Cluster Computing 9, 45–56, 2006
© 2006 Springer Science + Business Media, Inc. Manufactured in The United States.

Distributed File System Virtualization Techniques Supporting On-Demand Virtual Machine Environments for Grid Computing

MING ZHAO, JIAN ZHANG and RENATO J. FIGUEIREDO
Advanced Computing and Information Systems Laboratory, Electrical and Computer Engineering, University of Florida, Gainesville, Florida 32611, USA

Abstract. This paper presents a data management solution which allows fast Virtual Machine (VM) instantiation and efficient run-time execution to support VMs as execution environments in Grid computing. It is based on novel distributed file system virtualization techniques and is unique in that: (1) it provides on-demand cross-domain access to VM state for unmodified VM monitors; (2) it enables private file system channels for VM instantiation by secure tunneling and session-key based authentication; (3) it supports user-level and write-back disk caches, per-application caching policies and middleware-driven consistency models; and (4) it leverages application-specific meta-data associated with files to expedite data transfers. The paper reports on its performance in wide-area setups using VMware-based VMs. Results show that the solution delivers performance over 30% better than native NFS and with warm caches it can bring the application-perceived overheads below 10% compared to a local-disk setup. The solution also allows a VM with 1.6 GB virtual disk and 320 MB virtual memory to be cloned within 160 seconds for the first clone and within 25 seconds for subsequent clones.

Keywords: distributed file system, virtual machine, grid computing

1. Introduction

A fundamental goal of computational "Grids" is to allow flexible, secure sharing of resources distributed across different administrative domains [1]. To realize this vision, a key challenge that must be addressed by Grid middleware is the provisioning of execution environments that have flexible, customizable configurations and allow for secure execution of untrusted code from Grid users [2]. Such environments can be delivered by architectures that combine "classic" virtual machines (VMs) [3] and middleware for dynamic instantiation of VM instances on a per-user basis [4]. Efficient instantiation of VMs across distributed resources requires middleware support for transfer of large VM state files (e.g. memory, disk) and thus poses challenges to data management infrastructures. This paper shows that a solution for efficient and secure transfer of VM state across domains can be implemented by means of extensions to a user-level distributed file system virtualization layer.

Mechanisms that present in existing middleware can be utilized to support this functionality by treating VM-based computing sessions as processes to be scheduled (VM monitors) and data to be transferred (VM state). In order to fully exploit the benefits of a VM-based model of Grid computing, data management is key: without middleware support for transfer of VM state, computation is tied to the end-resources that have a copy of a user's VM; without support for the transfer of application data, computation is tied to the end-resources that have local access to a user's files. However, with appropriate data management support, the components of a Grid VM computing session can be distributed across three different logical entities: the "state server", which stores VM state; the "compute server", which provides the capability of instantiating VMs; and the "data server", which stores user data (Figure 1).

The proposed VM state provisioning solution is constructed upon a user-level distributed file system virtualization layer [5] which leverages the NFS [6] de-facto distributed file system standard and provides a basis for establishing dynamic **G**rid **V**irtual **F**ile **S**ystem (**GVFS**) sessions. GVFS extends upon the virtualization infrastructure at the user level to support on-demand, secure and high-performance access to Grid VM state. It leverages SSH tunneling and session-key based cross-domain authentication to provide private file system channels, and addresses performance limitations associated with typical NFS setups in wide-area environments (such as buffer caches with limited storage capacity and write-through policies) by allowing for user-level disk caching. It also supports application-driven meta-data at the file system level to allow for data requests being satisfied using partial- or full-file transfer selectively to efficiently handle VM state files. These mechanisms are implemented transparently to the kernels and applications, and hence support unmodified VM technologies, such as VMware [7], UML [8] and Xen [9], which use the file system to store VM state.

The paper also reports on the performance of this approach via experiments conducted with VMware-based VMs instantiated in wide-area environments. Experimental results show that it significantly improves the execution time of applications in Grid VMs compared to native NFS (the speedup is more than 30%), and it experiences relatively small overhead compared to VMs with locally stored state (less than 10% with warm proxy caches). Results also show that the use of on-demand transfer, disk caching and meta-data information allows fast instantiation of a large VM clone (less than 160

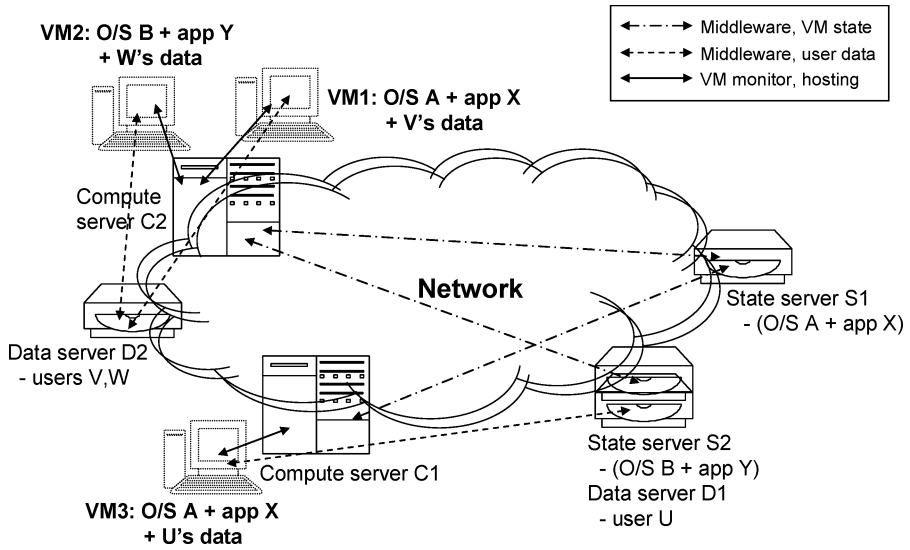

Figure 1. Middleware supported data management for both virtual machine state and user data allows for application-tailored VM instantiations (VM1, VM2 and VM3) across Grid resources (compute servers C1 and C2, state servers S1 and S2, data servers D1 and D2).

seconds for the first clone and about 25 seconds for subsequent clones), and substantially outperforms cloning via native NFS-based data access (more than 30 minutes) and SCP-based file transfer (more than 20 minutes).

The rest of this paper is organized as follows. Section 2 presents an introduction to GVFS. Sections 3, 4 and 5 describe the proposed GVFS extensions for cross-domain authentication and data encryption, caching and meta-data handling, respectively. Section 6 discusses the integration of GVFS with Grid VMs. Section 7 presents results from quantitative performance analyses. Section 8 examines related work, and Section 9 concludes the paper.

2. Background

Current Grid data management solutions typically employ file-staging techniques to transfer files between user accounts in the absence of a common file system. File staging approaches require the user to explicitly specify the files that need to be transferred (e.g. GridFTP [10]), or transfer entire files at the time they are opened (e.g. GASS [11]), which may lead to unnecessary data transfer. Data management solutions supporting on-demand transfer for Grids have also been investigated in related work, as discussed in Section 8. However, these solutions often require customized application libraries and/or file servers.

Previous work has shown that a data management model supporting cross-domain on-demand data transfers without requiring dynamically-linked libraries or changes to either applications or native O/S file system clients and servers can be achieved by way of two mechanisms—logical user accounts [12] and a distributed virtual file system [5]. Such a distributed virtual file system can be built through the use of a virtualization layer on top of NFS, a de-facto LAN distributed file system standard, allowing data to be transferred on-demand between Grid storage and compute servers for the duration of a computing session. The resulting virtual file system utilizes user level proxies to dynamically forward data requests and map between short-lived user identities allocated by middleware on behalf of a user [13].

Although the current deployment of a virtual file system session only leverages a single (per-user) native NFS server-side proxy [14], the design supports connections of proxies "in series" between a native NFS client and server. While a multi-proxy design may introduce more overhead from processing and forwarding RPC calls, there are important design goals that lead to its consideration:

Additional functionality: Extensions to the protocol can be implemented between proxies, again, without modifications to native NFS clients/servers or applications. For example, private file system channels can be realized by inter-proxy session-key authentication and encrypted data tunneling, which is explained in details in Section 3. Another possible functionality extension is inter-proxy cooperation for fine-grained cache coherence and consistency models.

Improved performance: The addition of a proxy at the client side enables the caching of file system data to improve access latency for requests that exhibit locality. For example, Section 4 describes the use of a level of caches additional to kernel-level memory buffers, enabled by proxy-controlled disk caching. Other possible extensions include inter-proxy high-speed data transfer protocols for large files (e.g. GridFTP [10]).

Figure 2 illustrates a multi-proxy GVFS setup, where two proxies work between the native NFS server and client cooperatively. The server-side proxy deployed on the VM state server *S* authenticates and forwards data accesses from the shadow account *shadow* on the compute server *C* to a VM state directory */home/vm/vm01* under the file account *vm* on *S*, and maps between the credentials of *shadow* and *vm* inside each RPC message. If the requests come from the client-side

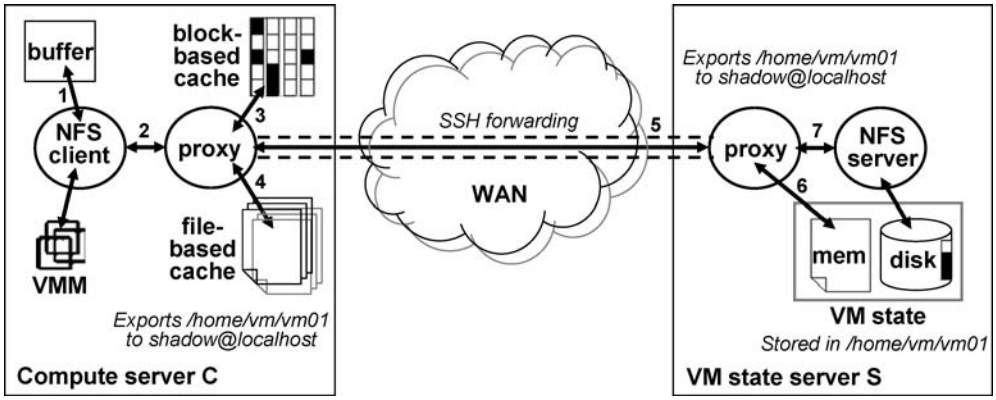

Figure 2. Proxy extensions for VM state transfers. At the compute server, the VM monitor (VMM) issues system calls that are processed by the kernel NFS client. Requests may hit in the kernel-level memory buffer cache (1); those that miss are processed by the user-level proxy (2). At the proxy, requests that hit in the block-based disk cache (3), or in the file-based disk cache if matching stored meta-data (4), are satisfied locally; proxy misses are forwarded as SSH-tunneled RPC calls to a remote proxy (5), which fetches data directly (for VM memory state) (6) or through the kernel NFS server (for VM disk state) (7).

proxy directly, the server-side proxy should export the directory to *shadow@C*. But in this example, the connections are forwarded via SSH tunneling, so the directory is exported to *shadow@localhost*. On the other hand, the client-side proxy on *C* authenticates and forwards data access from *shadow@localhost* to the server-side proxy, and more importantly, it implements the extensions to enable more functionality and better performance. These extensions will be discussed in details in the following sections, including private file system channels (Section 3), client-side proxy disk caching (Section 4) and application-specific meta-data handling (Section 5).

3. Private file system channels

3.1. Secure data tunneling

Security is always a major concern for data provisioning across wide-area environments. In the context of RPC-based applications, security in communication can be provided within or outside the RPC protocol. A key advantage of the latter approach lies in the fact that existing RPC-based clients and servers can be reused without modifications; it is the approach taken by GVFS.

Secure RPC-based connections can be established through the use of TCP/IP tunneling. A tunnel allows the encapsulation and encryption of datagrams at the client side, and corresponding decryption and de-capsulation at the remote site. It supports private communication channels in an application-transparent manner. The application-transparent property of tunneling is a key advantage of this technique and has found wide use in applications such as Virtual Private Networks (VPNs) and secure remote X-Windows sessions.

Tunneling of RPC-based connections can be achieved through mechanisms such as SSL and SSH. The latter is a de-facto standard for secure logins, and provides strong authentication, data privacy and integrity for remote login sessions, as well as tunneling of arbitrary TCP connections. GVFS leverages the functionality of SSH to create authenticated, encrypted tunnels between client-side and server-side proxies. (Tunneled GVFS connections are TCP-based. This, however, does not prevent GVFS from supporting UDP-based kernel clients and servers. A client-side proxy can receive RPC calls over UDP from localhost and forward to the server-side proxy using TCP, and the server-side proxy can receive RPC calls over the TCP tunnel and forward to localhost using UDP.)

The use of SSH to tunnel NFS traffic has been pursued by related efforts, such as Secure NFS [15]. A key differentiator of GVFS from previous approaches is that private file system sessions are established *dynamically by middleware on a per-session basis*, rather than statically by a system administrator for groups of users. Another key difference is the GVFS support for per-user identity mappings across network domains. Per-user tunnels and user mappings are key to establishing dynamic file system sessions in a Grid-oriented environment, where users belong to different administrative domains. A secure tunnel multiplexed by users faces the same limitations for cross-domain authentication as NFS, since RPC-based security must be used to authenticate users within a tunnel [16]. With a secure connection and per-user file system channels, the task of authenticating users can be independently carried out by each private file system channel, and the task of guaranteeing privacy and integrity can be leveraged from the secure connection.

3.2. Security model

GVFS private file system channels rely on existing kernel-level services at the client (file system mounting), server (file system exporting), user-level middleware-controlled proxies at both client and server, and SSH tunnels established between them. Hence, the deployment of GVFS involves the setup of appropriate trust relationships between client, server and middleware.

In the GVFS security model, it is assumed that the data management middleware supports logical user accounts: it can authenticate to a server-side "file account" and to a client-side "shadow account" as described in [12]. The data server administrator needs to trust Grid middleware to the extent that it allows access to an exported directory tree to be brokered by proxies (e.g. */home/vm/vm01* on server *S* in Figure 2). In essence, the server administrator delegates access control to one or more exported directories to the Grid middleware. This is a trust relationship that is similar to those found in other Grid data deployments (e.g. GridFTP [10]). The architecture of GVFS allows kernel export definitions to be implemented by local system administrators in a simple manner—the kernel server exports only to the localhost, and only the directories that should be accessible via GVFS. Users outside the localhost cannot directly mount file systems from the kernel—only via server-side proxies. A typical scenario, where a base home directory for Grid users is exported through GVFS, requires a single entry in an exports definition file.

Then, server-side proxies are responsible for authenticating accesses to those file systems exported by GVFS. This is accomplished by means of two mechanisms. The first authentication mechanism is independent from the proxy and consists of the client machine being able to present appropriate credentials (an SSH key, or an X.509 certificate for GSI-enabled SSH) to the server machine to establish a tunnel. Second, once the tunnel is established, it is necessary for the server-side proxy to authenticate requests received through it. Typically, NFS servers authenticate client requests by checking the origin of NFS calls and only allowing those that come from privileged ports of trusted IPs to proceed. In the GVFS setup, the originator of requests sent to the server-side proxy is the server's tunnel end-point. Hence, the server-side proxy receives requests from the localhost and from non-privileged ports and cannot authenticate the client based on trusted IP/port information. It thus becomes necessary to implement an alternative approach for inter-proxy authentication between tunnel end-points.

The approach of this paper consists of the dynamic creation of a random session key by middleware at the time the server-side proxy is started and its transmission over a separate private channel to the client-side proxy. Then the client-side proxy appends the key to each NFS procedure call, and the server-side proxy only authenticates a coming request if it is originated from the localhost and it has a session key that matches the server-side proxy's key. Hence, the use of session keys is completely transparent to kernel clients and servers and requires no changes to their implementations; it only applies to inter-proxy authentication between tunnel end-points. These session keys are used for authentication, similarly to X11/xauth, but not for encryption purposes. In the implementation, a session key is a randomly generated 128-bit string and encapsulated in original NFS RPC messages by replacing an unused credential field, so the run-time overhead of supporting this method is very small, consisting of only encapsulation and decapsulation of a session key, and a simple comparison between key values.

The server-side proxy thus needs to trust the Grid middleware security infrastructure to authenticate user credentials and establish an encrypted tunnel, create a random session key, and provide the key to the client-side proxy through a separate private channel. These mechanisms can be provided by existing Grid security infrastructure, such as Globus GSI. Finally, the client administrator needs to trust Grid middleware to the extent that it needs to allow NFS mount and unmount operations to be initiated by Grid middleware (possibly within a restricted set of allowed base directories, e.g. */home/vm/vm01/** in Figure 2). In current GVFS setups, this is implemented with the use of *sudo* entries for these commands.

4. Client-side proxy disk caching

4.1. Designs

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 [16]. 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 [17]. 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 GVFS is implemented by the file system proxy. A virtual file system 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. GVFS disk caching operates at the granularity of NFS RPC calls. The cache is generally 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.

GVFS proxy disk 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. Furthermore, write-back disk caching can avoid transfer of temporary files. After performing computing, a user or data scheduler can remove temporary files from the

working directory, which automatically triggers the proxy to invalidate cached dirty 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.

There are several distributed file systems that exploit the advantages of disk caching too, for example, AFS [18] 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, GVFS is unique to support customization on a per-user/application basis [19]. For instance, the cache size and write policy can be configured by user requirements/priorities, or optimized according to the knowledge of a Grid application. A more concrete example is enabling heterogeneous disk caching by meta-data handling and application-tailored knowledge, which is employed to support block-based caching for VM disk state and file-based caching for VM memory state (Section 5).

4.2. Deployment

Disk caching is beneficial to many VM technologies, including VMware [7], UML [8] and Xen [9], where VM state (e.g. virtual disk, memory) is stored as regular files or filesystems. As GVFS sessions are dynamically setup by middleware, disk caches are also dynamically created and managed by proxies on per-session basis. When a GVFS session starts, the proxy initializes the cache with middleware configured parameters, including cache path, size, associativity and policies. During the session, some of the parameters, including cache write and consistency policies, can also be reconfigured. When the session finishes, policies implemented by Grid middleware can drive the proxy to flush, write-back or preserve cached contents.

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 cache described in this paper 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.

Such middleware-driven consistency is sufficient to support many Grid applications, e.g. when tasks are known to be independent by a scheduler for high-throughput computing. This model is assumed in this paper because under a VM management system, such as VMPlant [20] and VMware VirtualCenter [21]: a VM with persistent state can be dedicated to a single user, where aggressive read and write caching with write delay can be used; a VM with non-persistent state can be read-shared among users while each user has independent redo logs, where read caching for state files and write-back caching for redo logs can be employed. Furthermore, it is also possible to achieve fine-grained cache coherence and consistency models by implementing call back and other inter-proxy coordination mechanisms, which are subjects of on-going investigations.

While caches of different proxies are normally independently configured and managed, it also allows 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 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.

5. Application-specific meta-data handling

Another extension made to GVFS is the handling of meta-data information. The main motivation is to use middleware information to generate meta-data for certain categories of files according to the knowledge of Grid applications. Then, a GVFS proxy can take advantage of the meta-data to improve data transfer. Meta-data contains the data characteristics of the file it is associated with, and defines a sequence of actions which should be taken on the file when it is accessed, where each action can be described as a command or in a script. When the proxy receives a NFS request to a file which has meta-data associated with, it processes the meta-data and executes the required actions on the file accordingly. In the current implementation, the meta-data file is stored in the same directory as the file it is associated with, and has a special filename so that it can be easily looked up.

For example, resuming a VMware VM requires reading the entire memory state file (typically in hundreds of MBytes). Transferring the entire contents of this file is time-consuming; however, with application-tailored knowledge, it can be pre-processed to generate a meta-data file specifying which blocks in the memory state are all zeros. Then, when the memory state file is requested, the client-side proxy, through processing of the meta-data, can service requests to zero-filled blocks locally, ask for only non-zero blocks from the server, then reconstruct the entire memory state and present it to the VM monitor. Normally the memory state contains many zero-filled blocks that can be filtered by this technique [22], and the traffic on the wire can be greatly reduced while instantiating a VM. For instance, when resuming a 512 MB-RAM Red Hat 7.3 VM which is suspended after boot-up, the client issues 65,750 NFS reads while 60,452 of them can be filtered out by this technique.

Another example of GVFS' meta-data handling capability is to help the transfer of large files and enable file-based disk caching. Inherited from the underlying NFS protocol, data transfer in GVFS is on-demand and block-by-block based

(typically 4 K to 32 Kbytes per block), which allows for partial transfer of files. Many applications can benefit from this property, especially when the working set of the accessed files are considerably smaller than the original sizes of the files. For example, accesses to the VM disk state are typically restricted to a working set that is much smaller (<10%) than the large disk state files. But when large files are indeed completely required by an application (e.g. when a remotely stored memory state file is requested by VMware to resume a VM), block-based data transfer becomes inefficient.

However, if Grid middleware can speculate in advance which files will be entirely required based on its knowledge of the application, it can generate meta-data for GVFS proxy to expedite the data transfer. The actions described in the meta-data can be "compress", "remotely copy", "uncompress" and "read locally", which means when the referred file is accessed by the client, instead of fetching the file block by block from the server, the proxy will: (1) compress the file on the server (e.g. using GZIP); (2) remotely copy the compressed file to the client (e.g. using GSI-enabled SCP) ; (3) uncompress it to the file cache (e.g. using GUNZIP); and (4) generate results for the request from the locally cached file. Once the file is cached all the following requests to the file will also be satisfied locally (Figure 2).

Hence, the proxy effectively establishes an on-demand fast file-based data channel, which can also be secure by employing SSH tunneling for data transfer, in addition to the traditional block-based NFS data channel, and a file-based cache which complements the block-based cache in GVFS to form a heterogeneous disk cache. The key to the success of this technique is the proper speculation of an application's behavior. Grid middleware should be able to accumulate knowledge for applications from their past behaviors and make intelligent decisions based on the knowledge. For instance, since for VMware the entire memory state file is always required from the state server before a VM can be resumed on the compute server, and since it is often highly compressible, the above technique can be applied very efficiently to expedite its transfer.

6. Integration with VM-based Grid computing

VMs can be deployed in a Grid in two different kinds of scenarios, which pose different requirements of data management to the distributed virtual file system. In the first scenario, the Grid user is allocated a dedicated VM which has a persistent virtual disk on the state server. It is suspended at the current state when the user leaves and resumed when the user comes again, while the user may or may not start computing sessions from the same server. When the session starts, the VM should be efficiently instantiated on the compute server, and after the session finishes, the modifications to the VM state from the user's executions should also be efficiently reflected on the state server. The extended GVFS can well support this scenario in that: (1) the use of meta-data handling can quickly restore the VM from its checkpointed state; (2) the on-demand block-based access pattern to the virtual disk can avoid the large overhead incurred from downloading and uploading the entire virtual disk; (3) proxy disk caches can exploit locality of references to the virtual disk and provide high-bandwidth, low-latency accesses to cached file blocks; (4) write-back caching can effectively hide the latencies of write operations perceived by the user, which are typically very large in a wide-area environment, and submit the modifications when the user is off-line or the session is idle.

In the other scenario, the state server stores a number of non-persistent VMs for the purpose of "cloning". These generic VMs have application-tailored hardware and software configurations, and when a VM is requested from a compute server, the state server is searched against the requirements of the desired VM. The best match is returned as the "golden" VM, which is then "cloned" at the compute server [20]. The cloning process entails copying the "golden" VM, restoring it from checkpointed state, and setting up the clone with customized configurations. After the new clone "comes to life", computing can start in the VM and modifications to the original state are stored in the form of redo logs. So data management in this scenario requires efficient transfer of the VM state from the state server to the compute server, and also efficient writes to the redo logs for checkpointing.

Similar to the first scenario, the extended GVFS can quickly instantiate a VM clone by using meta-data handling for the memory state file and on-demand block-based access to the disk state files. Instead of copying the entire virtual disk, only symbolic links are made to the disk state files on the compute server. After a computation starts, the proxy disk cache can help speedup access to the virtual disk after the cache becomes "warm", and write-back can help save user time for writes to the redo logs. However, a differentiation in this scenario is that a small set of golden VMs can be used to instantiate many clones, e.g. for concurrent execution of a high-throughput task. The proxy disk caches can exploit temporal locality among cloned instances and accelerate the cloning process. On the compute server, the cached data of memory and disk state files from previous clones can greatly expedite new clonings from the same golden VMs. And a second-level proxy cache can be setup on a LAN server, as explained in Section 4.2, to further exploit the locality of golden VMs which are cloned to computer servers in the same local network.

7. Performance evaluation

7.1. Experimental setup

A prototype of the approach discussed in this paper has been built upon the implementation of middleware-controlled user-level file system proxies. The core proxy code described in [5] has been extended to support private file system channels, client-side disk caching and meta-data handling. This section evaluates the performance of the proposed techniques

for supporting VMs in the Grid by analyzing the data from experiments of a group of typical benchmarks.

Experiments are conducted in both local-area and wide-area environments. The LAN state server is a dual-processor 1.3GHz Pentium-III cluster node with 1 GB of RAM and 18 GB of disk storage. The WAN state server is a dual-processor 1 GHz Pentium-III cluster node with 1 GB RAM and 45 GB disk. In Sections 7.2 and 7.3, the compute server is a 1.1 GHz Pentium-III cluster node with 1 GB of RAM and 18 GB of SCSI disk; in Section 7.4, the compute servers are cluster nodes which have two 2.4 GHz hyper-threaded Xeon processors with 1.5 GB RAM and 18 GB disk each. All of the compute servers run VMware GSX server 2.5 to support x86-based VMs. The compute servers are connected with the LAN state server in a 100 Mbit/s Ethernet at the University of Florida, and connected with the WAN state server through Abilene between Northwestern University and University of Florida. The RTT from the computer servers to the LAN state server is around 0.17 msec, while to the WAN state server is around 32 msec as measured by RTTometer [23].

In the experiments on GVFS with proxy caching, the cache is configured with 8 GByte capacity, 512 file banks and 16-way associativity. The proxy cache prototype currently supports NFS version 2, which limits the maximum size of an on-the-wire NFS read or write operation to 8 KB. Thus NFS version 2 with 8 KB block size is used for GVFS when the proxy caching is enabled. However, in the experiments on native NFS, NFS version 3 with 32 KB block size is used. Furthermore, all the experiments are initially setup with "cold" caches (both kernel buffer cache and possibly enabled proxy disk cache) by un-mounting and re-mounting the GVFS partition and flushing the proxy cache if it is used. Private file system channels are always employed in GVFS during the experiments.

7.2. Performance of private file system channels

Before the evaluation of GVFS' support for Grid VMs, it is necessary to understand the performance of GVFS itself compared to local-area NFS and local-disk file system. Previous work has shown the proxy-based distributed virtual file system performs well in comparison to native NFS in a local-area setup [5]. Hence here the investigation focuses on the performance of private file system channels enabled by GVFS and the performance of GVFS in wide-area environments. Experimental results shown in this subsection consider application-perceived performance measured as elapsed execution times for the following benchmarks:

SPECseis: a benchmark from the SPEC high-performance group. It consists of four phases, where the first phase generates a large trace file on disk and the last phase involves intensive seismic processing computations. The benchmark is tested in sequential mode with the small dataset. It models a scientific application that is both I/O-intensive and compute-intensive.

LaTeX: a benchmark designed to model an interactive document processing session. It is based on the generation of a PDF (Portable Document File) version of a 190-page document edited by LaTeX. It runs the "latex", "bibtex" and "dvipdf" programs in sequence and iterates 20 times, where each time a different version of one of the LaTeX input files is used.

These benchmarks are executed on the compute servers in various environments, where the working directories are either stored on local disks or mounted from the remote LAN or WAN state servers. To investigate the overhead incurred by private file system channels, in the LAN environment the performance of native NFS (**LAN/N**) is compared with GVFS (**LAN/G**) without proxy caches. Experiments conducted in the WAN environment must use GVFS to ensure data privacy and passing firewalls, but the performance of GVFS without proxy caches (**WAN/G**) and with caches (**WAN/GC**) are both measured against the performance of the local disk (**Local**), so as to investigate the overhead of GVFS and the performance improvement achieved by proxy caches.

The experiment results are summarized in Table 1. Consider the execution of the LaTeX benchmark. In the LAN scenarios, the overhead of private file system channels (LAN/N vs. LAN/G) is large at the beginning but is substantially reduced once the kernel buffer cache holds the working dataset. (The overhead of GVFS here consists of the overhead incurred by SSH tunneling and proxy processing of RPC calls, but for a relatively fast server, the latter is considerably small [5].) The results also show that the kernel buffer cache alone is not sufficient to lower WAN execution time of the LaTeX benchmark

Table 1

GVFS overheads for LaTeX and SPECseis benchmarks. The overhead data are calculated by comparing the execution times of the benchmarks in different scenarios: Local disk (Local), LAN on NFS (LAN/N), LAN on GVFS (LAN/G), WAN on GVFS without proxy caches (WAN/G) and with proxy caches (WAN/GC). For LaTeX benchmark, the comparisons for the execution time of the first iteration, the average execution time of the second to the twentieth iterations and the total execution time are listed. For SPECseis benchmark, the comparisons for the execution time of the first phase, the fourth phase and the total are listed. For both benchmarks, in the WAN/GC scenario the write-back cached data are submitted to server after the executions and the time is summed into the total execution time.

	LaTeX			SPECseis		
Overhead	1st run (%)	2nd–20th run (%)	Total (%)	Phase 1 (%)	Phase 4 (%)	Total (%)
LAN/G vs. LAN/N	124	7	13	47	0	9
WAN/G vs. Local	797	180	215	1500	1	265
WAN/GC vs. Local	691	17	60	24	0	26

in WAN/G, but the proxy cache can remarkably reduce the overhead to 17% in WAN/GC compared to Local. Two factors allow the combination of kernel buffer and proxy caches to outperform a solution with kernel 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.

For the execution of the SPECseis benchmark, the compute-intensive phase (phase 4) as expected achieves very close performance in all scenarios, but the performance of the I/O-intensive phase (phase 1) differentiates very much. It is reasonable to see the overhead of private file system channels becomes larger as more network communication requires more time for SSH tunneling. The overhead is especially large in WAN/G due to much higher network latency in the wide-area environment; however in WAN/GC the proxy cache effectively hides the latency and significantly reduces the overhead to 24% relative to Local. Besides, the write-back caching also helps to improve performance by avoiding transfer of temporary data to server. 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. The dirty cache contents are written back to the server after the execution and the time is summed into the total execution time of WAN/GC which is less than tenth of that of WAN/G.

In overall, the performance of GVFS' private file system sessions is close to NFS in LAN within 10% for both benchmarks, and with the help from proxy caches the overhead in WAN is within 20% for the LaTeX benchmark and within 30% for the SPECseis benchmark relative to the local-disk file system. Based on these observations, the following subsections focus on the performance of GVFS' support for VMs in Grid environments.

7.3. Performance of application execution within VMs

This subsection employs the same two benchmarks introduced in the previous subsection. The LaTeX benchmark is used to study a scenario where users interact with a VM to customize an execution environment for an application that can then be "cloned" by other users for execution [20]. In this environment, it is important that interactive sessions for VM setup show good response times to the Grid users. SPECseis is used to study the performance of an application that exhibits a mix of compute-intensive and I/O-intensive phases. In addition, a third benchmark is used to evaluate the performance of applications executing on GVFS-mounted VM environments:

Kernel compilation: a benchmark that represents file system usage in a software development environment, similar to the Andrew benchmark [24]. The kernel is a Linux 2.4.18 with the default configurations in a Red Hat 7.3 Workstation deployment, and the compilation consists of four major steps, "make dep", "make bzImage", "make modules" and "make modules_install", which involve substantial reads and writes on a large number of files.

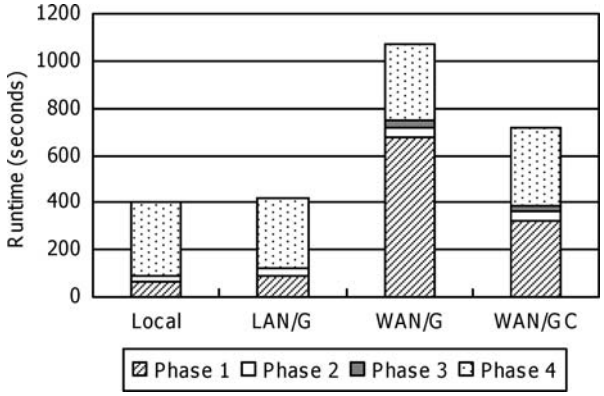

Figure 3. SPECseis benchmark execution times. The results show the runtimes for each execution phase in various scenarios.

The execution times of the above benchmarks within a VM, which has 512 MB RAM and 2 GB disk (in VMware plain disk mode [25]), installed with Linux Red Hat 7.3, the benchmark applications and their data sets, are measured in the following four different scenarios:

Local: The VM state files are stored in a local-disk file system.

LAN/G: The VM state files are stored in a directory mounted from the LAN state server via GVFS without proxy caches.

WAN/G: The VM state files are stored in a directory mounted from the WAN state server via GVFS without proxy caches.

WAN/GC: The VM state files are stored in a directory mounted from the WAN state server via GVFS with proxy caches.

Figure 3 shows the execution times for the four phases of the SPECseis benchmark. The performance of the compute-intensive part (phase 4) is within a 10% range across all scenarios. The results of the I/O intensive part (phase 1), however, shows a large difference between the WAN/G and WAN/GC scenarios—the latter is faster by a factor of 2.1. The benefit of a write-back policy is evident in the phase 1, where a large file that is used as an input to the following phases is created. The proxy cache also brings down the total execution time by 33 percent in the wide-area environment.

The LaTeX benchmark results in Figure 4 show that in the wide-area environment interactive users would experience a start-up latency of 225.67 seconds (WAN/G) or 217.33 seconds (WAN/GC). This overhead is substantial when compared to Local and LAN, which execute the first iteration in about 12 seconds. Nonetheless, the start-up overhead in these scenarios is much smaller than what one would experience if the entire VM state has to be downloaded from the state server at the beginning of a session (2818 seconds). During subsequent iterations, the kernel buffer can help to reduce the average response time for WAN/G to about 20 seconds. The proxy disk cache can further improve it for WAN/GC to very close to Local (8% slower) and LAN/G (6% slower), and 54% faster than WAN/G. The time needed to submit cached dirty blocks

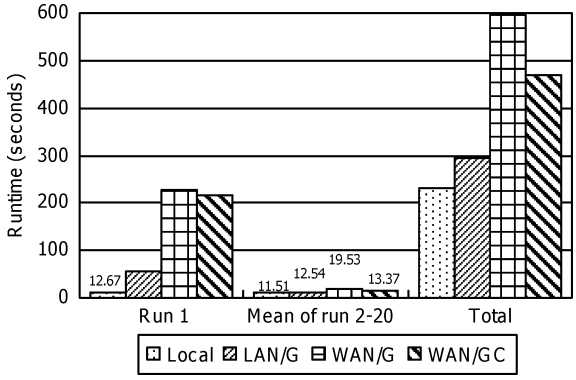

Figure 4. LaTeX benchmark execution times in various scenarios. The runtimes of the first run, the average runtimes of the following 19 runs, and the total runtimes are listed.

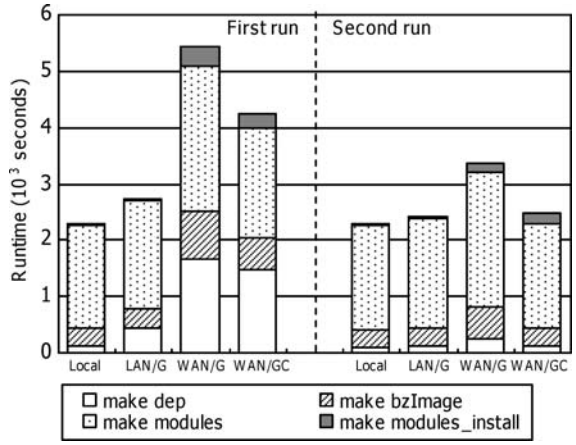

Figure 5. Kernel compilation benchmark execution times in various scenarios. The results show the runtimes for four different phases in two consecutive runs of the benchmark.

is around 160 seconds, which is also much shorter than the uploading time (4633 seconds) of the entire VM state.

Experimental results from the kernel compilation benchmark are illustrated in Figure 5. The first run of the benchmark in the WAN/GC scenario which begins with "cold" caches shows an 84% overhead compared to that of the Local scenario. However, for the second run, the "warm" caches help to bring the overhead down to 9%. And compared to the second run of the LAN scenario, it is less than 4% slower. The availability of the proxy cache allows WAN/GC to outperform WAN/G by more than 30 percent. As in the LaTeX case, the data show that the overhead experienced in an environment where program binaries and/or datasets are partially reused across iterations (e.g. in application development environments), the response times of the WAN-mounted virtual file system are acceptable.

7.4. Performance of VM cloning

Another benchmark is designed to investigate the performance of VM cloning under GVFS. The cloning scheme is as discussed in Section 6, which includes copying the configuration file, copying the memory state file, building symbolic links to the disk state files, configuring the clone, and at last resuming the new VM. The execution time of the benchmark is also measured in five different scenarios:

Local: The VM state files are stored in a local-disk file system.

WAN-S1: The VM state files are stored in a directory mounted from the WAN state server via GVFS. During the experiment, a single VM is cloned eight times to the compute server sequentially. The clonings are supported by GVFS with all extensions, including private file system channel, proxy disk caching and meta-data handling. It is designed to evaluate the performance when there is temporal locality among clonings.

WAN-S2: The setup is the same as WAN-S1 except that eight different VMs are each cloned once to the computer server sequentially. It is designed to evaluate the performance when there is no locality among clonings.

WAN-S3: The setup is the same as WAN-S2 except that a LAN server provides second-level proxy disk cache to the compute server. Eight different VMs are cloned, which are new to the compute server, but are pre-cached on the LAN server due to previous clones for other computer servers in the same LAN. This setup is designed to model a scenario where there is temporal locality among the VMs cloned to compute servers in the same LAN.

WAN-P: The VM state files are stored in a directory mounted via GVFS from the WAN state server to eight computer servers, which are eight nodes of a cluster. In the experiment, eight VMs are cloned to the compute servers in parallel.

Figure 6 shows the cloning times for a sequence of VMs which have 320 MB of virtual memory and 1.6 GB of virtual disk. In comparison with the range of GVFS-based cloning times shown in the figure, if the VM is cloned using SCP for full file copying, it takes approximately twenty minutes to transfer the entire state. If the VM state is not copied but read from a native NFS-mounted directory, the cloning takes

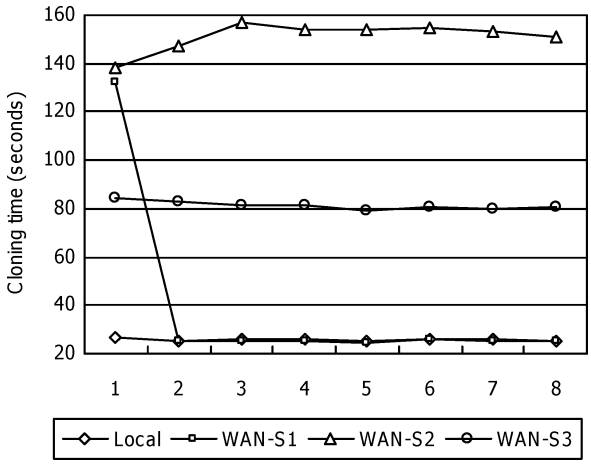

Figure 6. The times for a sequence of VM clonings (from 1 to 8). Each cloned VM has 320 MB of virtual memory and 1.6 GB of virtual disk. The results show the VM cloning times in different scenarios.

Table 2
Total time of cloning eight VMs in WAN-S1 and WAN-P when the caches (kernel buffer cache, proxy block-based cache and proxy file-based cache) are cold and warm.

	Total time when caches are cold	Total time when caches are warm
WAN-S1	1056 seconds	200 seconds
WAN-P	150.3 seconds	32 seconds

more than half an hour because the block-based transfer of the memory state file is very slow. However, the enhanced GVFS with proxy disk caches and meta-data support to compress (using GZIP) and transfer (using SCP) the VM's memory state can greatly speed up the cloning process to within 160 seconds. Furthermore, if there is temporal locality of access to the memory and disk state files among the clones, the proposed solution even allows the cloning to be performed within 25 seconds if data are cached on local disks or within 80 seconds if data are cached on a LAN server.

Table 2 compares sequential cloning with parallel cloning. In the experiment of WAN-P, the eight compute servers share a single state server and server-side GVFS proxy. But when the eight clonings start in parallel, each client-side GVFS proxy spawns a file-based data channel to fetch the memory state file on demand. The speedup from parallel cloning versus sequential cloning is more than 700% when the caches are cold and more than 600% when the caches are warm. Compared with the average time to clone a single VM in the sequential case, the total time for cloning eight VMs in parallel is 14% longer with cold caches and 24% longer with warm caches, which implies GVFS' support for VM cloning can scale to parallel cloning of large number of VMs very well. In both scenarios, the support from GVFS is on-demand and transparent to user and VM monitor. And, as demonstrated in Section 7.3, following a VM's instantiation via cloning, GVFS can also improve its run-time performance substantially.

8. Related work

Data management solutions such as GridFTP [10] and GASS [11] provide APIs upon which applications can be programmed to access data on the Grid. Legion [26] employs a modified NFS server to provide access to a remote file system. The Condor system [27] and Kangaroo [28] implement system call interception by means of either static or dynamic library linking to allow remote I/O. NeST [29] is a Grid storage appliance that supports only a restricted subset and anonymous accesses of the NFS protocol, and it 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 [30] and recently, Parrot file system [31], 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).

The self-certifying file system (SFS [32]) is another example of a file system that uses proxies to forward NFS protocol calls and implement cross-domain authentication and encryption. The approach of this paper differs from SFS in that it employs dynamically-created per-user file system proxies, allowing for middleware-controlled caching policies (e.g. write-back vs. write-through) on a per-user basis, and the setup of multiple levels of proxy caching. In contrast, SFS employs a single proxy server for multiple users.

GVFS uses SSH (or GSI-SSH) for the creation of encrypted tunnels rather than a customized algorithm. Implementations of SSH-based NFS tunnels have been pursued by the Secure NFS project [15]. But such tunnels are created statically by system administrators and are multiplexed by several users, while GVFS is capable of establishing dynamic private file system channels on a per-user basis.

There are related kernel-level DFS solutions that exploit the advantages of disk caching and aggressive caching. For example, AFS [18] transfers and caches entire files in the client disk, CacheFS supports disk-based caching of NFS blocks, and NFS V4 [33] 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, GVFS 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.

A related project [22] has investigated the optimizations on the migration of VMs, possibly across low-bandwidth links; [34] proposes a system that delivers virtual desktops to personal computer users via cache-based system management model. Common between their approaches and this paper are mechanisms supporting on-demand block transfers and VMM-tailored disk caches. The work presented in [35] also introduces techniques for low overhead live migration of VMs in LAN environments. In contrast to these approaches, the data management solution of this paper supports efficient instantiation of VMs in cross-domain wide-area environments, and the techniques are applicable to any VM technology that uses file systems to store VM state.

9. Conclusions

Grid computing with classic virtual machines promises the capability of provisioning a secure and highly flexible computing environment for its users. To achieve this goal, it is important that Grid middleware provides efficient data management service for VMs—for both VM state and user data. This paper shows user-level techniques that build on top of de-facto distributed file system implementations can provide an efficient framework for this purpose. These techniques can be applied to VMs of different kinds, so long as the VM monitor allows for VM state to be stored in file systems that can be mounted via NFS.

GVFS provides private file system channels for VM instantiation and leverages proxy disk caching to improve on the performance. Experimental results show that with "warm" caches the overhead of running applications inside VMs which are instantiated over a WAN via GVFS is relatively small compared to VMs stored on a local-disk file system. Results also show that the use of on-demand transfers and meta-data information allows instantiation of a 320 MB-RAM/1.6GB-disk Linux VM clone in less than 160 seconds for the first clone and less than 25 seconds for subsequent clones, considerably outperforming the cloning based on either transfer of entire VM state or native NFS-based state access.

Acknowledgments

Effort sponsored by the National Science Foundation under grants EIA-0224442, ACI-0219925, EEC-0228390 and NSF Middleware Initiative (NMI) grants ANI-0301108 and SCI-0438246. The authors also acknowledge a gift from VMware Inc. and SUR grants from IBM. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, IBM, or VMware. The authors would like to thank Peter Dinda at Northwestern University for providing access to resources.

References

[1] I. Foster, C. Kesselman and S. Tuecke, The anatomy of the grid: Enabling scalable virtual organizations, International Journal of Supercomputer Applications 15(3) (2001).

[2] A. Butt, S. Adabala, N. Kapadia, R. Figueiredo and J. Fortes, Grid-computing portals and security issues, Journal of Parallel and Distributed Computing 63(10) (2003) 1006–1014.

[3] R.P. Goldberg, Survey of virtual machine research, IEEE Computer Magazine 7(6) (1974) 34–45.

[4] R.J. Figueiredo, P.A. Dinda and J.A.B. Fortes, A case for grid computing on virtual machines, in: *Proc. International Conference on Distributed Computing Systems* (May 2003).

[5] R.J. Figueiredo, N. Kapadia and J.A.B. Fortes, Seamless access to decentralized storage services in computational grids via a virtual file system, Cluster Computing Journal 7(2) (2004) 113–122.

[6] B. Pawlowski, C Juszczak, P. Staubach, C. Smith, D. Lebel and D. Hitz, NFS version 3 design and implementation, in: *Proc. USENIX Summer Technical Conference* (1994).

[7] J. Sugerman, G. Venkitachalan and B-H. Lim, Virtualizing I/O devices on vmware workstation's hosted virtual machine monitor, in: *Proc. USENIX Annual Technical Conference* (June 2001).

[8] J. Dike, A user-mode port of the linux kernel, in: *Proc. the 4th Annual Linux Showcase and Conference*, USENIX Association, Atlanta, GA (October 2000).

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield, Xen and the art of virtualization, in: *Proc. ACM Symposium on Operating Systems Principles* (October 2003).

[10] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel and S. Tuecke, Secure, efficient data transport and replica management for high-performance data-intensive computing, in: *Proc. IEEE Mass Storage Conference* (2001).

[11] J. Bester, I. Foster, C. Kesselman, J. Tedesco and S. Tuecke, GASS: A data movement and access service for wide area computing systems, in: *Proc. the 6th Workshop on I/O in Parallel and Distributed Systems* (May 1999).

[12] N. Kapadia, R.J. Figueiredo and J.A.B. Fortes, Enhancing the scalability and usability of computational grids via logical user accounts and virtual file systems, in: *Proc. Heterogeneous Computing Workshop at the International Parallel and Distributed Processing Symposium* (April 2001).

[13] S. Adabala, A. Matsunaga, M. Tsugawa, R.J. Figueiredo and J.A.B. Fortes, Single sign-on in in-vigo: Role-based access via delegation mechanisms using short-lived user identities, in: *Proc. International Parallel and Distributed Processing Symposium* (April 2004).

[14] S. Adabala, V. Chadha, P. Chawla, R.J. Figueiredo, J.A.B. 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 on Complex Problem-Solving Environments for Grid Computing, Vol 21/6, pp. 896–909.

[15] J.C. Bowman, Secure NFS via SSH tunnel, http://www.math.ualberta.ca/imaging/snfs/

[16] B. Callaghan, *NFS Illustrated* (Addison-Wesley, 2002).

[17] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd edition (Morgan Kaufmann, 2002).

[18] 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) (March 1986) 184–201.

[19] M. Zhao, V. Chadha and R.J. Figueiredo, Supporting application-tailored grid file system sessions with wsrf-based services, in: *Proc. the 14th IEEE International Symposium on High Performance Distributed Computing* (July 2005).

[20] I.V. Krsul, A. Ganguly, J. Zhang, J.A.B. Fortes and R.J. Figueiredo, VMPlants: Providing and managing virtual machine execution environments for grid computing, in: *Proc. the 2004 ACM/IEEE conference on Supercomputing*, (July 2004).

[21] VMware Inc., VMware VirtualCenter user's manual, http://www.vmware.com/pdf/VC_Users_Manual_11.pdf, 13th Apr. 2004.

[22] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. Lam and M. Rosenblum, Optimizing the migration of virtual computers, in: *Proc. the 5th Symposium on Operating Systems Design and Implementation* (2002).

[23] A. Zeitoun, Z. Wang and S. Jamin, RTTometer: Measuring path minimum rtt with confidence, *IEEE Workshop on IP Operations and Management* (2003).

[24] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham and M.J. West, Scale and performance of a distributed file system, ACM Transactions on Computer Systems 6(1) (1988) 51–81.

[25] VMware Inc., GSX Server 2.5.1 user's manual, http://www.vmware.com/pdf/gsx251vm_manual.pdf.

[26] B. White, A. Grimshaw and A. Nguyen-Tuong, Grid-based file access: The legion I/O model, in: *Proc. the 9th IEEE International Symposium on High Performance Distributed Computing* (Aug. 2000) pp. 165–173.

[27] M. Litzkow, M. Livny and M.W. Mutka, Condor: A hunter of idle workstations, in: *Proc. the 8th International Conference on Distributed Computing Systems* (June 1988) pp. 104–111.

[28] D. Thain, J. Basney, S-C. Son and M. Livny, The kangaroo approach to data movement on the grid, in: *Proc. the 10th IEEE International Symposium on High Performance Distributed Computing* (Aug 2001) pp. 325–333.

[29] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. Arpaci-Dusseau, R. Arpaci-Dusseau and 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).

[30] 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* (Aug. 1998) pp. 207–233.

[31] D. Thain and M. Livny, Parrot: Transparent user-level middleware for data-intensive computing, in: *Proc. Workshop on Adaptive Grid Middleware*, New Orleans, Louisiana (September 2003).

[32] D. Mazieres, M. Kaminsky, M. Kaashoek and E. Witchel, Separating key management from file system security, in: *Proc. the 17th ACM Symposium on Operating System Principles*, (Dec. 1999).

[33] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson and R. Thurlow, The NFS version 4 protocol, in: *Proc. the 2nd International System Administration and Networking Conference* (May 2000).

[34] R. Chandra, N. Zeldovich, C. Sapuntzakis and M.S. Lam, The collective: A cache-based system management architecture, in: *Proc. the 2nd Symposium on Networked Systems Design and Implementation* (May 2005) pp. 259–272,.

[35] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt and A. Warfield, Live migration of virtual machines, in: *Proc. the 2nd Symposium on Networked Systems Design and Implementation* (May 2005).

Ming Zhao is a PhD candidate in the department of Electrical and Computer Engineering and a member of the Advance Computing and Information Systems Laboratory, at University of Florida. He received the degrees of BE and ME from Tsinghua University. His research interests are in the areas of computer architecture, operating systems and distributed computing.
E-mail: ming@acis.ufl.edu

Jian Zhang is a PhD student in the Department of Electrical and Computer Engineering at University of Florida and a member of the Advance Computing and Information Systems Laboratory (ACIS). Her research interest is in virtual machines and Grid computing. She is a member of the IEEE and the ACM.
E-mail: jzhang@ecel.ufl.edu

Renato J. Figueiredo received the B.S. and M.S. degrees in Electrical Engineering from the Universidade de Campinas in 1994 and 1995, respectively, and the Ph.D. degree in Electrical and Computer Engineering from Purdue University in 2001. From 2001 until 2002 he was on the faculty of the School of Electrical and Computer Engineering of Northwestern University at Evanston, Illinois. In 2002 he joined the Department of Electrical and Computer Engineering of the University of Florida as an Assistant Professor. His research interests are in the areas of computer architecture, operating systems, and distributed systems.
E-mail: renato@acis.ufl.edu