Proxy Managed Client-Side Disk Caching for the Virtual File System

Technical Report TR-ACIS-04-001

November, 2004

Advanced Computing and Information Systems Laboratory (ACIS)

Electrical and Computer Engineering – University of Florida

Gainesville, FL, USA 32611-6200

Ming Zhao, Renato J. Figueiredo

Electrical and Computer Engineering

University of Florida, Gainesville, Florida

Email: ming@acis.ufl.edu, renato@acis.ufl.edu

Abstract

This report discusses the proxy managed client-side disk caching for the Virtual File System (VFS). Disk caching is important for Grid data provisioning to hide the high network latency in wide-area environments. The mechanism described in this report is a novel approach which extends a distributed file system virtualization to provide efficient and seamless data access across Grid resources. It is unique in that, 1) it is dynamically created and managed by middleware-controlled proxy per VFS session, 2) it can be customized with per-application based configurations, 3) it supports write-back caching and hierarchical disk caching, and 4) it is implemented at user-level and thus can be seamlessly integrated with Grid applications and resources via the user-level VFS. The performance evaluation of VFS with the disk caching is also reported. Results show that the availability of proxy disk caches, deployed on either the compute server's local disk or its LAN cache server, improves VFS' performance by more than 300% for a benchmark (SPECseis) which consists of mixed computing- and I/O- intensive phases, and nearly 200% for a benchmark (LaTeX) which involves iterations of interactive modifications and compilations of documents.

1. Introduction

The fundamental goal of Grid computing [1] is to efficiently and seamlessly multiplex distributed resources of providers among users across wide area networks. To realize this vision, a key challenge is the efficient and seamless provisioning of data to Grid applications. From an application's point of view, it desires transparent data access across heterogeneous resources and convenient deployment on Grid without too much overhead to become "Grid-aware". Meanwhile, it expects the data access to occur with good performance even though the resources are distributed across wide area environments.

These two goals are difficult to achieve at the same time with current existing data provision approaches. However, a novel solution, which is capable of providing both efficient and seamless data access for Grid applications, can be constructed based on extensions to a proxy-based distributed file system virtualization [2]. This report discusses the design and implementation of one of the important extensions, the proxy managed client-side disk caching. It also reports on its performance for typical applications in various scenarios.

In the absence of a global file system, a number of middleware-based approaches have been studied for providing unified data management over heterogeneous services and protocols in Grid. Typically they can be classified into the following categories:

Custom application programming interfaces can be used to take advantage of data transfer methods provided by Grid middleware [3][4]. But applications have to be rewritten, and sometimes their source codes are just not available. Besides, a customized interface is normally only a modified subset of the file system abstraction presented by typical operating systems.

User-level libraries can be either statically or dynamically linked to applications to realize remote data access. However, statically re-linking requires applications being able to rebuilt [5], while dynamic linking does not work for statically linked applications [6]. And, normally they do not support a full file system abstraction either.

Custom remote file servers extend upon existing distributed file systems to achieve new functionalities and improvements [7]. NFS [8], the de facto distributed file system, is often chosen for leverage. But modifications to file servers are normally at kernel level, which are difficult to be widely deployed on Grid.

System call interception agents trace and modify file I/O system calls issued by applications for remote data access [9][10]. However, they require low-level process tracing capabilities that are complex to implement and highly O/S dependent, and cannot support non-POSIX compliant operations (e.g. setuid).

Besides the above categories, another sort of approaches which is similar to custom remote file servers has also been studied. But instead of modifying kernels, the customization is realized by user-level processes, namely *distributed file system proxies*, which intercept communications between unmodified file system clients and remote servers, and thus extend the original distributed file systems to fit the needs of Grid data provisioning. Unlike the other schemes, this sort of approaches does not require any modifications to applications or O/Ss.

Previous work has shown that such an approach can be achieved by a NFS-based virtualization layer, the Virtual File System (VFS) [2]. It is constructed by a middleware-controlled user-level proxy that forwards requests between unmodified NFS clients and servers. NFS is the most widely used distributed file system, which can be easily found on any Grid machines. VFS leverages existing O/S support for NFS, and is completely implemented at user-level, so it can be conveniently deployed on Grid resources. It also preserves an unmodified file system abstraction that matches what is provided by typical LAN setups, and thus can be seamless integrated with applications and provide transparent remote data access. Furthermore, it employs the concepts of logical user accounts [11] to decouple users from and scale applications across administrative domains.

The described VFS is a unique approach to provide on demand, user-transparent access to data for unmodified applications through native NFS clients/servers widely available in existing O/Ss. However, in order to support efficient data access, VFS must address the performance problem of NFS transactions in wide area environment, specifically, the high latency of remote data access. This report attacks this problem by employment of user-level disk caches via extensions to VFS. Disk caching is not typical in existing NFS implementations, but it is important because of its high-capacity and persistency. It can complement kernel-level memory buffer to form an effective cache hierarchy.

There are some distributed file system which also employ disk caching [12], however, VFS disk caching is unique in that: 1) it is dynamically created and managed by middleware-controlled proxy on per-session basis, 2) it can be customized with per-user and per-application based caching policies, 3) it supports write-back policy, cache sharing among file system sessions, and multiple level of disk cache hierarchy, and 4) it is implemented at user-level and thus can be seamlessly integrated with Grid resources via VFS. The contributions of this report are novel application-transparent techniques that scale VFS to multi-proxy file system and implement user-level disk caches managed by client-side proxy. The report also reports on the performance results collected from executing typical applications in different network setups.

The rest of the paper is summarized as follows. Section 2 explains the designs of the multi-proxy VFS and proxy managed client-side disk caching. Section 3 describes in details the implementation of proxy caching for

various NFS procedures. The results from performance evaluation of VFS with proxy caching are discussed in Section 4. Finally, section 5 examined related work and section 6 concludes the report.

2. Architecture

2.1 Previous Work

The concept of *logical user accounts* [11] is proposed to decouple users from underlying hardware and administrative domains. A logical user account is made up of two components, a *shadow account* and a *file account*. Users are allocated shadow accounts on compute servers to perform computing, while user data are stored under file accounts on file servers. Users' access to data is brokered by the VFS proxies, which dynamically multiplex shadow accounts among file accounts and provide on-demand data transfer during the computing sessions.

This approach virtualizes access to Grid data by intercepting, modifying and forwarding remote procedure calls (RPCs) of a de-facto distributed file system - NFS. It decouples data provision from applications and Grid data management: VFS is responsible for implementing the specific aspects of application-desired Grid data access (e.g. cross-domain, on-demand, security, and performance etc.), while applications can use generic O/S file I/O for data operations, and unmodified kernel file system clients and servers are responsible for low-level data handling and interfacing with O/S system calls.

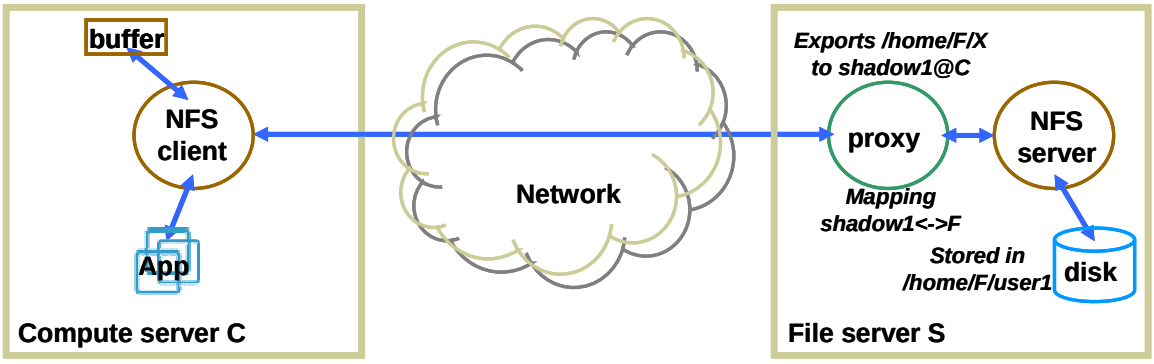

Figure 1: A VFS session established by a single server-side proxy. The shadow account *shadow1* is allocated to u*ser1* to run applications on the compute server *C*, and the user's data are stored under the file account *F* on the file server *S* (*/home/F/user1*). During the VFS session, the proxy exports the file directory to *shadow1@C*, authenticates and forwards requests from the native NFS client on *C* to the native NFS server on *S*, and maps the user identities between *shadow1* and *F* inside each RPC message. At the client side, only kernel buffer caches can leverage temporal locality of data accesses.

VFS leverages middleware support to dynamically create and manage logical user accounts and file system sessions on Grid resources. During each VFS session, a proxy is deployed at the file server side to broker the data access from the shadow account to the file account (Figure 1). Previous performance analyses have shown that in a local-area setup the application-perceived overhead introduced by the proxy is small [13], but in a wide-area setup the overhead is much larger, especially for an I/O intensive application.

2.2 Multi-Proxy VFS

The design of the VFS proxy allows it to behave both as a server (receiving RPC calls) and a client (issuing forwarded RPC calls, possibly with modified arguments), which supports connections of proxies "in series" between a native O/S client and server. While a multi-proxy may introduces more overhead from processing and forwarding RPC calls, it allows extensions to VFS which may support more functionalities and improve performance.

Extensions to the protocol can be implemented between proxies, again, without modifications to native O/S clients, servers or applications. For example, a Virtual Private Grid File System (VP/GFS) can be realized by inter-proxy session-key authentication and encrypted tunneling. Another possible functionality extension is inter-proxy cooperation for fine-grained cache consistency models. To improve performance, VFS proxy can be deployed at the client-side to dynamically create and manage disk caching of file system data, which will be

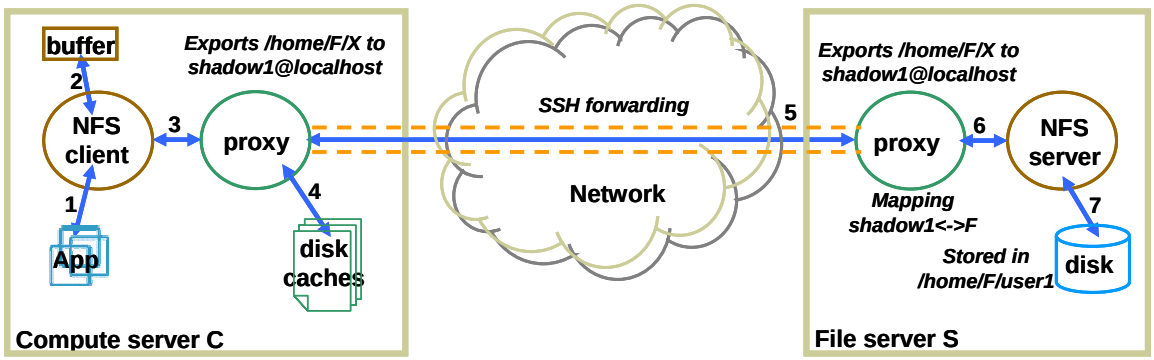

Figure 2: A multi-proxy VFS session. Besides the server-side proxy as in typical VFS setups, the client-side proxy manages the disk caches dynamically. At the compute server, the application issues system calls that are processed by the kernel NFS client (1). Requests may hit in the kernel-level memory buffer cache (2); those that miss are processed by the user-level proxy (3). At the client-side proxy, requests that hit in the disk cache are satisfied locally (3); proxy misses are forwarded as SSH-tunneled RPC calls to the server-side proxy (5), which fetches data through the kernel NFS server from disk (6)(7).

explored in details in the following sections. Other possible extensions include inter-proxy high-speed data channels to improve transfer of large files.

Figure 2 illustrates a multi-proxy VFS setup, where two proxies work between the native NFS server and client cooperatively. The server-side proxy deployed on the file server *S* authenticates and forwards data access from the shadow account *shadow1* on the compute server *C* to a user file directory */home/F/user1* under the file account *F* on *S*, and maps between the credentials of *shadow1* and *F* inside each RPC message. If the requests come from the client-side proxy directly, the server-side proxy should export the directory to *shadow1@C*. But in this example, the connections are forwarded via SSH tunneling, so the directory is exported to *shadow1@localhost*. On the other hand, the client-side proxy on *C* authenticates and forwards data access from *shadow1@localhost* to the server-side proxy, and more importantly, it creates and manages disk caches dynamically to complement the kernel buffer caches.

2.3 Proxy Managed Client-Side Disk Caching

Caching is a classic, successful technique to improve the performance of computer systems by exploiting temporal and spatial locality of references and providing high-bandwidth, low-latency access to cached data. The NFS protocol allows the results of various NFS requests to be cached by the NFS client [14]. However, although memory caching is generally implemented by NFS clients, disk caching is not typical. Disk caching is especially important in the context of a wide-area distributed file system, because the overhead of a network transaction is high compared to that of a local I/O access. The large storage capacity of disks implies great reduction on capacity and conflict misses [15]. Hence complementing the memory file system buffer with a disk cache can form an effective cache hierarchy: memory is used as a small but fast first level cache, while disk works as a relatively slower but much greater second level cache.

Disk caching in VFS is implemented by the file system proxy. A VFS can be established by a chain of proxies, where the native O/S client-side proxy can establish and manage disk caches, as illustrated in Figure 2. VFS disk caching operates at the granularity of NFS RPC calls. The cache is structured in a way similar to traditional block-based hardware designs: the disk cache contains file banks that hold frames in which data blocks and cache tags can be stored. Cache banks are created on the local disk by the proxy on demand. The indexing of banks and frames is based on a hash of the requested NFS file-handle and offset and allows for associative lookups. The hashing function is designed to exploit spatial locality by mapping consecutive blocks of a file into consecutive sets of a cache bank.

As VFS sessions are dynamically setup by middleware, disk caches are also dynamically created and managed by proxies on per-session basis. When a VFS session starts, the client-side proxy initializes the cache with middleware-configured parameters, including cache path, size, associativity and policies etc. During the session, some of the parameters, especially caching policies, can also be reconfigured. When the session finishes, proxy can clean up the cache with different middleware-controlled policies too, possibly, submitting dirty data, and flushing or keeping entire cache contents etc.

There are several distributed file systems that exploit the advantages of disk caching too, for example, AFS [12] transfers and caches entire files in the client disk, and CacheFS supports disk-based caching of NFS blocks. However, these designs require kernel support, and are not able to employ per-user or per-application caching policies. In contrast, VFS is unique to support customization of the user-level disk caching on a per-user/application basis. For instance, cache size and write policy can be configured by user requirements/priorities, or optimized according to the knowledge of an application. A more concrete example is enabling file-based disk caching by meta-data handling and application-tailored knowledge to support heterogeneous disk caching [16].

VFS proxy caching supports different policies for write operations: read-only, write-through and write-back, which can be configured by middleware for specific user and application per file system session. Write-back caching is an important feature in wide-area environments to hide long write latencies. Typically, kernel-level NFS clients are geared towards a local-area environment and implement a write policy with support for staging writes for a limited time in kernel memory buffers. Kernel extensions to support more aggressive solutions, such as long-term, high-capacity write-back buffers are unlikely to be undertaken; NFS clients are not aware of the existence of other potential sharing clients, thus maintaining consistency in this scenario is difficult.

The write-back proxy caching described in this report leverages middleware support to implement a session-based consistency model from a higher abstraction layer: it supports O/S signals for middleware-controlled writing back and flushing of cache contents. This model of middleware-driven consistency is assumed in this report; it is sufficient to support many Grid applications, e.g. when tasks are known to be independent by a scheduler for high-throughput computing. It is also possible to achieve fine-grained cache coherence and consistency models by implementing call back and other inter-proxy coordination, which are subjects of on-going investigations.

Furthermore, while caches of different proxies are normally independently configured and managed, it also allows for them to share read-only cached data for improving cache utilization and hit rates. On the other hand, a series of proxies, with independent caches of different capacities, can be cascaded between client and server, supporting scalability to a multi-level cache hierarchy. For example, a two-level hierarchy with GBytes of

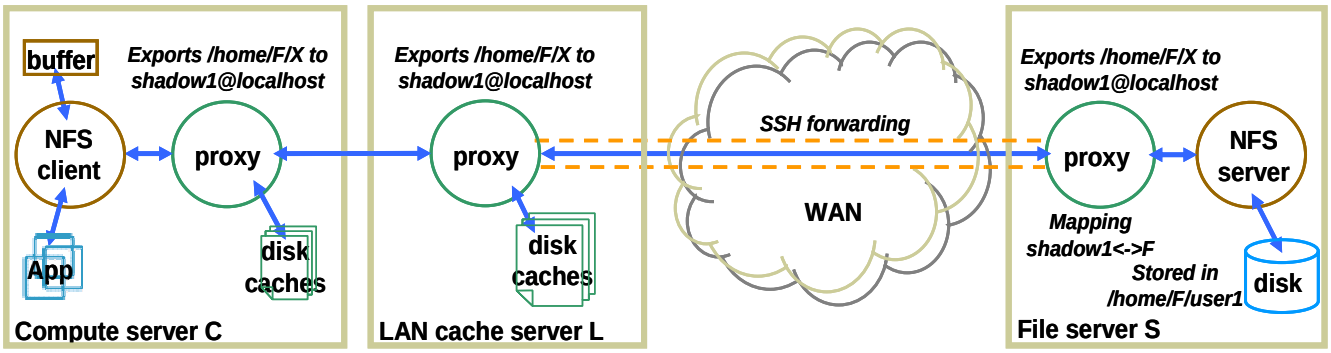

Figure 3: A two-level proxy disk caching hierarchy for VFS. The middle-tier proxy manages a disk cache with larger capacity than the disk cache on the compute server's local disk, in order to leverage temporal locality of data accesses from clients in the same LAN. The proxy also exports the user's file directory *home/F/user1* to the shadow account *shadow1@C*, and it authenticates and forwards the requests from the client-side proxy to the server-side proxy.

capacity in a node's local disk to exploit locality of data accesses from the node, and TBytes of capacity available from a LAN disk array server to exploit locality of data accesses from nodes in the LAN (Figure 3).

The proxy cache can be deployed in systems which do not have native kernel support for disk caching, e.g. Linux. It is worth to emphasize that VFS along with the proxy disk caching is completely implemented at user-level, which enables seamless integration with native O/S clients, servers and applications.

3. Implementation

In this section, the implementation of the proxy managed client-side disk caching is explained in details for various NFS procedures. The current prototype is constructed upon NFS V2, so the following discussions are also based on this version of the NFS protocol.

3.1 GETATTR Procedure

GETATTR retrieves the attributes for a specified file or directory. The procedure takes the file handle of an object as the argument, and returns the attributes for the object. Most NFS protocol client implementations caches the results of GETATTR calls to reduce over-the-wire requests to server, but still check periodically whether cached objects have changed on server or not. So if a client is using a large memory cache the GETATT call may be the most common NFS call from the client.

The first GETATTR call to a file system object is forwarded to the server, and the results are stored in a cache block. Then the following GETATTR calls to this object will get attributes from cache as long as the corresponding cache block is still valid. In certain conditions, for example, the object is removed, or the attributes are changed etc., the cached attributes should be invalidated or updated. In a single client scenario, there is no need to check server against cached objects. However, if multiple clients are sharing files, an appropriate consistency model should be carefully selected. It is possible to implement server call back functionality, where server-side proxy can invalidate or update the cache managed by client-side proxy when the cached objects are modified by other clients. Or, a client-polling based validation or invalidation mechanism can be used to update the cached attributes periodically.

3.2 SETATTR Procedure

SETATTR changes one or more of the attributes of a file system object on server. The new attributes and the target's file handle are specified in arguments. The attributes argument contains fields which are either -1, which indicates the field should be ignored, or the new value, which is used to reset the field. Only protection mode, user id, group id, file size, last access time and last modification time of the target object can be set from client.

In the implementation of proxy attributes cache, write-through with write-allocate policy is provided for SETATTR requests. If the attributes of target object are not cached, they will be fetched from server and cached on disk. Then proxy updates the cache and also forwards the request to server. But there are several special situations which should be taken care of, when client tries to set the file size. Firstly, if the client issues a SETATTR request to set the file size to a value less than the current file size, then the file should be truncated. Proxy should invalidate any cached write operations to that file which begin from an offset beyond the new file size. Secondly, if client sets the file size to a value greater than the current file size, it will cause logically zeroed data bytes to be added to the end of the file. Some servers may refuse to increase the size of a file through SETATTR procedure, but the client can work around this by writing a byte of data at the maximum offset. No matter how it is implemented in NFS client, the client-side proxy should consider updating the cached data blocks of that file, as described in Section 3.5.

3.3 LOOKUP Procedure

Procedure LOOKUP searches a directory for a specific name and returns the file handle as well as the attributes for the corresponding object. Normally before a client can manipulate an object, it should retrieve its file handle from server by LOOKUP request. The file attributes returned by LOOKUP procedure give the client an initial

idea of the object, and thereafter the client basically uses GETATTR procedure to get the file attributes. Both the returned file handle and attributes are cached by the proxy.

Almost every NFS procedure needs the file handles for target objects as arguments, but NFS client doesn't have to issue a LOOKUP request before each procedure, because the result of a LOOKUP procedure is usually cached in memory and reused during the life span of the object, unless the cached file handle is evicted due to lack of cache space. Proxy disk cache provides a second level disk cache which is usually much greater than memory cache, thus the eviction of a cached file handle from disk cache is much less likely to happen. Furthermore, the penalty of a memory cache miss is greatly reduced if the file handle is cached in disk, since the local disk I/O latency is generally much smaller than the latency of accessing a remote NFS server.

3.4 READ Procedure

Client issues a READ request to read the data from a given offset in a specified file. The file handle of the target object, the offset of where the read begins and the count of requested bytes of data are the arguments for READ procedure. Server returns the data and file attributes of the object. If the requested offset is greater than or equal to the actual file size, the returned data have zero byte length. If the offset plus the requested count is greater than the file size, the server returns a short read and the client will assume that the last byte of data is the end of file.

When proxy receives a READ request, it looks up cache according to the file handle and the offset. If the requested data is already cached, the proxy will return the data and the file attributes from cache without asking server. Otherwise, they are fetched from server, stored in cache and returned to client. Data cache is organized into cache banks, while each bank file contains certain number of cache blocks. It is very important to choose an appropriate hash function so that consecutive data blocks of a file can be stored in consecutive cache blocks of a cache bank, and spatial locality of read requests can be efficiently exploited. Otherwise, for sequential reads to consecutive data blocks of a file, multiple cache bank files have to be opened and sought, which may introduce substantial overhead.

Besides, a read request may ask for data less than the size of a NFS block, but proxy still fetches the entire block from server, in which sense a moderate read ahead is used. Sometimes, requested data in a read call may cross several blocks, and then proxy will fetch these blocks from cache or/and server and construct results for the client.

3.5 WRITE Procedure

Client issues a WRITE request to write the given data beginning from a specified offset in the target object. So the arguments of WRITE procedure contain the target's file handle, the beginning offset, and the actual data to be written. Server returns the file attributes after successfully performing the requested write. The side effect of a WRITE request is the update of the modification time in the attributes. However, the modification time of the file should not be changed unless the contents of the file are actually changed.

VFS disk caching can support both write-through and write-back with write-allocate policies for write operations, and middleware can control proxy to switch caches between these modes. If write-back caching is enabled, when proxy receives a WRITE request, it looks up cache according to the file handle and the offset. Upon cache hit, proxy will update the cache locally. Upon cache miss, the requested data block will be fetched from server, and updated and stored in cache. Each data cache block has been tagged by a state field: "invalid" state means the block is empty; "clean" state means the block stores data which are consistent with what on the server; "dirty" state means the block contains data which have not been written back to server yet. No matter the cached block is clean or dirty, the following READ or WRITE requests to the block will be satisfied locally.

In the current implementation, under two conditions a dirty cache block should be written back to server: firstly, the proxy receives a specific signal to submit all dirty blocks to server; secondly, the dirty block is evicted from cache due to lack of cache space. In the first case, to avoid going through every data block of the entire cache to look for dirty blocks, locations of all dirty blocks are recorded in an additional data structure. Then at submitting time, proxy is able to check only the dirty blocks and write them back if necessary. This record of dirty block information is persistently stored in disk, so even after the client crashes the locally cached writes can still be correctly submitted to server after it recovers.

The second condition that a dirty cache block should be written back is when there are no more clean blocks in the corresponding cache bank. The cache replacement policy favors empty or clean blocks over dirty ones. Basically, when a new data block needs to be inserted into cache, it first tries to use an invalid block. If there are no more invalid blocks, it then tries to randomly choose a clean block to evict and replace with the new block. Finally, if all blocks in the cache bank are dirty, it randomly choose one from them, write it back to server, and then replace it with the new data block.

NFS client may issue a write with data which cross the boundary of cache blocks, in which case, proxy fetches the blocks from server if necessary, has these blocks updated in cache, and then constructs the results for the client. Sometimes the data to be written begin from an offset greater than the current size of the file. When

caching such a write, the gap between the end of the current file and the offset where the write begins will be filled up with zeros.

3.6 CREATE and MKDIR Procedures

NFS client uses CREATE procedure to create a regular file, and uses MKDIR procedure to create a directory. The arguments include the file handle of the directory in which the new object is to be created, and the name and initial attributes that are to be assigned to the new object. Once the object is created, server returns its file handle and attributes to the client.

In current cache implementation, when proxy receives a CREATE/MKDIR request, it forwards the request to server and caches the returned file handle and attributes. In NFS the CREATE/MKDIR procedure does not support “exclusive create”, which means that the target object to be created may already exist, and the new object will overwrite the old one. So upon a CREATE/MKDIR request, proxy first invalidates the possibly cached old file handle and attributes and then inserts the new file handle and attributes into cache.

3.7 LINK and SYMLINK Procedure

LINK procedure creates a file with a given name in a specified directory, which is a hard link to an existing file. A hard link should have the property that changes to any hard-linked files are reflected in all linked files. When a hard link is made to a file, the link number in the file attributes should become one greater. From the perspective of NFS, hard-linked files may have different path and names, but their file handle is the same one.

SYMLINK procedure creates a symbolic link with a given name in a specified directory, which is a reference to a given existing file. The content of a symbolic link is the pathname to the file which the link refers to. The client can use READLINK procedure to read the pathname stored in a symbolic link. From the perspective of NFS, a symbolic link has a different file handle than the object it links to.

Only LINK has the effect of changing file attributes. So after forwarding the request to server, proxy will update the corresponding cached file attributes.

3.8 REMOVE, RENAME and RMDIR Procedures

Procedure REMOVE removes an entry from a directory. If the entry is the last reference to the corresponding file system object, the object may be destroyed and its file handle becomes stale. Otherwise, the number of links for the object is reduced by one, but the REMOVE request has no effect on the file data and the file handle

because the file is still valid. Procedure RENAME renames a directory entry identified by "*fromname*" in the directory "*fromdir*" to "*toname*" in directory "*todir*". If the directory "*todir*" already contains an entry with the name "*toname*", the existing target is removed before the rename occurs. The file handle of the target may or may not become stale, depending on whether the number of links to the object is or is not zero after being reduced by one. Procedure RMDIR removes a subdirectory from a directory. In the arguments, the subdirectory to be removed is specified by the name, and the directory from which the subdirectory is to be removed is given by the file handle.

If the file handle of the target object becomes stale after a REMOVE or RENAME call, the cached writes for this object should not be written back to server in the future. There are two different approaches to achieve this. One solution is that once a file handle becomes stale, proxy goes through the entire data cache and invalidates all blocks associated with this file handle. Another solution is that proxy only records the stale file handle in a list, but at submitting time it does not write back the dirty blocks which belong to a stale object. Compared to the first solution, the second one can save a lot of time, but it does waste some space because stale data may continue occupying cache blocks before the submitting. There is a trade off here, but time efficiency, which is more crucial to a disk cache, is preferred. Furthermore, because submit of dirty data often happens when a VFS session is idle or ended, the second approach can also achieve much shorter response time that the first one during an active session. In the implementation, the second solution is also improved as the following: when proxy tries to find an empty cache block to insert the new data, it also checks whether the cached data are from a stale object or not. If it is, then the cache block will be used as an empty one. In this way, the cache blocks occupied by stale objects will eventually be recycled.

4. Performance Evalutions

4.1 Experimental Setup

The proxy disk caching discussed in this report has been implemented as an extension to the open-source NFS proxy [2]. This section analyzes the performance of the enhanced VFS using a prototype deployment based on the extended file system proxy implementation.

The experiments are based on the execution of applications deployed in both local-area and wide-area network environments. The compute server is a single processor, 1.1GHz Pentium-III cluster node with 1GB of main memory, 18GB of SCSI disk and 100Mbit fast Ethernet network interface. It runs Linux Red Hat 7.3 with kernel 2.4. The proxy cache is configured with 512 file banks which are 16-way associative. It has a capacity of 8 GBytes but during the experiments normally only 1~3 GBytes are actually consumed.

The LAN file server is a dual-processor 1.8GHz Pentium III cluster data node, with 1GB of RAM and 576GB of disk storage (8x72GB SCSI disks, RAID5). It is connected to the compute server via a switched 100Mbit/s Ethernet network. The WAN file server is a dual-processor 1GHz Pentium-III cluster node with 1GB RAM and 45GB disk. It is located at Northwestern University, behind a firewall that only allows SSH traffic through, and connected to the compute server through Abilene.

During the local-area or wide-area experiments, SSH tunneling of VFS connections is always employed by file system proxies to achieve data privacy and also to allow connections to pass firewalls [17]. Every experiment begins with cold caches, including kernel buffer cache and possibly proxy disk caches, by un-mounting VFS and remount it, and flushing the proxy caches if disk caching is enabled. The results reported in the following sections are all averaged from multiple runs of the experiments. The proxy cache prototype is currently only implemented upon NFS version 2, which limits the maximum size of an on-the-wire NFS read or write operation to 8KB. Thus NFS version 2 with 8KB block size is leveraged by VFS when the proxy caching is used, but NFS-V3 with 32KB block size is used in all other scenarios.

4.2 Benchmark Applications

The following applications are selected as benchmarks to investigate the performance of VFS:

SPECseis is taken from the SPEC high-performance benchmark suite and implements processing algorithms used by seismologists to locate resources of oil. The sequential version of the benchmark is used, with the small dataset and four execution phases: generation of source/receiver geometry and seismic data (1), data stacking (2), time migration (3) and depth migration (4). The benchmark has been compiled with OmniCC 1.4 (front-end) and gcc 2.96 (back-end). It models a scientific application that has an I/O intensive component (phase 1, where a set of files is generated, including a large temporary file used as an input for the subsequent phases) and a compute-intensive part (phase 4). The benchmark binaries, input/output and configuration files are all stored in a directory mounted via VFS.

LaTex benchmark is designed to model an interactive document processing session. It is based on the generation of a Portable Document File (PDF) version of a 190-page document (a Ph.D. thesis) edited with LaTeX. The source document has 14 LaTeX files, 17 BibTeX files, and 52 encapsulated PostScript images, using 14 Mbytes of storage. The benchmark runs the “latex”, “bibtex” and “dvipdf” programs in sequence (a total of 20 iterations). Between iterations, a different version of one of the LaTeX input files is generated with diff/patch commands, modeling an environment where a user interactively modifies and compiles a document. The source document and the output files are stored in a directory mounted via VFS.

For comparison, the execution times of the above benchmarks are measured in five different scenarios:

Local: The working directory is stored in the compute server's local disk file system;

LAN: The working directory is VFS-mounted from the LAN file server;

WAN/NC: The working directory is VFS-mounted from the WAN file server without the proxy disk caching;

WAN/C: The working directory is VFS-mounted from the WAN file server with the proxy disk caching enabled on the compute server;

WAN/2C: The working directory is VFS-mounted from the WAN server with the proxy disk caching enabled on the compute server's LAN cache server.

The first two scenairos are used as performance references, where the three different scenarios in WAN are designed to investigate the effectiveness and efficiency of VFS with proxy disk caching on Grid resources.

4.3 Results and Analyses

Figure 4 shows the execution times for the four phases of the SPECseis benchmark in the various scenarios. As expected, the performance of the compute-intensive part (phase 4) is very close across all scenarios. The results of the I/O intensive part (phase 1), however, differentiates very much. The overhead caused by remote data access in the WAN/NC scenario is more than 14 folds compared to the Local scenario. But the existence of proxy disk caching with write-back policy reduces this overhead significantly to about 20% in the WAN/C scenario and to about 30% in the WAN/2C scenario.

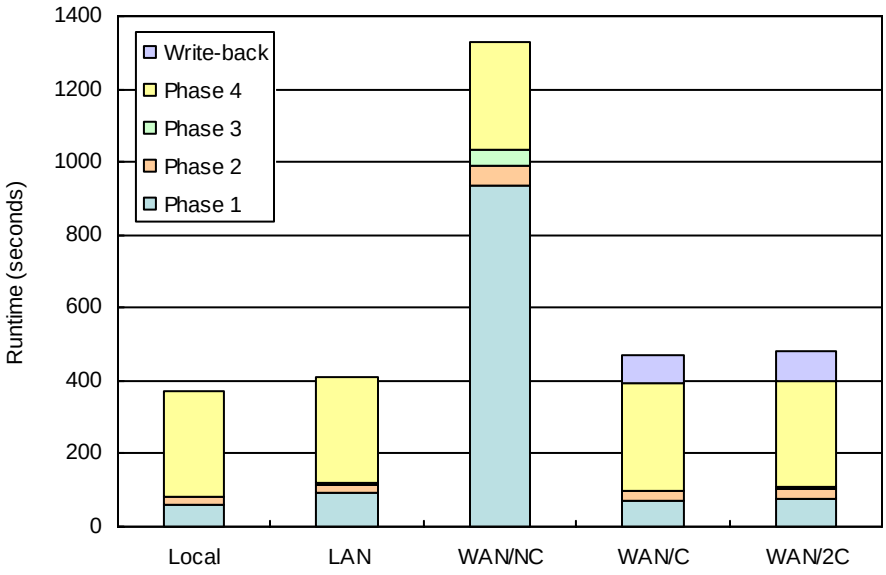

Figure 4: SPECseis benchmark execution times. The results show times for each execution phase, as well as for writing back dirty disk cache contents if necessary.

For overall runtimes in the wide-area environment, the speedup achieved by the local disk caching (WAN/C) and by the LAN disk caching (WAN/2C) is about 340% and 331% respectively (compared to WAN/NC). This is attributed to not only the leverage of temporal locality of data accesses, but also the avoidance of transfer of temporary data. In fact, the benchmark generates hundreds of Mbytes of data in the working directory during its calculations, while only tens of Mbytes are the required results. After the computing is finished, the temporal files are removed, which automatically triggers the proxy to invalidate cached data for those files. Thus when the dirty cache contents are written back to the server, only useful data are submitted, so that both bandwidth and time can be effectively saved. Furthermore, although the time used to submit dirty cache contents accounts for a considerate percentage in the overall runtime, the user/application perceived overhead can be smaller if the writing back takes advantage of idle session time or is parallelized with the computing-intensive phase, guided by the middleware scheduler.

The results from executions of the LaTeX benchmark in the various scenarios are illustrated in Figure 5. It shows that in the wide-area environment interactive users would experience much higher startup latency than the Local and LAN scenarios. During subsequent iterations, the “warm” kernel buffer helps to reduce the average runtime for WAN/NC scenario to about 39 seconds, which is still more than twice longer than the Local and LAN scenarios. The proxy disk caching on either the local disk (in WAN/C) or the LAN server (in WAN/2C) can further improve it to only 7 seconds slower than Local and LAN, but more than twice faster than kernel buffer only (in WAN/NC).

Two factors allow the combination of kernel buffer and proxy caches to outperform a solution with kernel

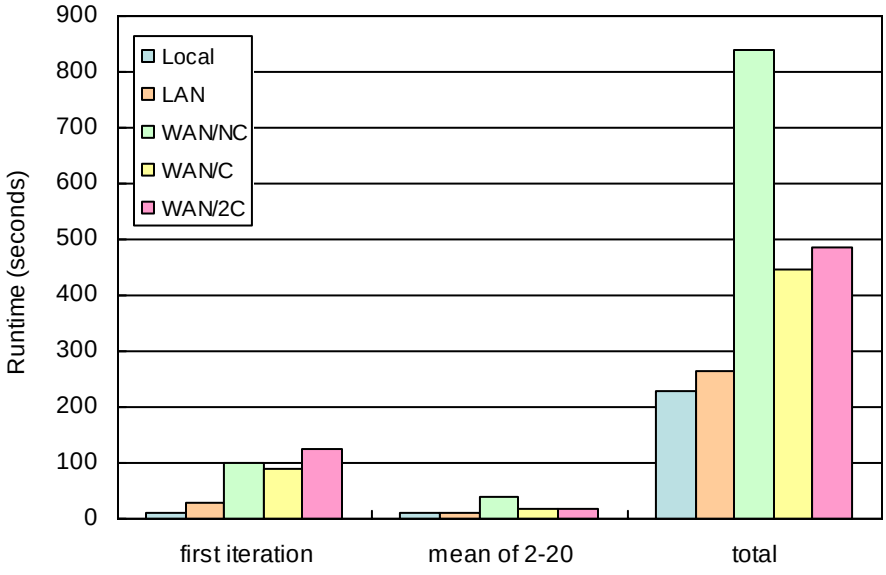

Figure 5: LaTeX Benchmark execution times. The runtimes of the first interation, the average runtimes of the following 19 iterations, and the total runtimes are listed. The time for writing back dirty proxy disk cache contents is 16s for both WAN/C and WAN/2C and summed into the total runtime.

buffer only. First, a larger storage capacity; second, the implementation of a write-back policy that allows write hits to complete without contacting the server. Furthermore, although the overhead caused by the middle-tier proxy and LAN disk caches makes the first iterations of WAN/2C to behave worse than WAN/NC, the overall runtime of the WAN/2C scenario still outperforms the WAN/NC scenario by nearly 200%. Considering the LAN cache server is capable of exploring more temporal data locality, the benefits of a second-level disk caching are even greater.

5. Related Work

Data management solutions such as GridFTP [3] and GASS [4] provide APIs upon which applications can be programmed to access data on the Grid. Legion [7] employs a modified NFS server to provide access to a remote file system. The Condor system [5] and Kangaroo [18] implement system call interception by means of either static or dynamic library linking to allow remote I/O. NeST [19] is a software Grid storage appliance that supports the NFS protocol, but only a restricted subset of the protocol and anonymous accesses are supported, and the solution does not integrate with unmodified O/S NFS clients. In contrast to these approaches, the solution of this paper allows unmodified applications to access Grid data using conventional operating system clients/servers, and supports legacy applications at the binary level. Previous effort on the UFO [9] and recently, Parrot file system [10], leverage system call tracing to allow applications to access remote files, but they require low-level process tracing capabilities that are complex to implement and highly O/S dependent, and cannot support non-POSIX compliant operations (e.g. setuid).

There are related kernel-level distributed file system solutions that exploit the advantages of disk caching and aggressive caching. For example, AFS [12] transfers and caches entire files in the client disk, CacheFS supports disk-based caching of NFS blocks, and NFS V4 [20] protocol includes provisions for aggressive caching. However, these designs require kernel support, are not able to employ per-user or per-application caching policies, and are not widely deployed in Grid setups. In contrast, VFS supports per-user/application based customization for caching, and leverages the implementations of NFS V2 and V3, which are conveniently available for a wide variety of platforms.

6. Conclusions

Grid computing promises the capability of leveraging resources distributed across wide area environments. To achieve this goal, it is important that Grid middleware provides efficient and seamless data management service

for applications. This report shows that a novel user-level approach which extends a NFS-based distributed file system virtualization with proxy managed client-side disk caching, which can be conveniently applied to unmodified applications and O/Ss. Performance evaluations based on executions of typical applications in a wide-area setup prove the proxy disk caching can remarkably improve on VFS and hence provide high-performance data provisioning on Grid resources.

VFS has been in use in the production portal of PUNCH [21] since the fall of 2000, and has supported the execution of thousands of jobs from users across the world. It is also the fundamental data management layer for In-VIGO [22], a virtualization middleware for computational Grids. Furthermore, VFS integrated with the proxy caching is also deployed in the SCOOP project [23], an In-VIGO supported Grid.

References

[1] I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, 1999.

[2] Renato J. Figueiredo, Nirav H. Kapadia, and José A. B. Fortes, *The PUNCH Virtual File System: Seamless Access to Decentralized Storage Services in a Computational Grid*, Proceedings of The Tenth IEEE International Symposium on High Performance Distributed Computing, p. 334 (2001).

[3] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, *Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing*, IEEE Mass Storage Conference, 2001.

[4] J. Bester, I. Foster, C. Kesselman, J. Tedesco and S. Tuecke, *GASS: A Data Movement and Access Service for Wide Area Computing Systems*, Proc. 6th Workshop on I/O in Parallel and Distributed Systems, May 1999.

[5] M. Litzkow, M. Livny, and M. W. Mutka, *Condor – A Hunter of Idle Workastations*, Proceedings of The 8th International Conference on Distributed Computing Systems, p. 104 (1988).

[6] D. Thain and M. Livny, *Multiple bypass: Interposition agents for distributed computing*, Journal of Cluster Computing, 4:39~47, 2001.

[7] B. White, A. Grimshaw, and A. Nguyen-Tuong, *Grid-based File Access: the Legion I/O Model*, in Proc. 9th IEEE Int. Symp. on High Performance Distributed Computing (HPDC), pp165-173, Aug 2000.

[8] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, *NFS version 3 design and implementation*, Proceedings of The USENIX Summer Technical Conference (1994).

[9] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman, *UFO: A personal global file system based on user-level extensions to the operating system*, ACM Transactions on Computer Systems, pages 207-233, August 1998.

[10] Douglas Thain and Miron Livny, *Parrot: An Application Environment for Data-Intensive Computing*, Journal of Parallel and Distributed Computing Practices, to appear in 2004.

[11] N. Kapadia, R. Figueiredo and J. A. B. Fortes, *Enhancing the Scalability and Usability of Computational Grids via Logical User Accounts and Virtual File Systems*, Proceedings of the Heterogeneous Computing Workshop (HCW) at the International Parallel and Distributed Processing Symposium (IPDPS), April 2001.

[12] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal and F. Smith, *Andrew: A Distributed Personal Computing Environment*, Communications of the ACM, 29(3) pp184-201, March 1986.

[13] R. J. Figueiredo, N. Kapadia, *J. A. B. Fortes, Seamless Access to Decentralized Storage Services in Computational Grids via a Virtual File System*, In Cluster Computing Journal 7(2), April 2004, 04/2004, p113-122.

[14] Brent Callaghan, *NFS Illustrated*, Addison-Wesley (2000).

[15] J. Hennessy and D. Patterson, *Computer Architecture: a Quantitative Approach*, 3rd edition, Morgan Kaufmann, 2002.

[16] Ming Zhao and Renato Figueiredo, *Distributed File System Support for Virtual Machines in Grid Computing*, In Proceedings of HPDC-13, 06/2004.

[17] R. Figueiredo, *VP/GFS: An Architecture for Virtual Private Grid File Systems*, In Technical Report TR-ACIS-03-001, ACIS Laboratory, Department of Electrical and Computer Engineering, University of Florida, 05/2003.

[18] D. Thain, J. Basney, S-C. Son and M. Livny, *The Kangaroo Approach to Data Movement on the Grid*, Proc. 10th IEEE Int. Symp. on High Performance Distributed Computing (HPDC), pp325-333, Aug 2001.

[19] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. Arpaci-Dusseau, R. Arpaci-Dusseau, M. Livny, *Flexibility, Manageability, and Performance in a Grid Storage Appliance*, in Proc. the Eleventh IEEE Symposium on High Performance Distributed Computing, Edinburgh, Scotland, July 2002.

[20] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson and R. Thurlow, *The NFS Version 4 Protocol*, in Proc. 2nd Intl. System Administration and Networking Conference, May 2000.

[21] N. H. Kapadia, R. J. O. Figueiredo, and J. A. B. Fortes, *PUNCH: Web Portal For Running Toools*, IEEE Micro, p. 38 (2000).

[22] S. Adabala, V. Chadha, P. Chawla, R. Figueiredo, J. Fortes, I. Krsul, A. Matsunaga, M. Tsugawa, J. Zhang, M. Zhao, L. Zhu, and X. Zhu, *From Virtualized Resources to Virtual Computing Grids: The In-VIGO System*, Future Generation Computing Systems, special issue, Complex Problem-Solving Environments for Grid Computing, 04/2004.

[23] SCOOP In-VIGO Web Portal, http://invigo-scoop.coastal.ufl.edu