Distributed File System Support for Virtual Machines in Grid Computing

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

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 access to VM state for unmodified VM monitors; 2) it supports user-level and write-back disk caches, per-application caching policies and middleware-driven consistency models; and 3) it supports the use of meta-data associated with files to expedite data transfers. The paper reports on its performance in a WAN setup using VMware-based VMs. Results show that the solution delivers performance over 30% better than native NFS and can bring application-perceived overheads below 10% relatively to a local disk setup. The solution also allows a VM with 1.6GB virtual disk and 320MB virtual memory to be cloned within 160 seconds when it is first instantiated (and within 25 seconds for subsequent clones).

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 transfer of VM state across domains can be implemented by means of extensions to a proxy-based distributed Grid virtual file system [5]. (In the context of the paper, VMs are referred as "classic" instruction-set VMs as defined in [3])

The architecture leverages existing implementations of a de-facto distributed file system standard for local-area networks (NFS [6]) and extends it at the user level to support 1) middleware-controlled encrypted file system channels and cross-domain authentication, 2) network latency hiding through client-side caching, and 3) meta-data at the file system level to efficiently handle VM memory state files. These mechanisms are implemented without requiring modifications to existing NFS clients and servers, and support the execution of unmodified application binaries – including para-virtualized VMs and user-mode VMs, which use file system to store machine state (e.g. VMWare hosted I/O [7], UML [8],).

This paper also reports on the performance of VMware-based VMs instantiated from state stored in wide-area distributed file systems – both conventional and proxy-enhanced NFS. Experimental results show that the proxy-enhanced file system improves the execution time of applications and experiences a relative small overhead with respect to locally stored VM state. Finally, results show that the use of on-demand transfers and meta-data information allows instantiation of a 320MB-RAM/1.6GB-disk Linux VM clone in less than 160 seconds for the first clone (and about 25 seconds for subsequent clones), considerably outperforming cloning based on transfer of entire files (in excess of 1100 seconds).

The contribution of this paper is a novel solution that extends user-level proxies to support on-demand, high-performance transfers for Grid VMs. It builds on and extends upon a distributed virtual file system infrastructure that provides a basis for establishing per-session, Grid-wide file system sessions. The solution 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 (write-back) disk caches. In addition, the solution supports application-driven meta-data information to allow clients

to satisfy requests using on-demand block-based or file-based transfers selectively. It does so in a manner that is transparent to the kernel (and to applications). Hence, it is not specific to a particular VM technology, and supports existing hosted VMs that allow an NFS file system to store machine state in regular files/filesystems. The paper also analyzes the performance of this solution quantitatively in a wide-area network environment, and demonstrates that it can outperform unmodified NFS and SCP-based file copying, in both VM instantiation through cloning and run-time execution.

The rest of this paper is organized as follows. Section 2 introduces an architecture for grid computing on VMs and discusses alternatives for handling transfer of VM state under this model. Section 3 describes distributed virtual file system techniques for supporting VMs, and Section 4 presents results and discussions on the performance of this solution. Section 5 discusses related work, and Section 6 concludes the paper.

2. Grid Computing Using Virtual Machines

A Grid computing system that supports unmodified applications (such as commercial applications for which source code access is not available) faces the challenge of preserving the integrity of resources in the presence of untrusted users and/or applications. Considerations of resource security, user isolation, legacy applications and flexibility in the customization of execution environments have led to architectures that employ "classic" VMs for Grid computing [4] to support problem-solving environments. A flexible, application-centric solution can be built based on the fact that, once defined, a VM execution environment can be encapsulated, archived by middleware and then be made available to users. Upon request, such a VM can be "cloned" and instantiated by middleware to exploit the computing power of distributed Grid resources. "Cloning" of a VM entails copying its states from a "golden" VM, configuring it with user specific information and then restoring it for the Grid user.

Mechanisms 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). Hence, data management is the 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 image; without support for 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 (Figure 1), the components of a Grid VM session can be distributed across three different logical entities: the "image server", which stores VM base configuration images; the "compute server", which provides the capability of instantiating VMs; and the "data server", which stores user data.

Support for VMs that can be dynamically instantiated across Grid resources poses challenges to the data management infrastructure. For various VM technologies, such as hosted I/O VMWare [7], User-Mode Linux [8], and Xen [9], the image of a VM can be represented by memory/disk state files or filesystems that are often large (GBytes) and must be transferred efficiently from an image server to a compute server (Figure 1). Two approaches are conceivable: one is to transfer the entire VM state before instantiation; another is to leverage host O/S support for on-demand transfer of file system data to allow the VM state transferred on demand as requested by the VM monitor. Full-state transfers have the disadvantages of large setup latencies, and transfer/storage of unnecessary data on the compute server – typical VM sessions only reference a small fraction of the disk state during execution [10]. On-demand transfers are possible at the granularity of file blocks in distributed file systems

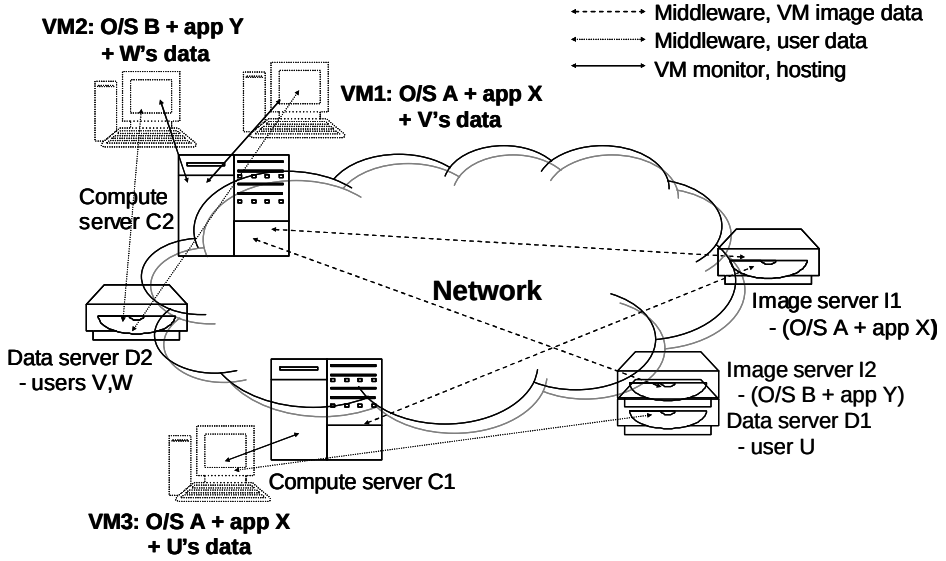

Figure 1: Middleware supported data management for both virtual machine images and user file systems allows for application-tailored VM instantiation (VM1, VM2, VM3) across Grid resources (compute servers C1/C2, image servers I1/I2, data servers D1/D2).

such as NFS, but the network latency of accessing a server can lead to poor performance for writes, and if capacity misses occur frequently in memory file system buffers. The techniques presented in Section 3 describe a data management infrastructure that supports efficient on-demand transfer of data associated with a VM's image.

In combination with complementary middleware for resource and user management, and for user interfaces, the availability of Grid VMs enables the design of problem-solving systems that are highly flexible. For example, the In-VIGO[1] [11] system allows on-demand creation of a per-user "virtual workspace" that provides a Web browser-capable interactive graphical interface to a Linux-based VM clone, as well as an upload/download facility through a file manager that runs within the VM. The workspace is dynamically built by the middleware in a user-transparent manner by cloning a suspended VM image and configuring the clone with user-specific information, and by mounting the user's Grid virtual file system inside the VM clone.

3. Grid Virtual File System for Virtual Machines

3.1. 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 [12]), or transfer entire files at the time they are opened (e.g. GASS [13]), which may lead to unnecessary data transfers (e.g. of an entire VM image, even when only a fraction of it is required for computation). Data management solutions supporting on-demand transfer for Grids have also been investigated in related work, as discussed in Section 5. However, these solutions require customized application libraries and/or file servers.

Previous work has shown that a data management model supporting on-demand data transfers without requiring dynamically-linked libraries or changes to native O/S file system clients and servers can be achieved by way of two mechanisms – logical user accounts [14] and a distributed virtual file system [5]. Such a distributed virtual file system can be built through the use of virtualization layer on top of NFS, a de-facto distributed file system standard, allowing data to be transferred on-demand between Grid storage and compute servers for the duration of a computing session. This functionality is realized via extensions to existing NFS implementations that are at user-level, requiring neither modification of O/S clients and servers, nor of applications. The resulting Grid virtual file system (GVFS) utilizes user level proxies to dynamically map between short-lived user identities allocated by middleware on behalf of a user [15] Furthermore, data transfer in GVFS is on demand and transparent to the user.

By supporting unmodified applications, GVFS can inherently support implementations of different flavors of VM technologies, including commercial and open-source designs such as VMware, UML and Xen. However, because VM state data are often large in size and remote accesses often have high latencies, extensions to the Grid virtual file system are necessary to improve its performance in this environment. The extensions proposed in this paper are illustrated in Figure 2 and described in the remaining of this section. The extensions are oriented towards file system sessions established for VM transfers, but are generally applicable and not tied to any particular VM implementation.

3.2. Extensions to support VM transfer

3.2.1. Disk-based file system caches.

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 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]. Complementing the memory cache with a disk cache can form an effective cache hierarchy. There are implementations of distributed file systems that exploit these advantages, for example, AFS 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 extended to employ client-side proxy managed disk cache in a unique way, through user-level proxies that can be customized on a per-user or per-application basis. For instance, cache size and write policy can be optimized according to the knowledge of a Grid application. A more concrete example is enabling file-based disk caching by meta-data handling and application-tailored knowledge to support heterogeneous disk caching (Section 3.2.2). The proxy cache can be deployed in systems which do not have native kernel support for disk caching, e.g. Linux. Because the proxy behaves both as a server (receiving RPC calls) and a client

[1] The In-VIGO prototype can be accessed from http://invigo.acis.ufl.edu; courtesy accounts are available.

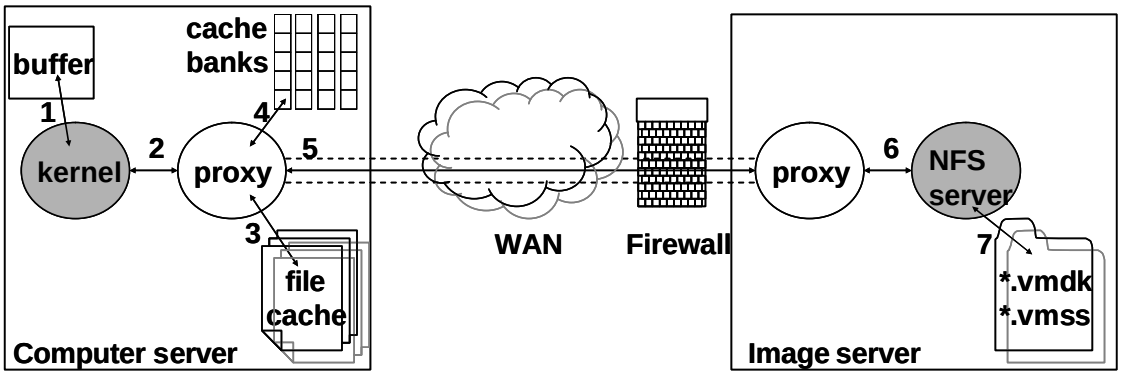

Figure 2: Proxy extensions for VM image transfers. At the compute server, the VM monitor issues system calls that are processed by the NFS client. Requests may hit in the memory file systems buffer (1); those that miss are processed by the user-level proxy (2). At the proxy, requests that hit in the 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), then to the kernel server (6), and finally satisfied from data on the server (7).

(issuing RPC calls), it is possible to establish a virtual file system by forwarding along a chain of multiple proxies. Thus in addition to the server-side proxy (responsible for authenticating requests and mapping identities), another proxy can be started at the client-side to establish and manage disk caches, as illustrated in Figure 2. Furthermore, a series of proxies, with independent caches of different sizes, can be cascaded between client and server, supporting scalability to a multi-level cache hierarchy.

Disk caching in GVFS is implemented by the file system proxy and operates at the granularity of NFS RPC calls [18]. 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. Caches of different proxies can be independently managed; they may be configured with different sizes, associativities, as well as data block sizes (up to the NFS protocol limit of 32KB). The design also allows for different proxies to share disk caches for read-only data.

The GVFS proxy disk cache also supports the write-back policy for write operations, an important feature in wide-area environments to hide long write latencies. Write-back caching can be applied to VMs in Grid computing in different ways. It supports write-back of persistent virtual disks that transparently complements kernel-level buffering and application-level write-back schemes with high-capacity storage. It can also support write-back of redo logs for non-persistent disks of VMs that may be migrated across Grid resources.

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. This model of middleware-driven consistency is assumed in this paper; it is sufficient to support many Grid applications, e.g. when tasks are known to be independent by a scheduler for high-throughput computing (e.g. as in Condor [19]).

3.2.2. 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. When the proxy receives a NFS request to a file which has meta-data associated with, it processes the meta-data and takes the described 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. The meta-data contain the data characteristics of the file it is associated with, and define a sequence of actions which should be taken on the file when it is accessed.

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, the memory state file 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, request only non-zero blocks from the server, then reconstruct the whole memory state and present it to the VM monitor. Normally the memory state contains many zero-filled blocks [9] that can be filtered by this technique, and the traffic on the wire can be greatly reduced while instantiating a VM. For instance, when resuming a 512MB-RAM RedHat 7.3 VM which is suspended in the post-boot state, the client issues 65,750 NFS reads while 60452 of them can be filtered out by the above 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 4K to 32Kbytes in size), 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 virtual disk of a "classic" VM are typically restricted to a working set that is much smaller (<10%) than the large virtual disk file [20][9]. But when large files are indeed completely required by client 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", "remote 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 server, the proxy will: 1) compress the file on the server (e.g. using GZIP); 2) remote 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). The file cache can also support write-back, which includes similar steps of compressing, uploading and uncompressing.

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 caching scheme. 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 image 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 the transfer of the memory state file.

3.2.3. Support for persistent and non-persistent Grid VMs

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 image 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 image 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 cache can exploit locality of references to the virtual disk and provide high-bandwidth, low-latency access to cached file blocks; and 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 image server stores a number of non-persistent VMs for the purpose of "cloning". These generic images have application-tailored hardware and software configurations, and when a VM is requested from a compute server, the image server is searched against the requirements of the desired VM. The best match is returned as the "golden" image, which is then "cloned" to the compute server. The cloning process entails copying the state from the "golden" image, 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 VM state from image server to compute server, and also efficient writes to redo logs for checkpointing.

Similar to the first scenario, the extended GVFS can quickly instantiate a cloned VM by meta-data handling for memory state file and on-demand block-based access to virtual disk files. Instead of copying the entire virtual disk, only symbolic links are made to the virtual disk files on 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 images can be used to instantiate many clones, e.g. for concurrent execution of a high-throughput task. The proxy disk cache can exploit temporal locality among cloned instances and accelerate the cloning process. On the compute server, the cached data of memory state and virtual disk from previous clones can greatly expedite new clonings from the same "golden" images. And a second-level proxy cache can be setup on a LAN server, as explained in Section 3.2.1, to further exploit the locality and provide high speed access to the state of golden images for clonings to compute servers in the same local network.

4. Performance

4.1. 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 [21]. The core proxy code has been extended to support private data channels [22], client-side disk caching and meta-data handling. This section evaluates the performance for supporting VM in the Grids by analyzing the data from experiments of a group of benchmarks.

Experiments are conducted in both local-area and wide-area environments. The LAN image server is a dual-processor 1.8GHz Pentium III cluster node with 1GB of RAM and 576GB of disk storage. The WAN image server is a dual-processor 1GHz Pentium-III cluster node with 1GB RAM and 45GB disk. In the experiments on application execution (Section 4.2), the compute server is a 1.1GHz Pentium-III cluster node with 1GB of main memory and 18GB of SCSI disk; in the experiments on VM cloning (Section 4.3), the compute servers are cluster nodes which have four 2.4GHz Xeon processors with 1GB RAM and 18GB disk each. The compute servers are installed VMware GSX server 2.5 to support x86-based VMs. They are connected with the LAN image server in a 100Mbit/s Ethernet at the University of Florida, and connected with the WAN image server through Abilene between Northwestern University and University of Florida. The proxy cache is configured with 512 file banks which are 16-way associative, and has a capacity of 8GBytes, but typically only 1~3GBtyes of cache has been actually used in the experiments.

4.2. Experiments on application execution

4.2.1. Benchmarks and scenarios

Three benchmarks are selected to evaluate the performance of run-time execution of typical applications:

SPECseis96 is taken from the SPEC high-performance group benchmarks. 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 benchmark is 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 generated by the "patch" command.

It is worth to be emphasized that a VM-based Grid execution environment allows users to customize an execution environment for an application, encapsulate its virtual state and replicate it across distributed resources by means of dynamic instantiation of VM copies. Users of a VM-based Grid can be provided with an interactive environment to customize their VMs, and the middleware is capable of archiving, replicating and instantiating such environments on any available physical resource capable of supporting VMs. In this environment, it is important that interactive sessions for VM setup show good response times to the Grid user. Thus, the Latex benchmark is chosen to study this scenario.

Kernel compilation represents file system usage in a software development environment, similar to the Andrew benchmark [23]. The kernel is a Red Hat Linux 2.4.18, 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.

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

Local: The VM state is stored in a local disk file system.

LAN: The VM state is stored in a directory NFS-mounted from the LAN image server. Data access is forwarded by GVFS proxies via SSH tunnels.

WAN: The VM state is stored in a directory NFS-mounted from the WAN image server. Data access is forwarded by GVFS proxies via SSH tunnels.

WAN+C: The setup is the same as the WAN scenario except that client-side proxy disk caching is enabled.

4.2.2. Results and analysis

The experiments are initially setup with "cold" caches (both kernel buffer cache and proxy disk cache) by un-mounting and mounting the virtual file system, and flushing the proxy caches before an execution. Figure 3 shows the execution times for the four phases of the SPECseis. 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 then WAN and WAN+C 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 the total execution time down 33 percent in the wide-area environment.

The LaTeX benchmark results in Figure 4 show that in wide-area environment interactive users would experience a startup latency of 225.67 seconds (WAN), or 217.33 seconds (WAN+C). 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 would have to be downloaded from the image 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 scenario to about 20 seconds. The proxy disk cache can further improve the average response time for WAN+C scenario to very close to that of Local (8% slower) and LAN (6% slower) scenarios, but 54% faster than that of non-cached WAN scenario. The time needed to flush cached dirty blocks if write-back is enabled is about 160 seconds, which is also much shorter than the uploading time (4633 seconds) of the entire state.

Experimental results from the kernel compilation benchmark are illustrated in Figure 5. The first run of the benchmark in WAN+C scenario which begins with "cold" caches shows an 84% overhead compared to that of 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 LAN scenario, it is less than 4% slower. The availability of the proxy cache allows WAN+C to outperform WAN 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 re-used across iterations (e.g. in application development environments), the response times of the WAN-mounted virtual file system are acceptable.

4.3. Experiments on virtual machine cloning

4.3.1. Benchmark and scenarios

Another benchmark is designed to investigate the performance of GVFS' support for cloning VMs. The cloning scheme is as discussed in Section 3.2.3, which includes copying the VM configuration file, copying the VM memory state file, building symbolic links to the virtual disk files, configuring the cloned VM, and at last resume the new VM. The execution time of the benchmark is also measured in five different scenarios:

Local: The VM images are stored in a local disk file system.

WAN-S1: The VM images are stored in a directory NFS-mounted from the WAN image server. During the experiment, a single VM image is cloned eight times to the compute server sequentially. The clonings are supported by GVFS with all extensions, including private data channels, 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 images are each cloned to the computer server once sequentially. It is designed to evaluate the performance when there is no locality among clonings.

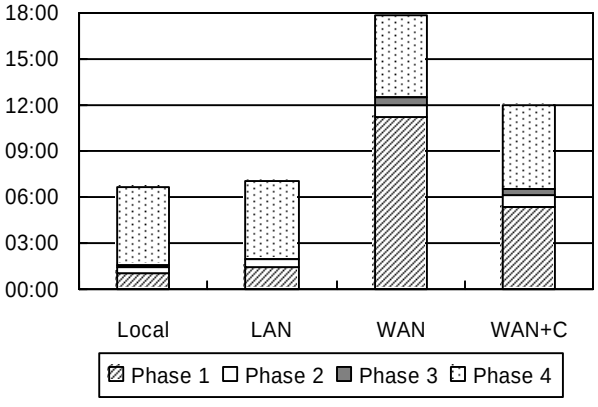

Figure 3: SPECseis benchmark execution times (minutes:seconds). The results show times for each execution phase.

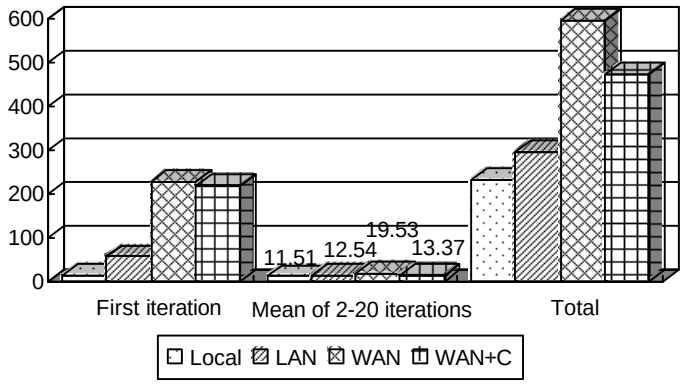

Figure 4: LaTeX benchmark execution times (seconds). The execution times of the first iteration, the average execution times of the following 19 iterations, and the total execution times are listed.

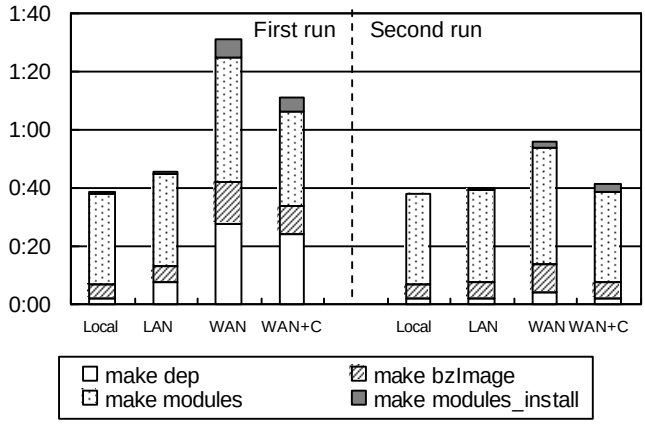

Figure 5: Kernel compilation benchmark execution times (hours:minutes). The results show times for four different phases. Results have been colleted for two consecutive runs of the benchmark; in the first run, buffer and proxy cache are "cold", while in the second run they are "warm".

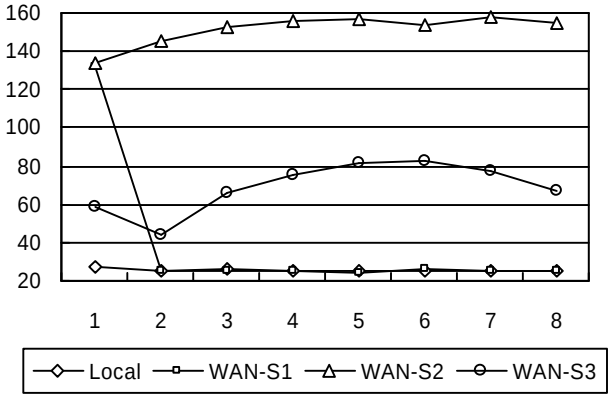

Figure 6: VM cloning times (seconds) for a sequence of images (from 1 to 8) with 320MB of memory and 1.6GB of virtual disk. It takes 1127 seconds if copying an image entirely by SCP. If only copying the memory state and accessing the virtual disk from a directory mounted from the image server without GVFS' support, it takes 2060 seconds to clone a VM.

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 images 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 images are stored in a directory NFS-mounted from the WAN image server to eight computer servers, which are eight nodes of a cluster. During the experiment, eight VM images are cloned to the compute servers in parallel. The clonings are supported by GVFS with all extensions.

4.3.2. Results and analysis

Figure 6 shows the cloning times for a sequence of VM images which have 320MB of memory and 1.6GB of virtual disk. In comparison with the range of GVFS-based cloning times shown in these figures, if the VM is cloned using SCP for full file copying, it takes approximately twenty minutes to transfer the entire image. If the VM state is not copied but read from a pure NFS-mounted directory, the cloning takes 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 memory state and virtual disk files among the clones, the proposed solution even allows 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 1 compares sequential cloning with parallel cloning. In the experiment on WAN-P scenario, the eight compute servers share a single image server and server-side GVFS proxy. But when the eight clonings start in parallel, each client-side GVFS proxy on every compute server 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. In either scenarios, the support from GVFS is on-demand, and transparent to user and VM monitor. And, as demonstrated in Section 4.2, following a machine's instantiation via cloning, GVFS can also improve the VM's run-time performance substantially.

Table 1: Total time of cloning eight VM images 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

5. Related Work

Data management solutions such as GridFTP [12] and GASS [13] provide APIs upon which applications can be programmed to access data on the Grid. Legion [24] employs a modified NFS server to provide access to a remote file system. The Condor system and Kangaroo utilize remote I/O mechanisms implemented via bypass mechanisms that rely on system call trapping by dynamic library linking to allow applications to access files [19] [25]; a library-based approach to the movement of VM images is also taken in [9]. However, library-based approaches would not work with statically linked software. NeST [26] is a software Grid storage appliance that supports the NFS protocol, among others; however 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, the solution of this paper allows unmodified applications to access Grid data using conventional operating system clients/servers.

The self-certifying file system (SFS [27]) is another example of a file system that uses proxies that forward NFS protocol calls and implement cross-domain authentication and encryption. The approach of this paper differs from SFS in several ways. A key difference is that the approach of this paper 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.

The NFS V4 [28] protocol includes provisions for aggressive caching. However, V4 implementations have not been deployed in Grid setups; implementations of versions 2 and 3 of the protocol are available for a wide variety of platforms.

A related project has investigated solutions that improve the performance of the migration of classic VMs [10]. Their work focuses on mechanisms to transfer images of virtual desktops, possibly across low-bandwidth links. Common between their approach and this paper are mechanisms for on-demand block transfers, and optimizations based on the observation that zero-filled blocks are common in suspended VM memory images. A key difference lies in fact that the techniques of this paper are independent from applications and are implemented through the interception of NFS/RPC calls and reusing O/S clients and servers available in typical Grid resources, while their approach uses modified libraries as a means of intercepting VMM accesses to files and employs a customized protocol.

6. Conclusions and future work

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 that 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 monitor allows for state to be stored in file systems that can be mounted via NFS.

Results show that user-level proxy caches improve upon the performance of conventional NFS over a wide-area network. Results also show that, with "warm" caches, the enhanced file system leverages native O/S support for buffer caches and has small overhead when compared to a local-disk file system. Finally, results show that the use of on-demand transfers and meta-data information allows instantiation of a 320MB-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 cloning based on transfer of entire files, and on non-enhanced NFS.

Directions for future work include distributed virtual file system support for efficient checkpointing and migration of VM instances for load-balancing and fault-tolerant execution, and dynamic profiling of application data access behavior to support pre-fetching and high-bandwidth transfers of large data blocks in a selective manner, using protocols such as GridFTP for inter-proxy transfers.

Acknowledgements

Effort sponsored by the National Science Foundation under grants EIA-0224442, EEC-0228390, ACI-0219925 and NSF Middleware Initiative (NMI) collaborative grant ANI-0301108. The authors also acknowledge a gift from VMware Corporation and a SUR grant 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, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations". International J. Supercomputer Applications, 15(3), 2001.

[2] A. Butt, S. Adabala, N. Kapadia, R. Figueiredo, J. Fortes, "Grid-computing Portals and Security Issues", Journal of Parallel and Distributed Computing (JPDC), 63(10), pp. 1006-1014, 2003.

[3] Robert, P. Goldberg. "Survey of virtual machine research". IEEE Computer Magazine, 7(6):34-45, 1974.

[4] R. Figueiredo, P. A. Dinda, J. A. B. Fortes, “A Case for Grid Computing on Virtual Machines”, Proc. International Conference on Distributed Computing Systems (ICDCS), May 2003.

[5] R. Figueiredo, N. Kapadia and J. A. B. Fortes. "The PUNCH Virtual File System: Seamless Access to Decentralized Storage Services in a Computational Grid", Proc. IEEE International Symposium on High Performance Distributed Computing (HPDC), August 2001.

[6] B. Pawlowski, C Juszczak, P. Staubach, C. Smith, D. Lebel and D. Hitz, “NFS Version 3 Design and Implementation”, 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”, Proceedings of the USENIX Annual Technical Conference, June 2001.

[8] J. Dike, “A User-mode Port of the Linux Kernel”, Proc. 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”, Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), October 2003.

[10] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. Lam and M. Rosenblum, “Optimizing the Migration of Virtual Computers”, Proceedings of the 5th Symposium on Operating Systems Design and Implementation, 2002.

[11] 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”, to appear, Future Generation Computing Systems, special issue, Complex Problem-Solving Environments for Grid Computing, David Walker and Elias Houstis, Editors.

[12] 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.

[13] 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.

[14] 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.

[15] S. Adabala, A. Matsunaga, M. Tsugawa, R. Figueiredo and J. Fortes, “Single Sign-On in In-VIGO: Role-based Access via Delegation Mechanisms Using Short-lived User Identities”, to appear, Proc. of the International Parallel and Distributed Processing Symposium (IPDPS), April 2004.

[16] B. Callaghan, “NFS Illustrated”, Addison-Wesley (2000).

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

[18] M. Zhao, “Proxy Managed Disk Cache for Grid Virtual File System”. In Technical Report TR-ACIS-04-001, ACIS Laboratory, Department of Electrical and Computer Engineering, University of Florida, 05/2004.

[19] M. Litzkow, M. Livny and M. W. Mutka, “Condor: a Hunter of Idle Workstations”, Proc. 8th Int. Conf. on Distributed Computing Systems, pp104-111, June 1988.

[20] E. Deelman et al., “GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists”, Proceedings of High Performance Distributed Computing (HPDC), 2003.

[21] N. Kapadia, J. Fortes, “PUNCH: An Architecture for Web-Enabled Wide-Area Network-Computing”, Cluster Computing: the Journal of Networks, Software Tools and Applications, 2(2), 153-164 (Sept. 1999).

[22] 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.

[23] 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):51-81, February 1988.

[24] 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.

[25] 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.

[26] 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”, Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing, Edinburgh, Scotland, July 2002.

[27] D. Mazieres, M. Kaminsky, M. Kaashoek, E. Witchel, “Separating Key Management from File System Security”, Proc. 17th ACM Symposium on Operating System Principles (SOSP), Dec 1999.

[28] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson and R. Thurlow, “The NFS Version 4 Protocol”, Proc. 2nd Intl. System Administration and Networking (SANE) Conference, May 2000.