not presents a tradeoff. Its value cannot be too small; otherwise delegations are given out so often that needs many callbacks to maintain the consistency. In contrast a long expiration time snags a client from getting delegations and possibly hurts its performance, and the proxy server also needs to track a large number of files and sharers. In the latter case, the proxy server can reduce the amount of state by proactively issuing callbacks on the least recently accessed files and then evicting their entries.

4.3.4. Failure Handling

Failure handling is more important to this consistency protocol than the invalidation-polling based model, because the state stored at the proxy server is crucial to strong consistency. But delegations also provide the proxy clients opportunities to continue serving application data requests even in presence of server crash or network partition. After the server comes back it can reconstruct the session's state by issuing special callbacks to all the known participating clients. To realize this, the client list data structure mentioned above is always stored directly in disk.

This type of callback is different because it targets at the entire cache rather than a specific file. The read delegation holders will invalidate all the cached attributes and thus require revalidation of every cached file when it is reaccessed. A proxy client that has write delegations will also reply the callback with a list of locally modified files so that the proxy server can rebuild the hash table. Note that before every client has answered the callback, the proxy server should block all the incoming requests. However, this grace period is considerably short because it only requires a single multicasted callback to the clients.

The nature of disk caching guarantees that a proxy client would not lose anything after it recovers from a crash and it can easily reconstruct the list of dirty blocks by scanning the entire cache once. However, it needs to contact the server to reconcile any conflicts happened during its crash. Therefore, it invalidates the entire attributes cache to force revalidation. And for the files that have dirty data cached, it tries to write back a single block for each file. This helps the client to reacquire the delegations if the files are not modified by others during the crash; otherwise, the cached dirty data are considered corrupted, and an NFS error will be reported when the application tries to use them.

5. Experimental Evaluation

The proposed approach is evaluated in this section by experiments on both microbenchmarks and application benchmarks. Microbenchmarks exercise GVFS with simple programs to demonstrate its performance compared to conventional DFSs, while application benchmarks use real scientific tools to investigate GVFS in typical Grid computing scenarios.

The emphasis of the experiments is in wide area environments, which are emulated using NIST Net [8]. Each link between the file system clients and server is configured with a typical wide-area RTT of 40ms and bandwidth of 4Mbps. Six file system clients and one file system server are set up on VMware-based VMs,

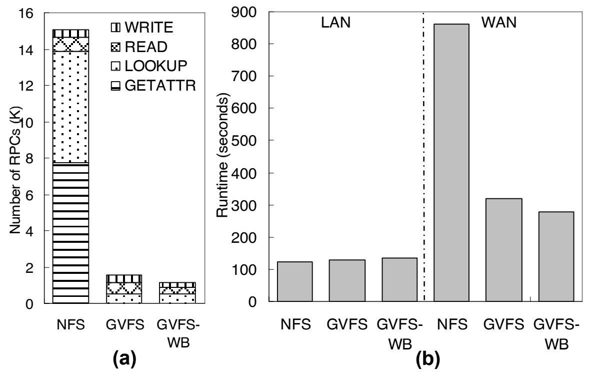

Figure 4: The number of RPCs transferred over the network (a) and the runtime (b) of the Make benchmark executed on NFS, GVFS with read-only caching (GVFS) and GVFS with write-back caching (GVFS-WB).

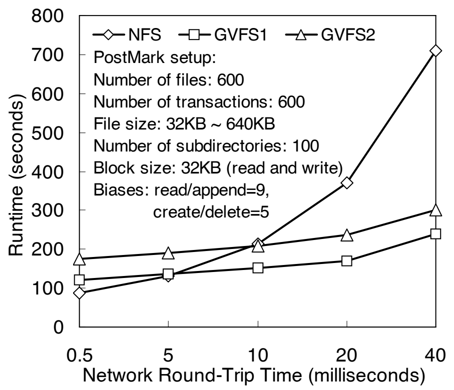

Figure 5: The runtime of PostMark in various network setups over NFS, GVFS with default kernel buffer setup (GVFS1) and GVFS with kernel attributes caching disabled (GVFS2).

which are hosted on two physical servers. Each physical server has dual 2.4GHz hyper-threaded Xeon processors and 1.5GB memory. Each VM is configured with 256MB memory and runs SUSE LINUX 9.2. The use of network emulator and VMs facilitates the quick deployment of a controllable, duplicable experimental setup. However, timekeeping within a VM is often inaccurate, so the system clock on a physical server is used to measure time, which suffices the granularity required by this evaluation.

The experiments are mainly conducted on file systems mounted through native NFS v3 and NFS v3 based GVFS both with ACL disabled. The file system server exports the file system with write delay and synchronous access. Every experiment is started with cold kernel buffer and GVFS disk caches by unmouting the file system and flushing the disk cache.

5.1. Microbenchmarks

5.1.1. Single Client Scenario

The first benchmark demonstrates the performance edge of GVFS caching in a single client scenario. It runs "make" on an application's source code (Tcl/Tk8.4.5), similar to the Andrew benchmark [18]. The make takes 357 C sources and 103 headers to generate 168 objects. It is executed on a file system mounted from the server via three different setups: native NFS (NFS), GVFS with read-only caching (GVFS) and GVFS with write-back caching (GVFS-WB). This benchmark mainly exercises the GETATTR, LOOKUP, READ and WRITE RPCs. The execution times and RPC counts are reported in Figure 4.

The data from executions on NFS show that, although the make only accesses hundreds of files, it generates tens of thousands of cache consistency checks (GETATTR calls) in the process of cross-referencing existing files to generate new objects. However, the results from GVFS prove that the disk cache can virtually satisfy all of them with its consistency model. When the client uses invalidation-polling with a typical period (e.g. 30 seconds), only tens of GETINV calls are required; with delegation callback based consistency there are no extra calls. The larger capacity of the disk cache also substantially reduces the number of LOOKUP calls, and the use of write-back further decreases the number of READs and WRITEs. Consequently, in WAN environment GVFS runs the benchmark three times faster than NFS.

The runtimes of the benchmark in a 100Mbps LAN is also measured to investigate the performance penalty for RPC interception and cache management in GVFS. The results show that the overhead is very small: only 4% with read-only caching and 8% when write-back is also used. As network latency grows GVFS caching's overhead should be overcome by the gain from saving network trips. This is confirmed by experiments on a file system benchmark, PostMark [15].

The benchmark is executed on NFS and GVFS in different network environments by varying the end-to-end latency. Two GVFS setups are used: one with the default kernel NFS buffer configuration (GVFS1), on which the invalidation-polling consistency model can be overlaid; the other with kernel attributes caching disabled (GVFS2), which gives GVFS the base to realize strong consistency with delegation and callback. Figure 5 shows that both GVFS setups outperform NFS after the RTT exceeds 10ms and they achieve more than 2-fold speedup when the latency is 40ms as in a typical WAN environment.

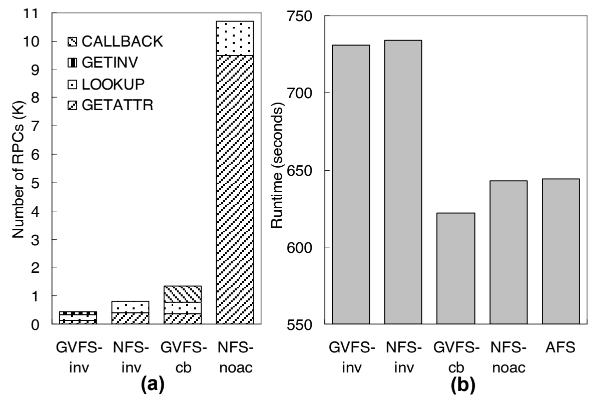

Figure 6: The number of RPCs transferred over the network (a) and the runtime (b) of the Lock benchmark executed across WAN with different setups: NFS with 30s revalidation period (NFS-inv), NFS with no attributes cache (NFS-noac), GVFS with 30s invalidation period (GVFS-inv), GVFS with delegation and callback (GVFS-cb), and AFS. The RPC counts used by AFS are not shown because it uses a different RPC protocol and not comparable.

5.1.2. Multiple Clients Scenario

This benchmark studies the behavior of GVFS' different consistency models when supporting cooperative, multiple-client workloads. It uses a popular mutual exclusion mechanism on file systems: file-based locks. In the experiment, six distributed clients compete for a lock by creating an independent temporary file and attempting to hard-link it to the shared lock file. If a client gets the lock, it pauses for a period of ten seconds and then releases the lock by unlinking the lock file. Otherwise, it pauses for a second and tries for the lock again. After a client releases the lock, it also pauses for a second and then rejoins the competition till it succeeds for ten times.

This experiment is conducted in WAN with different consistency models. It serves as a good example of the tradeoff between consistency and performance. When consistency is relaxed, a client may not see the release of lock immediately, and the previous owner of lock tends to get the lock again. On the other hand, stronger consistency provides better fairness among the clients but also consumes more bandwidth and generates higher server loads due to large number of consistency calls. To demonstrate the first case, the benchmark is executed on NFS and GVFS both with a revalidation/invalidation period of 30 seconds (NFS-inv and GVFS-inv). For the second case, the experiment is conducted with NFS with no attributes cache (NFS-noac) and GVFS with delegation and callback (GVFS-cb).

By analyzing the distribution of lock acquires from the experimental results, it is confirmed that fairness can be achieved with the strong consistency models but not with the weak ones. Further, Figure 6 shows that, in the latter case the benchmark takes nearly twice longer to execute, also because of the delay for a lock release being realized by other clients.

The overhead involved in achieving the same level of consistency is significantly different between NFS and GVFS. GVFS' client polling protocol uses 44% less consistency checks (GETATTR, GETINV) than NFS (GETATTR). In the stronger consistency case the difference between NFS and GVFS is even more dramatic. The consistency related calls issued by NFS (GETATTR) outnumbers that of GVFS (GETATTR, CALLBACK) by more than 10-fold. Hence substantial bandwidth and load are saved by using GVFS.

Note that although the benchmark runs faster on GVFS than on NFS, the advantage is not so large as in the number of RPCs. This is because most of the extra RPCs' latencies are overlapped with lock owners' pausing times during the execution.

As a reference, another traditional DFS that delivers strong consistency, AFS (OpenAFS 1.2.11), is also tested with the benchmark. The above experiments prove that GVFS can flexibly and efficiently provide different application-tailored consistency models, which is difficult to achieve with traditional DFSs.

5.2. Applications Benchmarks

5.2.1. Software Repository

The wide-area shared software repository scenario discussed in Section 3 is studied with NanoMOS, a 2-D n-MOSFET simulator. This is a compute-intensive application and benefits from parallel execution on Grid resources. A wide-area file system supports it by allowing the WAN users to read-share the application and its required software, including MATLAB with the MPI toolbox (MPITB), and also allowing the local administrator to maintain the repository at the same time. The experiment executes NanoMOS in parallel on six machines for eight iterations, while between the fourth and fifth run a software update happens. Two cases are considered: update to MPITB only, and to the entire MATLAB package. The repository is shared among these servers via native NFS, or GVFS with invalidation polling based consistency. The runtimes of the NanoMOS executions are shown in Figure 7.

NanoMOS' working dataset is relatively small (about 30MB per client), so both NFS and GVFS clients can cache it and reduce the runtime since the second run. But the difference is that the NFS client has to frequently check consistency for the cached data

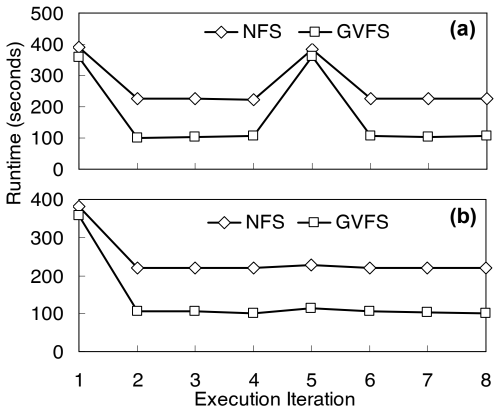

Figure 7: The runtime of parallel NanoMOS executions on six WAN resources. The software is shared via native NFS, or GVFS with 30s invalidation period. Between the 4th and 5th run an update happens on: (a) the entire MATLAB directory; (b) the MPITB directory only.

(about 2.7K GETATTRs per client per run), which can be almost eliminated by the GVFS client with its cache consistency. As result, GVFS delivers more than 2-fold speedup compared to NFS. When an update happens, the NFS client cannot know how many files are affected (the MATLAB package consists of 14K files/directories, while MPITB has only 540), so it has to always issue the same volume of consistency checks for the entire package. However, the GVFS client only uses invalidations proportional to the size of the update and batches them together in a few transactions (MATLAB update needs about 30 GETINV calls per client; MPITB update needs only two calls per client).

5.2.2. Scientific Data Processing

Another benchmark uses a coastal ocean hydrodynamics modeling application CH1D. It works in a scenario as discussed in Section 3, where real-time data are accumulated on coastal observation sites, and meanwhile processed on off-site computing centers. A wide-area file system helps the programs to share data naturally without explicitly transferring data back and forth. In the experiment, the data-producing program runs consecutively for 15 times, and each run gives the data-processing program 30 more input files. The data are shared between the programs via native NFS, or GVFS with delegation and callback.

Similar to the previous benchmark, the input dataset to the data-processing program is small and even 15 runs of data can still fit into its NFS client's kernel buffer. However, as the dataset grows the amount of consistency that the kernel client has to maintain also increases accordingly. The runtimes of the application (Figure 8) clearly demonstrate this trend: the overhead

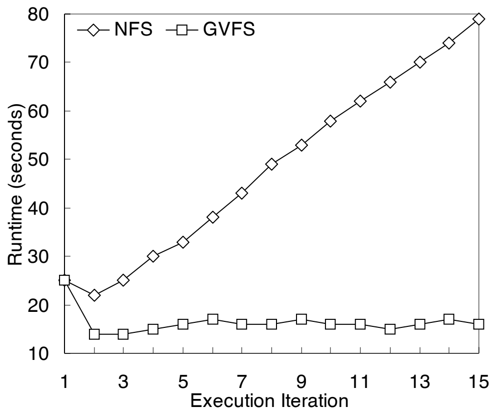

Figure 8: CH1D runtime. Data generation and processing are performed across WAN, where data are shared via native NFS, or GVFS with delegation callback based consistency. The data-processing program starts each run with 30 more input files from the data-producing program.

of cache consistency increases linearly as the size of the dataset. In contrast, with GVFS' consistency model this overhead is much smaller and remains almost constant for each run (only 30 callbacks). Accordingly, the performance speedup achieved by GVFS also grows as the dataset does and at the 15th run the benchmark runs already 5 times faster than on NFS.

Previous work has studied the above applications in different setups (e.g. single-client based NanoMOS execution). The experiments in this paper further prove the effectiveness and scalability of GVFS consistency models. In addition, previous results from comparison with NFS v2 show larger speedup than those compared with NFS v3 in this paper, because v3 does a better job than v2 by reducing the number of consistency checks [6]. But as demonstrated by the above experiments, it is still considerably less efficient than GVFS in wide area. Finally, it is also conceivable that a GVFS implementation overlaid on top of NFS v4 can take advantage of the available open/close operations and compound calls, and further provide good performance and consistency with the application-tailored models.

6. Conclusions

This paper shows that application-tailored cache consistency models and performance enhancements can be implemented in a manner that overlays existing DFS client/server implementations at the user level. A relaxed consistency model based on invalidation polling and a strong one based on delegation callback are evaluated in wide area environments. Microbenchmark and application based experiments show that large numbers of long-latency consistency-related calls can be avoided, and order-of-magnitude application-perceived speedups are achieved.

Solutions of this type are important in situations where applications designed in local area environments are scheduled by distributed/Grid computing middleware to execute in resources available across wide area networks. Through the use of virtualization, the DFS interface presented to applications is preserved and existing LAN implementations are leveraged, while WAN-specific features (encryption, identity mapping, latency-hiding [12][32]) are handled by middleware aware of application characteristics. In particular, WSRF-compliant data management services [33] can be used to realize the scheduling, customizing and isolating independent GVFS sessions with cache and consistency tailored to the data sharing patterns and application requirements.

7. Acknowledgement

This research is sponsored by NSF under grants EIA-0224442, EEC-0228390, ACI-0219925, ANI-0301108 and SCI-0438246. The authors also acknowledge a SUR grant from IBM, and Justin Davis and Peter Sheng for providing access to a benchmark application. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

8. References

[1] S. Adabala et al, “From Virtualized Resources to Virtual Computing Grids: The In-VIGO System”, *Future Generation Computing Systems*, Vol. 21/6, 2005.

[2] A. Alexandrov et al, “UFO: A Personal Global File System Based on User-level Extensions to the Operating System”, *ACM TOCS*, Vol. 16, pp. 207-233, 1998.

[3] P. Barham et al, “Xen and the Art of Virtualization”, *Proc. ACM SOSP*, October 2003.

[4] J. Bent et al, “Explicit Control in a Batch-Aware Distributed File System”, *Proc. 1st NSDI*, 2004.

[5] M. Blaze, “A Cryptographic File System for Unix”, *Proc. 1st ACM Conference on CCS*, pp. 9-16, 1993.

[6] B. Callaghan, NFS Illustrated, Addison-Wesley, 2002.

[7] B. Callaghan, T. Lyon, “The Automouner”, *Proceedings of the Winter 1989 USENIX Conference*, 1989, pp 43-51.

[8] M. Carson, D. Santay, “NIST Net: A Linux-based Network Emulation Tool”, *ACM SIGCOMM Computer Communication Review*, Vol. 33, No. 3, July 2003.

[9] B. Chun et al, “PlanetLab: An Overlay Testbed for Broad-Coverage Services”, *ACM SIGCOMM Computer Communications Review*, Vol. 33, No. 3, July 2003.

[10] J. Dike, “A User-mode Port of the Linux Kernel”, *Proc. 4th Annual Linux Showcase and Conference*, 2000.

[11] R. Figueiredo et al, “A Case for Grid Computing on Virtual Machines”, *Proc. 23rd ICDCS*, 2003.

[12] R. Figueiredo et al, “The PUNCH Virtual File System: Seamless Access to Decentralized Storage Services in a Computational Grid”, *Proc. 10th HPDC*, August 2001.

[13] I. Foster et al, “The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration”, *OGSI WG*, *GGF*, June 22, 2002.

[14] K. Fu et al, “Fast and Secure Distributed Read-Only File System”, *ACM TOS*, Vol. 20, No. 1, 2002.

[15] J. Katcher, “PostMark: A New File System Benehmark”, *Technical Report TR-3022*, Network Appliance, 1997.

[16] B. Krishnamurthy and C. E. Wills, “Study of Piggyback Cache Validation for Proxy Caches in the World Wide Web”, *Proc. USITS’97*, pp. 1-12, December 1997.

[17] T. Garfinkel, “Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools”, *Proc. Network and Distributed Systems Security Symp.*, 2003.

[18] J. Howard et al, “Scale and Performance in a Distributed File System”, *ACM TOCS*, Vol. 6, Issue 1, 1998.

[19] M. Kozuch, M. Satyanarayanan, “Internet Suspend/Resume”, *Proc. 4th WMCSA*, 2002.

[20] J. Kubiatowicz et al, “OceanStore: An Architecture for Global-Scale Persistent Storage”, *Proc. ASPLOS*, 2000.

[21] I. Krsul et al, “VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing”, *Proc. Supercomputing*, November 2004.

[22] M. Litzkow et al, “Condor: a Hunter of Idle Workstations”, *Proc. 8th ICDCS*, pp104-111, June 1988.

[23] R. Macklem, “Not Quite NFS, Soft Cache Consistency for NFS”, *Proc. USENIX Winter Technical Conf.*, 1994.

[24] D. Mazières, “A Toolkit for User-level File Systems”, *Proc. 2001 USENIX Technical Conference*, June 2001.

[25] J. Paladugula et al, “Support for Data-Intensive, Variable-Granularity Grid Applications via Distributed File System Virtualization - A Case Study of Light Scattering Spectroscopy”, *Proc. 1st CLADE*, June 2004.

[26] Y. Saito et al, “Taming Aggressive Replication in the Pangaea Wide-area File System”, *Proc. 5th OSDI*, 2002.

[27] C. Sapuntzakis et al, “Virtual Appliances for Deploying and Maintaining Software”, *Proc. 17th LISA*, 2003.

[28] J. Sugerman et al, “Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor”, *Proc. USENIX Annual Technical Conference*, 2001.

[29] V. Srinivasan and J. Mogul, “Spritely NFS: Experiements with Cache-Consistency Protocols”, *Proc. 12th SOSP*, December 3-6, 1989, pages 45-57.

[30] SunSoft, “Cache File System (CacheFS)”, *White-Paper*, Sun Microsystems, Incorporated, February 1994.

[31] B. White et al, “LegionFS: A Secure and Scalable File System Supporting Cross-Domain High-Performance Applications”, *Proc. Supercomputing Conference*, 2001.

[32] M. Zhao, J. Zhang and R. J. Figueiredo, “Distributed File System Virtualization Techniques Supporting On-Demand Virtual Machine Environments for Grid Computing”, *Cluster Computing*, Vol. 9, Issue 1, 2006.

[33] M. Zhao, V. Chadha and R. J. Figueiredo, “Supporting Application-Tailored Grid File System Sessions with WSRF-Based Services”, *Proc. 14th HPDC*, July 2005.