University of Alberta

VIRTUAL APPLICATION APPLIANCES ON CLUSTERS

by

Erkan Unal

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Spring 2010
Edmonton, Alberta

Examining Committee

Paul Lu, Computing Science

Alex Brown, Chemistry

Martin Jägersand, Computing Science

To My Parents
For always loving and supporting me.

Abstract

Variations between the software environments (e.g., installed applications, versions of libraries) on different high-performance computing (HPC) systems lead to a heterogeneity problem. Therefore, we design an optimized, homogeneous virtual machine (VM) called a virtual application appliance (VAA). Scientists can package scientific applications, and all supporting software components, as VAAs and run them independently from the underlying heterogeneous HPC systems. However, securely moving data in and out of the VAA and controlling the execution of applications are not trivial for a non-computer scientist. Consequently, we develop two automated stage-in/stage-out secure data movement mechanisms. We also explore a migration mechanism to further simplify the control of the VAA execution.

Empirical evaluation results show that VAAs achieve near-native performance in widely used bioinformatics applications that we tested. Data movement, VM boot up, shutdown and migration overheads of VAAs are negligible with respect to total run-times.

Acknowledgements

I would like to thank my supervisor Paul Lu for his guidance, patience and continuous support during my studies. Most importantly, he taught me how to be a good scientist. I must thank the Trellis research group members Cam Macdonell, Jordan Patterson and Jeremy Nickurak for their help when I had some problems with my research. Mark Berjanskii from the Prion research group also deserves to be acknowledged for sharing his knowledge and providing GROMACS data and his Python scripts for the evaluation part of my thesis.

My “buddy” Anum Usman (aka “Anum the Eks”). Thank you for your great friendship and all the fun moments.

Finally, I would like to thank my mom and dad for their unconditional love and support. I am so lucky to have them.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

BBS	Bioinformatics Benchmark System
CPU	Central Processing Unit
CSM	Copy over Shared Memory
FCS	Forced Command Script
GCC	GNU Compiler Collection
HMM	Hidden Markov Models
HPC	High Performance Computing
KVM	Kernel Virtual Machine
MAC	Media Access Control
NIC	Network Interface Card
OS	Operating System
PBS	Portable Batch System
SCN	Secure Copy over Network
SSH	Secure Shell
VA	Virtual Appliance
VAA	Virtual Application Appliances
VD	Virtual Disk
VDE	Virtual Distributed Ethernet
VDR	Virtual Disk Repository
VM	Virtual Machine
VMES	VM Execution Script
VMM	Virtual Machine Monitor
WAN	Wide Area Network
WHS	Wrapper Host Script

Chapter 1

Introduction

Scientists need to use high performance computing (HPC) centers extensively. From biology to physics, their research may involve complex and long running computations and/or simulations. Processing time of scientific applications may take years in a limited low-budget workstation configuration depending on the characteristics of data. Universities are investing heavily to build HPC centers (e.g. clusters) with high computation power to get results faster for scientific applications. Also, projects like Trellis [39] are looking for ways to aggregate different HPC centers to increase the computational power. The Trellis group, with the help of computational biologists and chemists, had performed the CISS-1 and CISS-2 experiments for this purpose [38]. As long as scientists need to run more computations and simulations in a relatively short time, demand for HPC centers will increase.

However, in most cases, adapting the scientist's work to an HPC center is not a straightforward task due to incompatibilities between the software systems that the scientist uses to prepare her work and the ones on the HPC center. Operating systems (OSes), library versions and even security infrastructures may be incompatible with some of the software systems that are intended to be installed or executed on them. Problems due to incompatible software systems can delay the scientific work or limit the scientist with a specific software environment.

We can clarify the problems caused by incompatible software resources by explaining how a scientist uses the cluster from the preparation of her work to the execution. The scientist starts to prepare her data and data analysis components beforehand. One common way is to use her workstation for the preparation. It is convenient because the scientist has full control of her system, however, she has restricted access to the cluster environment. She needs to prepare programs, scripts and application-specific configuration files for her own purpose in her workstation. Then, she debugs and tests all the details of the components with different sets of sample data.

The preparation takes several steps. Let us consider a biologist who uses a molecular dynamics application called GROMACS version 3.2.1 [7]. GROMACS works on the UNIX systems; therefore, the first step is the installation of the application to her workstation by including necessary external libraries. The installation can be done by herself or by an experienced third person. Then,

she needs to decide the requirements of her work. She probably has to prepare a script for the execution of GROMACS and data analysis. She can use a scripting language such as Python [45] or Perl [28]. In our example, let us say it is Python. After debugging, corrections and tests, when she is satisfied with the performance of the components, she is ready to deploy her work to the cluster.

During the deployment phase, she may not find the suitable execution environment for her work. The GROMACS version that her work is prepared on may not be available or different than 3.2.1 (Figure 1.1). She has to ask the system administrator to install the specific version that she needs. A possible answer would be that it is not possible due to other people's job dependency on the currently installed GROMACS version. However, this problem can be solved by returning to the preparation phase and adjusting her work to the GROMACS version on the cluster. However, this reworking process may cause significant delays. Similar compatibility problems may apply to Python and any other third party dependencies of her scripts. Therefore, the reworking process becomes more complicated.

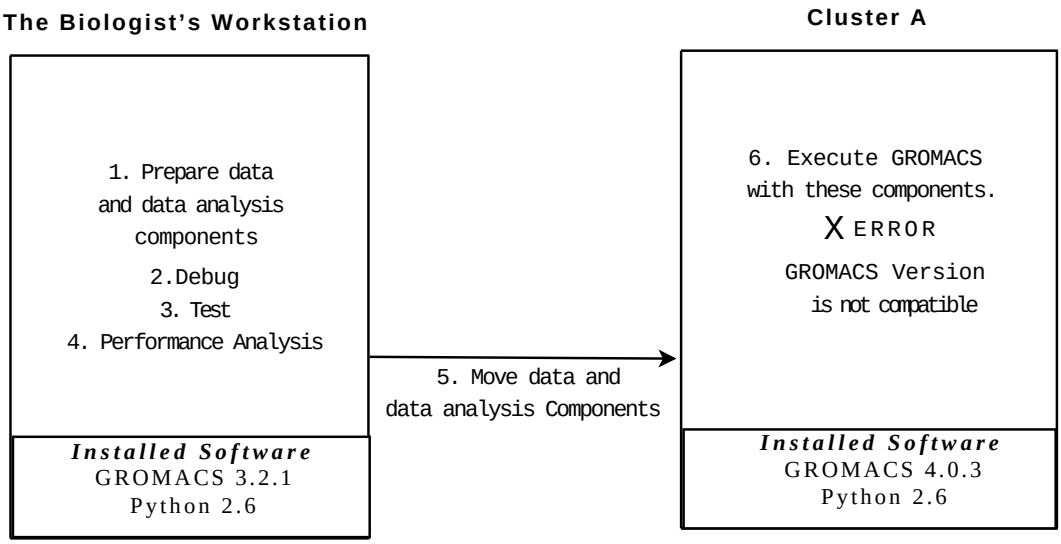


Figure 1.1: A problem scenario in one cluster

Deployment problems increase if she wants to use other clusters for more computational power (Figure 1.2). In addition to the single cluster problems, she has to consider where to store the input and output files on each cluster. The convenient placement is one single location for all the output files and spread out the input files across the clusters. This placement simplifies data analysis. However, she has to do extra work for the placement of data or use a cross-domain file system like TrellisNFS [11]. Even if a cross domain file system is available, as the Figure 1.2 shows that the scientist can execute her work only on Cluster C. Therefore, she cannot use the computational power of other two clusters due to GROMACS version incompatibility on Cluster A and lack of GROMACS software on Cluster B.

One well-known way is to use Virtual Machines (VMs) to solve the above-mentioned problems.

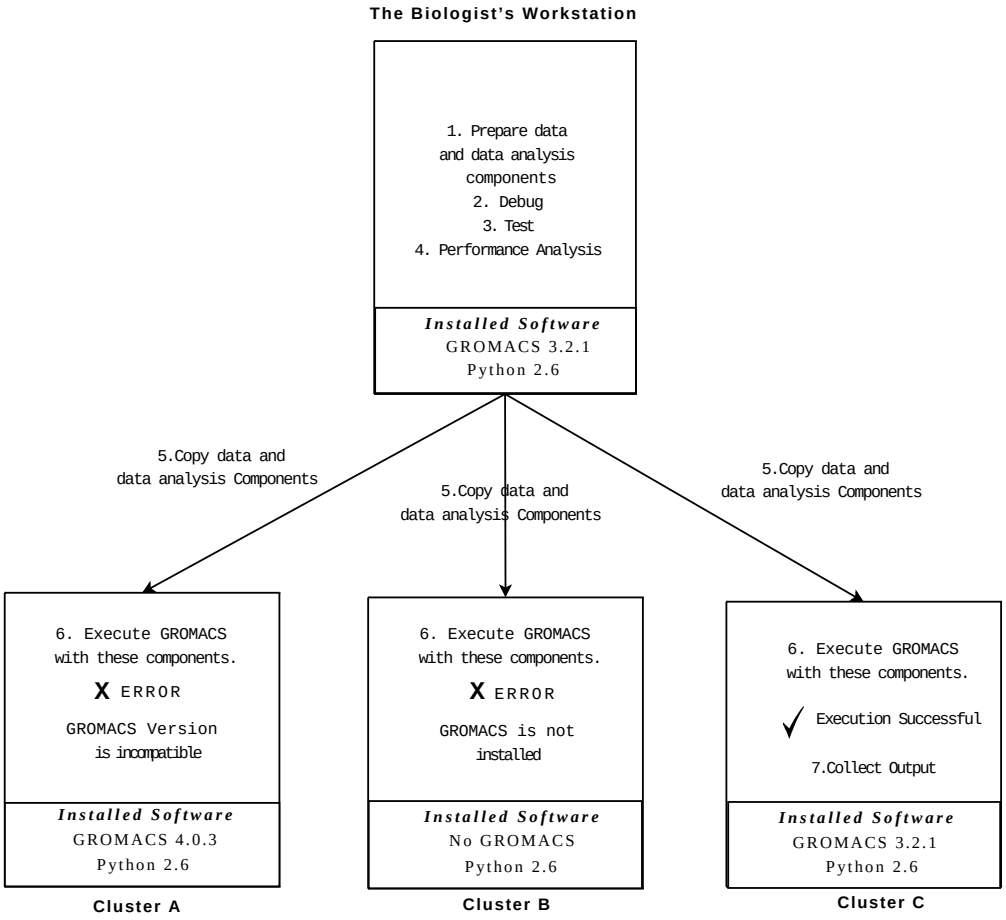


Figure 1.2: A problem scenario in multiple clusters

A VM is an efficient solution in HPC to address the heterogeneity problem for the deployment of software systems. A VM can be deployed such that software is executed inside a VM even if the software is incompatible with the underlying system (i.e. the host system). One can easily create a VM and put every necessary requirement of the software inside a VM image. These components include an operating system (OS), libraries, application specific configuration files and virtual versions of its hardware resources. Hence, a VM creates a fully functional environment for the software without any change to the underlying system. The only requirement of the VM solution is a single application installed in the underlying system to execute VMs. This type of application is called virtual machine monitor (VMM) or hypervisor.

In Figure 1.3, we can see the VM solution to the problems in Figure 1.2. In Cluster A, the VM solves the version incompatibility problem. In Cluster B, the VM can exeute the user's job even if there is no GROMACS software present on the underlying system. Cluster B also shows that the only requirement is an application such as KVM [25] that executes the VM. Consequently, without the VM, the user has to limit herself to Cluster C but with the VM, she can use the computational power of other clusters too.

Apart from addressing the heterogeneity problem, HPC applications can also benefit from VM migration. In this dissertation, we examine VM migration as a recovery mechanism after failovers and workaround for the cluster's wall-time limit. VMMs can save the VM states at regular intervals. Later, VMMs can use these saved states to return to the previous state of the execution in case a failure happens. This technique is known as application checkpointing. Also, the batch scheduler terminates the job after the wall-time has run out to prevent the user or user's job from monopolizing the cluster. Therefore, in case the user's job needs more time to complete the execution, the VM state can be saved just before the wall-time has run out and restored from this saved state by resubmitting the user's job to the batch scheduler again.

Although VMs have all of these benefits, they introduce two main problems [32]. First, most of the contemporary VM products can run only on x86 systems. Since x86 systems are ubiquitous, this limitation is not a big problem. The second and most important problem is that applications run slower when running on just an OS (host OS) and hardware because a VMM puts another software layer between the OS and hardware.

To maximize the performance of a single application inside the VMs, packaging the applications as virtual appliances (VAs) is useful [40], [46]. A virtual appliance (VA) can be defined as a highly optimized, pre-built and ready-to-run software package with a compatible operating system, running under a virtual machine (VM) (Figure 1.4). A VA is a new software distribution mechanism such that software can be easily delivered independent of the physical structure of the computing environment. The VA size can be reduced by eliminating unnecessary parts of the guest OS, so that, a VA employs less memory and executes faster than a general purpose VM [35].

Throughout this dissertation, the term virtual application appliance (VAA) is used. VAA refers

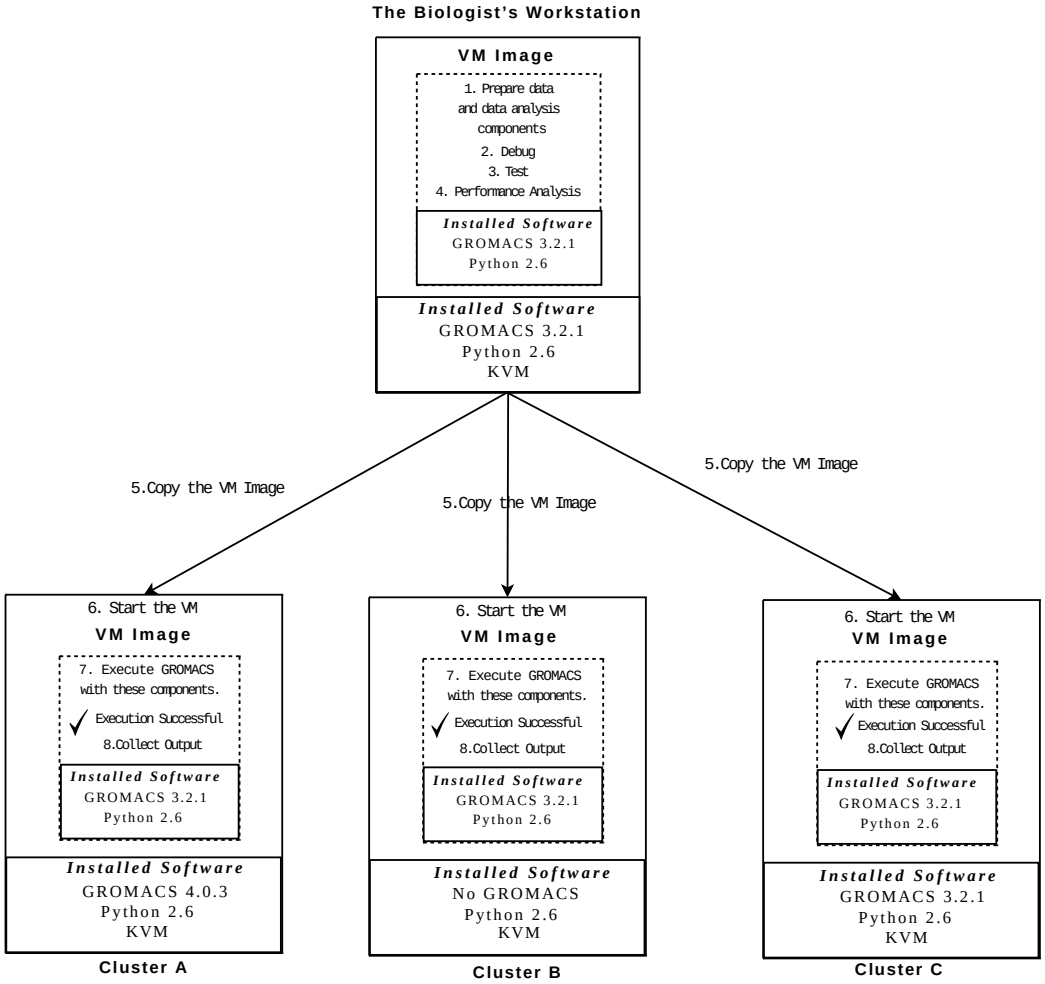


Figure 1.3: VM solution to the software heterogeneity problem

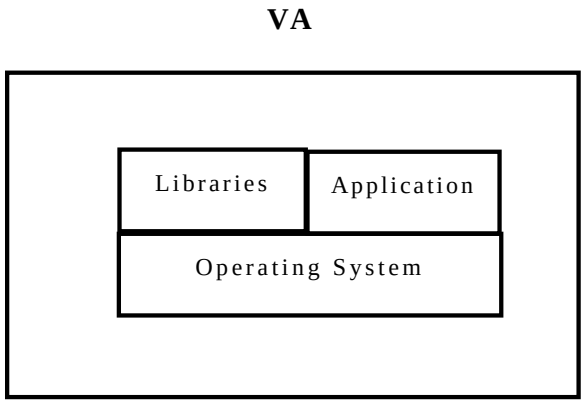


Figure 1.4: The VA contents

to a special kind of VA that runs to produce a result and then ends execution after the result. In other words, VAAs exclude VAs that are long-running such as databases and web servers. Therefore, most scientific applications packaged as VAs can be examples of VAAs such as GROMACS, GAFolder [18] and HMMer [21]. All the ideas and solutions throughout this dissertation are applicable to VAAs but they may not be applicable to all VAs.

Although VAAs are useful in HPC, they provide few or no adaptability features for the clusters. In the cluster, users generally have no idea about VAAs and how to use them. The users only interact with the host OS. Also, If we allow VAA accounts access to the users' accounts on the host without any restrcition, the unwanted operations can be executed on the host. Therefore, some security problems arise if we do not add strong authorization mechanism between the VAA and host. Further, in the cluster, the users submit jobs to a batch scheduler which decides when to run them according to availability of resources. In this case, interaction with the users may not be possible and/or feasible. Therefore, automated and secure submission, authentication and data movement mechanisms should exist between VAAs and host machines.

In the context of HPC workloads, the work done in this thesis proposes a solution for the efficient execution of VAAs as well as automated and secure authentication and authorization techniques and two automated stage-in stage-out data movement mechanisms. The first mechanism uses the network secured by the secure shell-based (SSH-based) secure channel protocol. The second mechanism uses a shared memory region between the VAA and host secured by the OS's restrictions on file access. Additionally, the VM migration functionality is examined. Finally, an evaluation of the techniques used in this thesis is provided.

1.1 Contributions

Contributions of this thesis can be enumerated as follows:

1. A bash script skeleton has been developed to automate data movement between the VM and the host in a secure and efficient manner. Two different mechanisms are designed for this purpose: secure copy over network and copy over shared memory. In the copy over shared memory mechanism, security is established by the OS's restrictions on file access. However, the secure copy over network mechanism uses SSH and its forced command feature. This architecture eliminates interactivity due to the login process and encrypts all data movements. Also, the user does not need to know anything about the VMs, security or details of data movement.

2. VM Migration with a simple save and restore technique is examined. Basically, this method saves the state of the VM and when it is necessary restores the VM from this saved state. We have not implemented the automated migration in our scripts, however, we evaluate the overheads of this migration technique.

3. An experimental study shows the performance impacts of our work. Benchmarks include overheads due to data movement and virtualization and time spent for the save and restore phases during the migration. One important contribution of this evaluation is the use of real data provided by bioinformatics researchers for the GROMACS experiments.

1.2 Concluding Remarks

In this chapter, we discussed the problems that a scientist may encounter if she wants to use an HPC center. We also explained how VMs can solve these problems. Then, we showed that VMs can be further optimized as VAAs. Finally we briefly explained our techniques to adapt VAAs to the cluster environment for the benefit of the user. In the next chapter, we discuss the motivation of this thesis in detail and provide some background knowledge on virtualization.

Chapter 2

Motivation and Background

In Chapter 1, we introduced the problems that a scientist can encounter when she wants to deploy her work to clusters. Also, we briefly explained our VAA solution for these problems. In this chapter, we provide the details about advantages of the VAAs on the cluster. We continue with presenting the challenges of adapting the VAAs to the cluster environment. Finally, a real life motivating example is followed by background knowledge on virtualization.

2.1 Benefits of Virtual Application Appliances on Clusters

In general, the VAAs have three major benefits over traditional software. First, the VAAs can run on any x86 system independent of the OS. The only requirement is the installation of a virtual machine monitor (VMM) application such as KVM which manages and runs the VMs. Second, the installation cost is minimal. An application is generally ready to run once its VM image files are copied to the system. The application is pre-configured in its package so that it has no extra configuration burden for the specific OS environment. Third, the VAAs can possibly simplify the maintenance of the software environment. The VAAs do not have the library, application or OS dependencies in the underlying system. Hence the OS, libraries and applications do not have to be optimized to work with each other in every VAA installation.

In the cluster environment, there are also specific benefits of the VAAs for the software heterogeneity problem which is the main problem in installing and configuring applications on the cluster. The cluster machines may have different OSes installed or different versions of the same OS installed. Even if we get rid of these problems by installing the same OS and library files on every machine, several library files and system files may still conflict with the application that is intended to be used on the cluster. Further, different applications may require different versions of these files. Furthermore, the users may have different preferences about application versions and/or configurations (Section 2.1.1). The VAA's isolated nature from the underlying system makes the VAA ideal solution for the software heterogeneity problem on the clusters. Since all the necessary files and library files are included to the VAA package, there is no need to seek for these files in the underlying

system. Also, each VAA has a separate execution environment, therefore, the library conflicts and OS problems can be overcome by packaging the application, compatible OS and library files as a VAA.

2.1.1 Benefits to the User

The VAA helps the user customize the application execution environment for her specific purpose. For example, the user may want to run an application that requires a specific version of Python [45]. In the usual scenario, the user may not find this software on the cluster or this version of Python may not be installed to the cluster because other users may be using another version of Python. Further, the desired Python version may be incompatible with the installed version of the OS. The VAA solves these problems by creating a user specific software execution environment with all the necessary applications, compatible OS and libraries. Therefore, the user does not need to worry about various conflicts she may encounter on the cluster. She can only use this pre-configured VM and execute her application on the cluster.

2.1.2 Benefits of VM Migration

As we stated in Chapter 1, we examine two benefits of VM migration. First, an application can be recovered after failovers by saving the states of the VM (i.e. application checkpointing). Second, the wall-time limit of the cluster can be avoided. This section explains these two conditions and the solutions of VM migration.

The application checkpointing technique saves the state of the application at regular intervals. Later, in case a failure happens, the application can be restarted from one of these states. However, not all the applications, compilers or OSes support the application checkpointing. Hence, one can package an application as a VAA and, regardless of the support for the checkpointing in the system, the VAA state can be saved regularly by the VMM. Then, after a failover, the VAA can be restored from one of the saved states and the application execution continues inside the VAA.

Another benefit of migration for the users is that it is a workaround for the cluster wall-time limit. The system administrators generally put a maximum execution time limit to prevent the user job from monopolizing the cluster. If the execution time of the user's job exceeds the wall-time, the batch scheduler simply kills the process when the wall-time has run out. However, the user's job may last more than the wall-time. Therefore, the user needs to adjust her work and make assumptions about the run-time of her scripts and/or programs and then she has to split her work to several tasks and submit them to the cluster separately. Therefore, delays are unavoidable. However, if she packages the applications of her job as a VAA, she can take advantage of the VM migration mechanism. For example, in Figure 2.1, let us say the wall-time is 24 hours for the cluster and the scientist estimates the GROMACS job will end in 5 days. We can script the VAA execution and save it in every 24 hours and resubmit it again after every 24 hours. After every resubmission, the VAA

is restored from the last saved state and continue to the execution without any data loss.

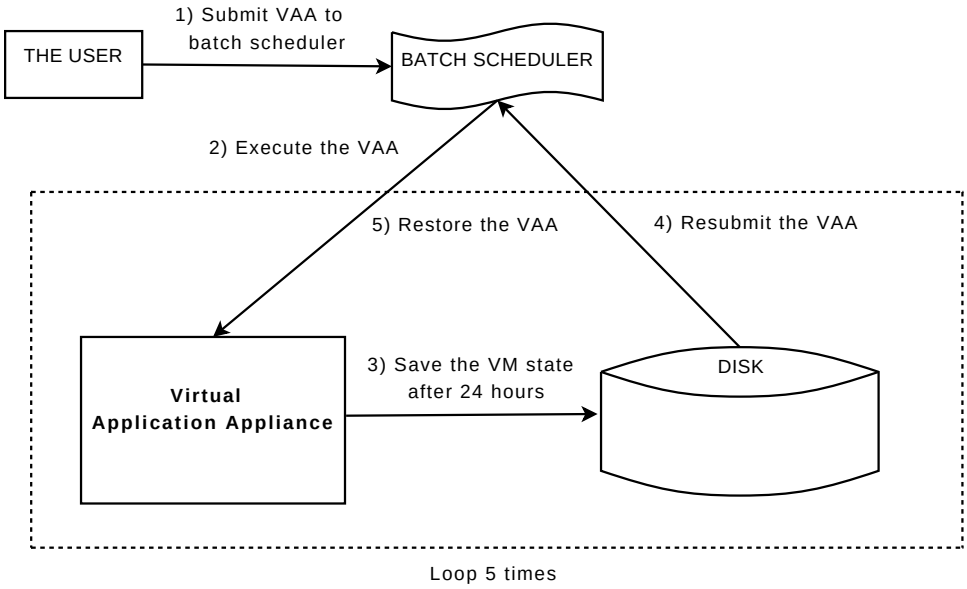


Figure 2.1: Save and restore type migration: Wall-time avoidance.

2.2 Challenges of Using Virtual Application Appliances on Clusters

The VAAs run on the clusters as the other user jobs, therefore, the VAAs should be adapted to the cluster environment. In the cluster environment, the user has to submit the job to a batch scheduler. After the submission, interaction with the user is not desired and may not be possible. Additionally, since the VAA has its own execution environment, input data has to be copied into the VAA and output data has to be transferred back to the host. Therefore, the VAA execution should be automated from the submission to the termination of the VAA. Hence, the challenges are the execution of the application inside the VM, authentication of the user to the VAA and data movement between the VAA and host. Also, the authorization of the data movement operations and encryption of data during data movement are important. Without any authorization and encryption, third parties who gain control of the VAA can see the data and access to the user account on the host unconditionally. In this section, we discuss the above-mentioned challenges and briefly explain our approach to these challenges.

2.2.1 Virtual Application Appliance Execution and Data Movement

A simple manual VAA execution involves several steps. Initially, the user has to login to the VM with a standard login procedure of the OS inside the VM (guest OS). Then, the user copies necessary input files and runs the application. Finally, the user transfers the output data from the VM to the

host.

There are several problems associated with manual execution due to interaction with the user. In a typical cluster, the user submits the job to a batch scheduler. The batch scheduler queues the job and if the resources are available, the batch scheduler executes the job according to the job's requirements. The waiting time due to queuing is not predictable and depends on the number of jobs and their priorities in the queue. Further, some batch schedulers do not support interactive programs at all or system administrators may disable the interactivity. Furthermore, when the job begins, the user should be ready to login to the VAA. If the VAA execution begins when the user is not available, the VAA sits idle and wastes CPU time. Therefore, after the submission, it is not feasible or even possible to interact with the VAA for authentication purposes. Similar problems apply to data movement. The user has to be ready before and after the VAA execution to initiate the data movement operations between the VAA and host. Consequently, the user's involvement reduces efficiency of the user's work as well as efficiency of the cluster.

2.2.2 Security

The data movement and authentication operations should be secured by some mechanism especially if the VAA and data are in different administrative domains (e.g. on Cluster A and Cluster B). The main reason of the security requirement is that the data and authentication information may pass through the wide area network (WAN) (e.g. Internet). Therefore, the data transferred across the WAN should be encrypted. Also, the security operations should minimize the user's involvement due to the interactivity problem that is mentioned in the previous section. Therefore, we cannot allow the user to enter some authentication information such as the username and password.

Our secure copy over network mechanism simply solves the interactivity problem by automating the security of the data movement and authentication operations with the SSH's data encryption and public/private key authentication. Also, our scripts automate the application execution, authoriztaion and data movement operations. Additionally, our copy over shared memory mechanism uses a shared memory region on the host and eliminates the SSH's encryption and authentication overheads if the VAA and data are on the same host.

2.3 Motivating Example

The University Of Alberta's Prion Project Group's [18] members who are involved in bioinformatics-related research use the GROMACS molecular dynamics application for their experiments. They use clusters because of their large number of jobs. The WestGrid clusters [17] or University Of Alberta's cluster are the primary platforms to be used by the researchers. In order to prepare their experiments, the researchers need to script the execution of GROMACS for the generation of the models and analysis of the results. Then, the GROMACS job is moved to the cluster but they encounter several problems.

One problem is the GROMACS version mismatch. The researchers need 3.2.1 to run their simulations but, especially, the new clusters have higher versions of GROMACS. One workaround for them is to install GROMACS to their home directories. However, the GROMACS developers suggest the GCC version 3.X compilers which may not be available on all the clusters. Either, they have to wait for the system administrator to install the software or choose another cluster.

They have also reported additional problems due to incompatibilities from their previous work. Any script that they need to use for different purposes has a potential to suffer from version incompatibilities or the software that they need may not be available at all (e.g. Python). Most importantly, they should either modify their code for different cluster environments with various software configurations or limit themselves to one set of cluster that they can use their scripts. The first choice, the code modification, puts extra effort outside of their scientific research and the second one, the limitation to one set of clusters, reduces their computational power.

Also, they stated another problem due to the wall-time limit of the clusters. For example, the WestGrid's glacier cluster has a wall-time limit of 10 days. However, some of their work needs more time to finish. Although, GROMACS has a restart option, the researchers find it unreliable in older versions. Therefore, the researchers often run their work in their workstations. Hence, the work is completed in a long time.

The researchers also added that monitoring the execution of the job is not trivial with current mechanisms [37]. They stated that they need to monitor the output files and do some analysis on them. If the results do not converge then they need to kill the job and submit their work with different parameters.

As this example shows, there are couple of benefits of the VAAs to scientists. Scientists reduce preparation time of scientific applications and delays due to heterogeneous software environments. Most importantly they can be independent of the software environment available on the cluster. Finally, scientific applications can be executed without the wall-time limit of the clusters.

2.4 Background

Virtualization created its own concepts and terms. In this dissertation's context, we only consider the system virtual machines which exclude the language-based virtual machines such as the Java virtual machine. Hence, in this section, after a brief discussion on virtualization's current popularity, several of the concepts will be introduced namely VMMs and its types, full virtualization, paravirtualization and virtual disk.

2.4.1 How Virtualization Became Widespread?

Virtualization of hardware and software resources are not a new idea but it gained its popularity after the 1990s. Certain companies such as VMware [23] and Microsoft [33] have developed products for enterprises. Unlike their early use for the software development projects [16], the enterprises started

to use them for several purposes such as server consolidation, security and fault-recovery [13]. This potential of virtualization stimulated the CPU vendors and they released CPUs with virtualization extensions.

A popular use of the VMs in enterprises is to achieve higher CPU utilization through server consolidation [36]. The researches show that most of the server CPUs are under-utilized (less than 30% utilized) in the enterprise data centers. Hence, the enterprises install the applications that traditionally run on dedicated servers to two or more VMs. Therefore, these VMs can be consolidated into one or more physical servers. By doing so, the enterprises reduce the power consumption and maintenance costs.

2.4.2 Concepts and Terms

In this section, we introduce some concepts and terms from the virtualization literature. We use most of these terms throughout this dissertation.

Virtual Machine Monitor and Types

Virtual machine monitors (VMMs) bring virtualization to the computer systems. The VMM simply puts a thin software layer on top of the hardware or OS and allows multiple VMs to run. Each VM has its own operating system (guest OS) and virtualized hardware resources. Although, the guest OS communicates with the VMM instead of directly with the physical hardware, the VM has an illusion of running on a completely physical system. In this context, the VMM becomes a hardware emulator and an isolator of the VMs.

We can categorize the VMMs in two types. The first type of the VMMs runs directly on top of the hardware. (Figure 2.2). Therefore, they act as OSes and virtual machine execution platforms. Examples of this type include Xen [5], VMware ESX server and KVM. The second type of the VMMs run on top of the OS (Figure 2.3). Therefore, they run as processes and, to some extent, controlled by the OS. Examples of this type are VMware server/workstation and Microsoft virtual PC/server.

KVM is easy to install on clusters. The main KVM module is integrated into the mainline Linux kernel after the version 2.6.20. Therefore, one does not need to install a separate VMM. However, some other VMMs such as VMware ESX server have to be installed separately. Needless to say, this installation comes with a lot of maintenance problems and may not be possible due to some dependencies to the Linux platform.

Full Virtualization and Paravirtualization

Full virtualization refers to full simulation of hardware resources that can be found in a computer system. In this type of virtualization, the guest OS does not aware that it is running on the VMM and the VMM controls the I/O requests from the VM. The OS can be installed without any modification

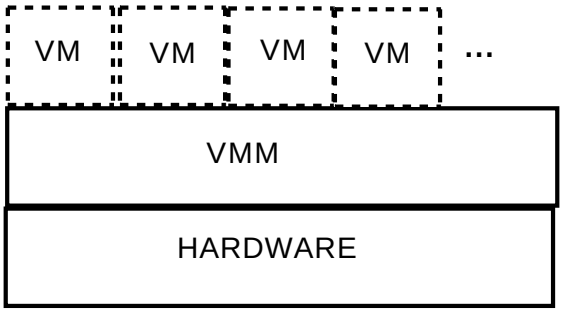


Figure 2.2: The VMM that runs on top of the hardware

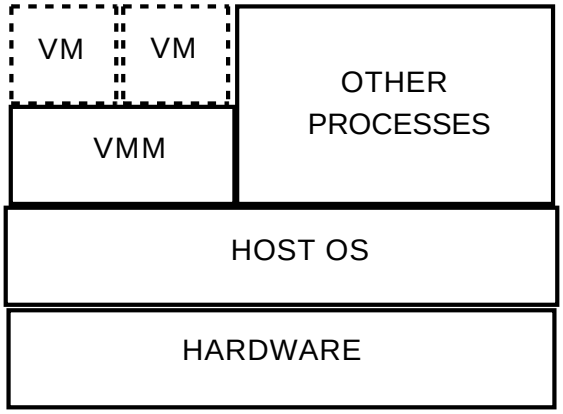


Figure 2.3: The VMM that runs on top of the OS

and the OS uses the VMM's simulated hardware. Generally, these hardware components are simple and do not reflect all the features of the physical ones.

Paravirtualization reverses the idea of full virtualization. In a paravirtualized environment, OSes are somewhat aware that they are running on virtualized environment. In the beginning, Xen started paravirtualization to find a solution to traditionally non virtualizable instructions of the x86 instruction set because they were causing significant performance degradation. To achieve that, Linux kernel is modified to add virtualized versions of these instructions. However, after Intel and AMD released their new CPU products with virtualization extensions, this idea become irrelevant and OSes started to run without any modification and performance degradation. Then, paravirtualization changed its direction to other hardware resources. Instead of modifying the OS, hardware drivers started to become aware that they are running on virtualized environment. *virtio*, a KVM feature, is one of the examples of this type of paravirtualization.

Virtual Disk

A file that is responsible for storing the VM's data is called virtual disk (VD). VD can also be defined as virtualized hard drive. The VD may have several formats and sizes. One can define a virtual disk with 8 GB capacity and allocate it before storing the data. Then, the size of the VD becomes 8 GB.

However, one can also create a VD with 8 GB capacity and enable dynamic allocation. Then, the VD grows as the data that is written on it grows until the 8 GB limit has reached.

Another words that can refer to the VD are the VM image or VM disk image. However, in this dissertation, when we use the VM image, we refer to the combination of VDs that makes the VM since the VM content can be stored in several VDs. Hence, the VM disk image refers to only one VD that is part of the VM.

2.5 Concluding Remarks

In this chapter, we discussed the motivation behind our mechanism and it is followed by background knowledge on virtualization. We started with explaining the benefits of the VAAs in general, on the clusters and for the users. Then, we stated the challenges of the VAA execution on the cluster. We concluded the chapter with brief history of virtualization and defining full virtualization, paravirtualization, virtual machine monitors and its types and virtual disk concept.

In the next chapter, we continue with all the aspects of our VAA design and implementation. We explain the security infrastructure, virtual disk layout of our VAAs, data movement mechanisms and scripts for the automated execution of the VAAs. Finally, we comment on the migration functionality.

Chapter 3

Design and Implementation of Virtual Application Appliances

In the previous chapter, we discussed our motivation behind our virtual application appliance (VAA) design. We also commented on background concepts related to virtual machines. In this chapter, we present the design and implementation of the VAAs. Throughout this chapter, when we talk about the design and implementation of the VAAs, we include all the scripts to automate the execution, security infrastructure and data movement mechanisms. Hence, we discuss the complete architecture and implementation details of our work.

3.1 Key Design Decisions

Our goal is to create a portable VAA for the cluster environment that handles security, data movement and application execution automatically and transparent from the user. We make design decisions under four category to accomplish this goal:

1. **Security of the operations (Section 3.2)**: We need to move data securely, handle authentication automatically and check that the data movement operations are properly authorized. To achieve this goal, we decide to use secure shell (SSH) because of SSH's wide availability on clusters. The SSH mechanisms such as encryption ensures that the data transfer is secure, the public-key authentication handles the authentication of the user and the forced command with our forced command script checks the authorization of the SSH operations. However, the policy on how the SSH mechanisms are used is another important design decision. We decide that the best policy is to initiate all the SSH calls from the VAA.

2. **Virtual Disk Layout (Section 3.3)**: We separate a VAA into four VM disks. The main reason behind this idea is to construct new VAAs easily by reusing the existing VM disks since most of the software infrastructure are similar for different applications such as the type of the OS and the SSH keys for security. Hence, we also add a virtual disk repository (VDR) to our

design to store these VM disks in a convenient location for easy access. Different VAAs can be created by selecting the necessary VM disks from the repository.

3. **Stage-in Stage-out of Data (Section 3.4)**: We need to find ways to move data fast, reliably and securely. If data and the VAA are on the different administrative domains, we decide to move data over the network by issuing SSH calls. We call this data movement mechanism as secure copy over network (SCN). However, if both the data and VAA are on the same host, we design another option to move data between the VAA and host over shared memory region on the host. The shared memory region option eliminates the SSH authentication and encryption overheads and authorization is handled by the OS. We call this data movement mechanism as copy over shared memory (CSM).

4. **VAA operations for security, data movement and application execution (Section 3.5)**: Our design requires five script files in different locations to automate VAA operations. We have to separate the VAA operations in five scripts because of their location dependence. The first script, the remote submission script, performs the operations to prepare an user job for submission to a batch scheduler on a remote cluster. This script has to be located on the host that the user prepares her input files. The second script, the local submission script, performs the operations to submit the user job to the batch scheduler on the head node of the cluster. This script has to be located on the head node of the cluster. The third script, the wrapper host script (WHS), performs the operations on the cluster outside of the VAA, in other words, on one of the nodes of the cluster that the VAA is running. Therefore, this script has to be located somewhere that is accessible by all the nodes of the cluster. The fourth script, the VM execution script (VMES), performs the operations inside the VAA. This script has to be located inside the VAA. The fifth script, the forced command script (FCS), performs the forced command operations that are authorization of the SSH calls, decompression of the output compressed file and cleaning up at the beginning and end of the VAA execution. Hence, this script also has to be located somewhere that the user submits her work.

3.2 Security Infrastructure

The security infrastructure of our design relies on the SSH mechanisms to secure VAA execution because of SSH's wide availability on the clusters. However, there are several policy options based on who is the initiator of the SSH calls. The initiator is basically the entity that starts and coordinates SSH calls. In the VAA context, the initiator may be the host or the VM.

Although both the host and VM initiator policies can establish the same degree of protection and privacy, several other criteria should also be considered to choose the most effective security policy in the cluster environment. The following section enumerates these criteria based on efficiency, portability and transparency. Then, Section 3.2.2 explains why the VM initiator is chosen.

3.2.1 Criteria of the Proper Security Policy

Three main criteria are considered to choose the policy of our security infrastructure:

1. **Efficiency**: An efficient security policy should not introduce significant overhead to performance. It should not include extra SSH calls that can be eliminated easily with other policies. Efficiency is especially important for the VAAs because they already have the data movement, hypervisor and guest OS overheads.

2. **Portability**: A portable security policy should be applicable to different systems easily. It should not introduce significant configuration changes on the cluster. In our design, SSH, by itself, mostly covers the portability criterion with its wide availability and easy installation properties.

3. **Transparency**: A transparent security policy should be as transparent as possible to the user. The user's involvement should be minimized for authentication, authorization and data security.

3.2.2 Security Policies Based On Initiators

We explain the implementation of the VM initiator and host initiator policies in the next two sections. Then, in the following section, we compare these two policies and state the reasons why we choose the VM initiator as our security policy.

3.2.2.1 VM Initiator

The VM initiator security policy initiates all the SSH calls from the VM and can be implemented by several steps without any user involvement (Figure 3.1). Initially, the VAA designer should create an account inside the VM to execute the application. The VAA designer creates a non-passphrase protected private and public key combination for that account of the VM. Then, the system administrator puts this public key to all the users' *authorized_keys* file on the cluster. Therefore, the users of the cluster give permission to the VM to execute commands. However, if we allow unrestricted access to the cluster user's files from the VAA, then a malicious person who gain access to the VAA can reach the cluster user's files unconditionally. Hence, we prepare a forced command script that restricts the VAA to only specific data movement operations to a specific location, so that, the user can be protected from the malicious attacks. In this case, the system administrator should attach this script's path to the public key of the VAA account (Section 3.5.5).

3.2.2.2 Host Initiator

The host initiator security policy initiates all the SSH calls from the host (Figure 3.2). In order to implement this type of security policy, the user has to be involved significantly. Initially, the user has to create a public and private key of herself. Since the user is probably in the multiuser

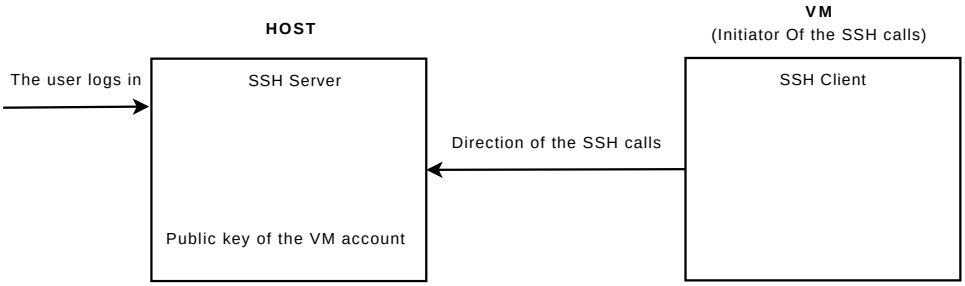


Figure 3.1: The VM initiator security policy

cluster system, a non-passphrase protected private key is not secure. Therefore, she should enter a passphrase during the creation of keys. Then, the designer of the VAA should include the cluster user's public key to the VAA's *authorized keys* file. After that, she has to run *ssh-agent* on the host and add her private key to this agent's cache. This operation forces the user to enter her passphrase. Only after that, the user can submit the job to the batch scheduler.

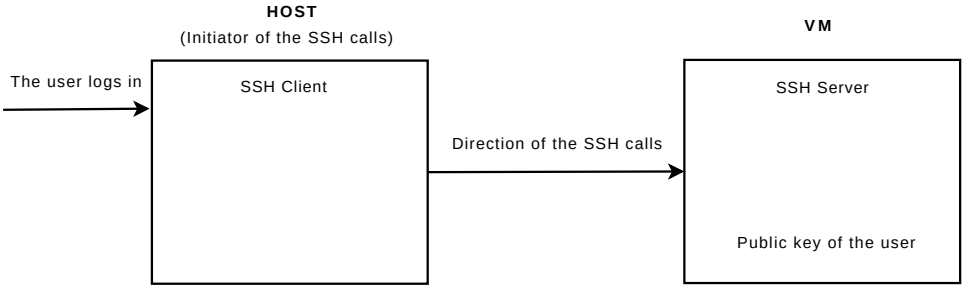


Figure 3.2: The Host initiator security policy

3.2.2.3 The Design Choice: VM Initiator

Our experience show that the VM initiator policy is the best in terms of transparency and efficiency criteria (Table 3.1). First, the VM initiator is more efficient because it eliminates all the extra SSH calls of the host initiator. The host initiator policy should poll the application state by extra SSH calls. Every extra SSH call becomes a bottleneck in terms of efficiency. Also, the user on the cluster does not have to run *ssh-agent* type of programs in the VM initiator to store the private keys because the cluster user's private key is not used at all. However, the host initiator has to use the cluster user's private key to communicate with the VM. Second, the VM initiator is more transparent to the user than the host initiator because almost everything is done inside the VAA or by the system administrator. However, in the host initiator policy, the user has to create the public/private keys and run *ssh-agent* to cache the private key before the submission. Also, the user has to provide the public key to the VAA designer in order to authenticate to the VAA. The user's requirements of creating keys and communication with the VAA designer increases the user's involvement, therefore, reduces the transparency. Third, in the VM initiator, a system administrator can easily do the configuration

automatically or by request. However, in the host initiator, by some mechanism, the administrator has to change the keys of the VAA. This operation causes extra management burden.

Security Policy	Efficiency	Portability	Transparency
Host Initiator	poor	good	poor
VM Initiator	good	good	good

Table 3.1: Comparison of the security policies

Although the VM initiator policy has all these advantages, it has a non-passphrase protected private key problem, therefore, it needs extra security and design arrangements. A non-passphrase protected private key can be a security concern if it can be seen by third parties. However, this drawback can be relieved by enforcing strong restrictions on operations of the VAA with a forced command script (Section 3.5.5). A forced command script is a requirement for the proper authorization of the VAA operations. Also, in order to pass arguments of the application and necessary parameters for the SSH connection, a parameter virtual disk should be manipulated by a program and should be mounted to the VM.

3.3 VM Disks and Virtual Disk Repository

A virtual disk repository (VDR), in our design, stores the VM disks of the VAAs in a convenient place on the cluster (Figure 3.3). Hence, the user or the designer choose from a selection of VM disks for different VAA constructions. Otherwise, creating a VAA from scratch is more complicated and has higher overhead. In this section, we explain the software composition of our VAA design. Then, we discuss the types of VM disks to store software to execute a VAA. Finally, we provide the details of the VDR concept that we use in this work.

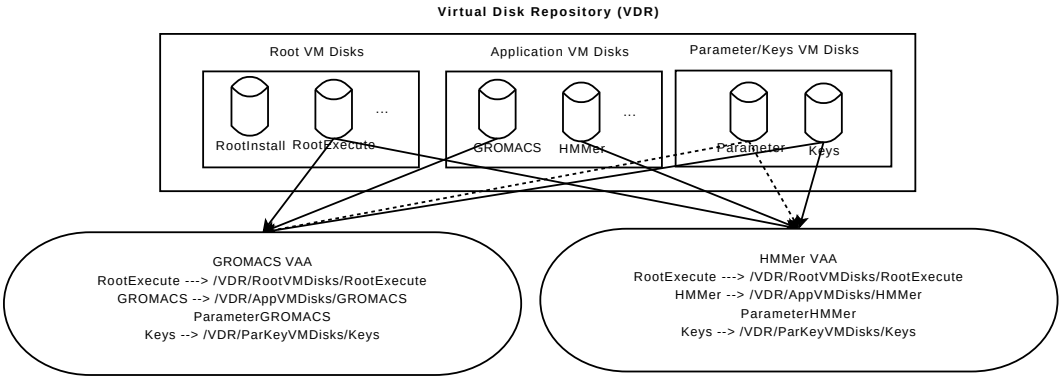


Figure 3.3: A virtual disk repository (VDR) example

In our design, the software composition of a VAA consists of a guest OS, SSH client, main application and auxiliary applications. The guest OS is chosen based on the needs of the main application. The SSH client is for communicating with the host and establishing the security infrastructure. The

main application is a scientific application that a VAA is prepared for. For example, for the GROMACS VAA, the main application is GROMACS. However, if the user wants to script the execution of GROMACS, she needs a scripting interpreter such as Python. Also, some of the executables inside the GROMACS VAA need the C preprocessor (cpp) to run inside the VM. Therefore, the auxiliary applications are for the execution of the user scripts or the execution of the tools inside the application packages. Additionally, a VAA configured with CSM should have a guest OS kernel module to interact with the shared memory region on the host and our C programs to read/write from/to the shared memory region on the host.

We store all the above-mentioned software, SSH-related parameters and key files in four VM disks for each VAA (Figure 3.3). The virtual machine monitor (VMM), the software which manages virtual machines, mounts these four VM disks to execute a VAA. In this dissertation, disk images are named as root, application, keys and parameter. A root VM disk is the bootable disk image that stores the system libraries and guest OS. We install the auxiliary applications to the application VM disk. However, if it simplifies the installation of the auxiliary applications, one can install them to the root VM disk too. For example, if the auxiliary applications reside in the software repository of the Linux distribution, we generally cannot install these auxiliary applications to a nonstandard location. As the name implies, an application VM disk stores the main application binaries and possibly some auxiliary applications. A keys VM disk stores the private keys of the VAA accounts for the SSH connections. A parameter VM disk image stores the SSH parameters. These SSH parameters, namely the username, IP address and current working directory of the host, are the minimum necessary parameters for the VM to initiate SSH calls to the host. In our design, the host cannot manipulate the VM after it starts execution, therefore, the VM needs to know how to communicate with the host.

The advantage of using separate VM disks for a VAA is reusability. One can install the application to an application VM disk along with a root image that contains necessary software for the installation. Figure 3.3 shows this disk as *RootInstall*. However, later, when she wants to execute the VAA, she can mount another root VM disk which eliminates unnecessary software for the execution. Figure 3.3 shows this disk as *RootExecute*. The *RootExecute* VM disk is the slimmed version of *RootInstall* VM disk [35]. For example, the HMMer's *RootExecute* VM disk does not contain the GNU compiler collection (GCC) because the GCC is only necessary for the compilation of HMMer but not for the execution of HMMer. Hence, one can use a root VM disk for the installation of several applications (e.g. *RootInstall*) to application VM disks and another root VM disk to execute several different VAAs (e.g. *RootExecute*). Consequently, she does not need to create a VAA from scratch by reusing the suitable root image. Also, we can save disk space by few root VM disks for every VAA instead of a complete VM disk for each VAA that also contains the software of a root image.

The VDR stores VM disks for each VAA. The repository can be a collection of symbolic links

that points to the locations of the VM disks or the repository can directly store the VM disks under a certain directory structure (Figure 3.3). The advantage of using symbolic links is that the VM disk images can be stored anywhere on the host and one can add links that point to these VM disk images to the repository. However, if a link is broken or the VM disk is inaccessible, the VAA execution cannot start. A convenient way is to store the VM disks or the links under three categories: root, application, parameter/keys. With this structure, a VAA can easily be constructed by mounting one VM disk from root and application category along with parameter and keys VM disks (Figure 3.3).

3.4 Data Movement Mechanisms: Secure Copy Over Network and Copy Over Shared Memory

We consider two different design options for the data movement mechanisms (Figure 3.4). The first design option establishes data movement over the network through the secure channel of the SSH protocol. We refer to this option as the secure copy over network (SCN) data movement mechanism. The second design option establishes data movement through the local shared memory region between the VAA and host. We refer to this second option as the copy over shared memory (CSM) data movement mechanism. In this section, we discuss the architecture, advantages and disadvantages of these data movement mechanisms.

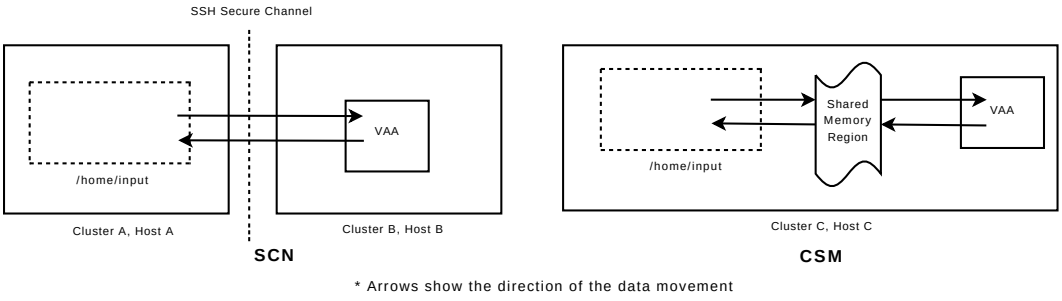


Figure 3.4: Secure copy over network (SCN) and copy over shared memory (CSM) data movement mechanisms

The first design option, SCN, is useful when data has to be moved over the network between the host and VAA. This is generally the case when the VAA and data are on the different administrative domains such as on two different clusters. SSH and its security features implement authentication and authorization of the user. Also, SSH encrypts data that is transferred across the network. However, every SSH connection has authentication and per-byte encryption overheads. If the VAA and data are on the same host, we can simply use CSM and eliminate encryption and authentication overheads.

SCN's architecture is the combination of an SSH client on the VAA, SSH server and virtual networking infrastructure on the host. In this architecture, an SSH client on the VAA communicates with an SSH server on the host. SSH's own authentication mechanism handles the authentication

part of the security. However, for the authorization, we do not just rely on the file access rights of the OS and put a forced command script between the host and VAA (Section 3.5.5). This forced command script checks the SSH calls that are initiated from the VAA for validity and only allows read/write operations to a specific folder which we call *virtual_root*.

The second design option, CSM, has lower overhead than SCN if complete local execution is possible. In other words, CSM allows faster VAA executions by eliminating the network and SSH authentication and authorization overheads if the input files and VAAs are on the same host. However, CSM is not applicable if the VAA needs to run on the different administrative domains because the shared memory region can only be shared between the VAA and host locally. Also, most virtual machine monitors (VMMs) do not support the shared memory architecture between the host and the VAA. Therefore, we use a non-standard modification to Linux KVM and a kernel module for the guest OS that add shared memory support. However, SCN can use widely available SSH implementations and unmodified VMMs.

The CSM's architecture is implemented by using Cam Macdonell's work on KVM [31]. In this architecture, a shared memory region which is accessible by both the host and VAA is created on the host. Basically, both parties read/write to this shared memory region to transfer data. In our implementation, we use Macdonell's implementation which requires a KVM modification to add support for a shared memory region between the host and VAA. This modification also requires a kernel module installed to the VAA to communicate with the shared memory region on the host. Instead of the SSH-based copy, two special programs are implemented by us to extract/insert input and output files from/to a shared memory region. However, the length of the output data being moved is communicated via SSH. Hence, we still need the SSH client on the VAA and networking infrastructure and SSH server on the host.

3.5 Automated Execution of Virtual Application Appliances on Clusters

We develop five separate script files to perform the data movement operations, authorization and VAA execution (Figure 3.5). The first script, the remote submission script, prepares the input files and sends the SSH parameters and VAA arguments to the head node of the cluster. The second script, the local submission script, submits the user's job to the batch scheduler with the necessary arguments for the VAA execution. The third script, the wrapper host script (WHS), executes the operations initiated from the host and the fourth script, the VM execution script (VMES), executes the operations initiated by the VAA. In CSM, the WHS also copies the compressed output file from the shared memory region, decompresses it on the host and executes the clean-up operations. The fifth script, the forced command script (FCS), is designed for validity and authorization check for the SSH operations initiated from the VAA. Additionally, the FCS decompresses output files and executes the clean-up operations in the SCN mechanism. We use the Bash scripting language in our

implementation because of its wide availability in the Linux OSes. However, these operations can be easily ported to other scripting languages.

Throughout this chapter when we talk about the VAA arguments, we refer to either the arguments of the main application or arguments of the user script that executes several commands related to the main application execution. For instance, on the one hand, the user may prefer running *mdrun* executable directly from the GROMACS VAA by providing several command-line arguments (Figure 3.7). In this case, the VAA arguments are the arguments of the *mdrun* executable. On the other hand, she may optionally run her script file (i.e. *run.sh*) that executes several GROMACS executables and other operations that are necessary for her work (Figure 3.13). In this case, the VAA arguments refer to the arguments of the user script.

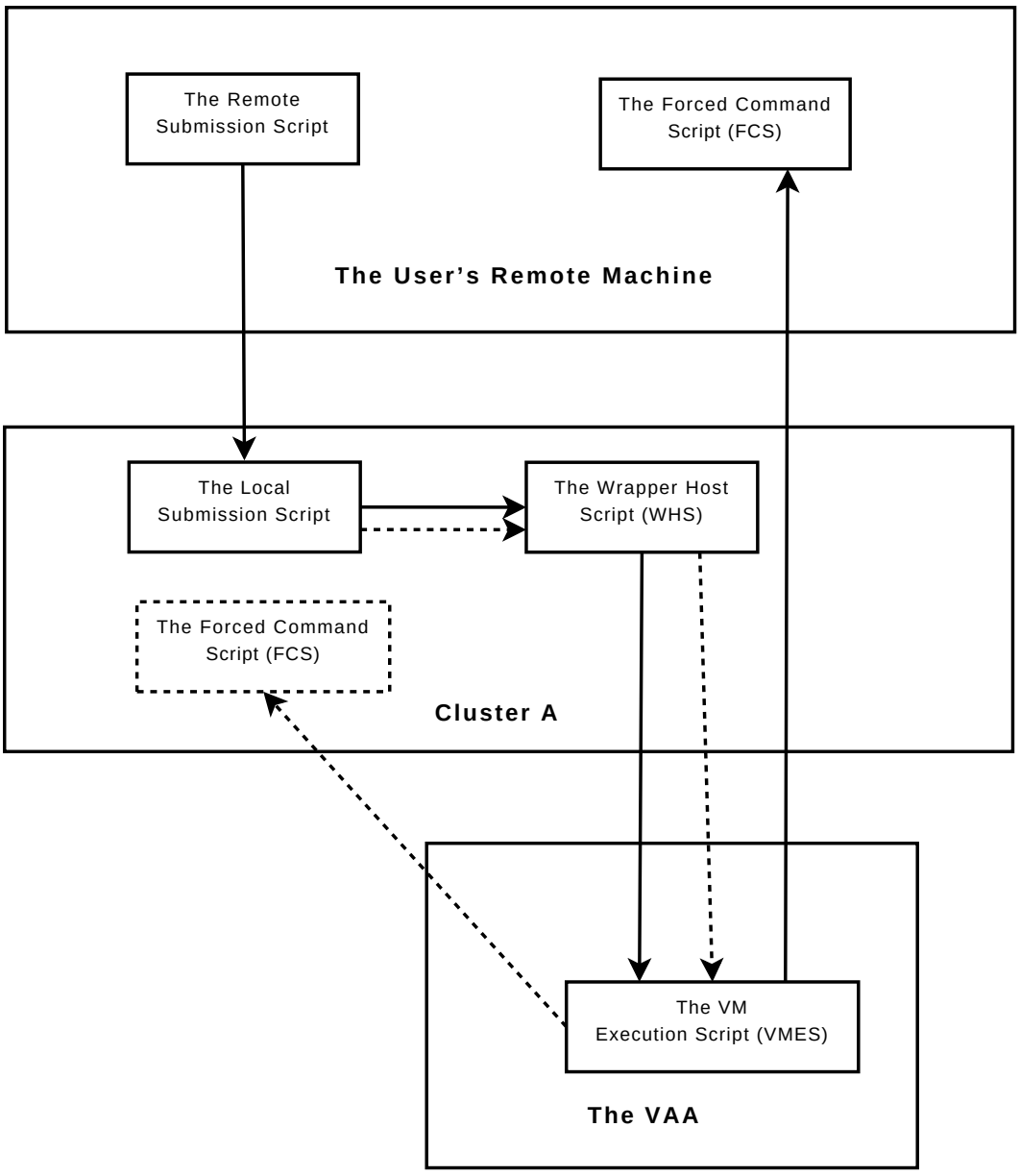
In the following sections, we explain the operations that are executed by each script in detail. We examined the migration functionality in the previous chapter. Therefore, at the end of this section, we comment on possible additions to these scripts if the migration functionality is needed.

3.5.1 Remote Submission Script

The remote submission script allows the user to submit her work from a remote location other than the cluster that she plans to execute her work (Figure 3.6). In our design, the user does not need to login to the cluster that the VAA execution takes place. Instead, she can prepare her input files on her workstation or on another cluster that has an internet access. Then, she submits her work from one of these remote locations. We call each of these remote locations as the user's remote machine.

As Figure 3.6 shows, the remote submission script defines a task number to distinguish the jobs that are using the same VAA at line 11. In our implementation, the task number is just a folder name that the job is submitted. Other options can be a ticket mechanism that assigns a unique job number to each job or a batch scheduler's task number that is assigned to each job. For example, TORQUE and PBS assign a unique job number to each submission. Immediately following the definition of the task number, the remote submission script writes the VAA arguments to the arguments file at line 14, then compresses the input files at line 17 and prepares the SSH parameters to be sent to the cluster at line 21. These parameters are used by the VAA to connect directly to the user's remote machine. Therefore, at the beginning of the VAA execution, the VAA gets the input files and VAA arguments directly from the user's current working directory on the user's remote machine. At the end of the VAA execution, the VAA puts the output files directly to the user's current working directory on the user's remote machine. The only assumption of the remote submission script is that the user has permission to authenticate to the cluster. Finally, at lines 24 to 27, the remote submission script connects to the cluster's head node, creates the execution folder on the cluster and then submits the user's job to the cluster's batch scheduler by calling the local submission script (Section 3.5.2).

The remote submission can only work for the SCN data movement mechanism (Figure 3.5). In CSM, since the user has to login to the cluster and make sure all the input files are on the cluster



*Dashed arrows represents the CSM mechanism and the other arrows represents the SCN mechanism.

Figure 3.5: The scripts and their execution order for both mechanisms

```
1 #!/bin/sh
2
3 #Assign current directory on the host to the crdir environment variable.
4 crdir=‘pwd‘
5
6 #IP of the localhost
7 localip=‘hostname -i‘
8
9 #Name of the current directory
10 dirname=‘echo $crdir | awk -F ’/’ ’{print $NF}’‘
11 tnumber=$dirname
12
13 #Write arguments of the application to the arguments file.
14 echo "$@" > args${dirname}
15
16 #Compress input files.
17 tar -zvcf ${tnumber}input.tar.gz *
18
19 #Prepare the command for the remote host \
20 #to write the ssh parameters to the paramater VM disk.
21 parcommand="parameteradd $tnumber $crdir $localip \
22    $USER \"/usr/botha10b/GromacsExec/$dirname/parameter-flat.vmdk\""
23 #Submit the job to the batch scheduler on the remote host
24 ssh $USER@headnode.cluster "mkdir /usr/botha10b/GromacsExec/$dirname; \
25         cd /usr/botha10b/GromacsExec/$dirname; \
26           export VMMemory=2048; export jobname=$dirname; \
27             submitmdrunPBSApp64BitRM $parcommand"
```

Figure 3.6: A sample GROMACS VAA remote submission script for the PBS and TORQUE batch schedulers

(Section 3.4), the above-mentioned preparation lines should be implemented in the local submission script (Figure 3.6 at lines 3 to 22). Figure 3.7 shows general and example submission command-lines for the GROMACS VAA (*[arguments]* refer to the GROMACS VAA's arguments for the execution of GROMACS application).

General submission command-line for the Gromacs VAA:
./submitremoteGROMACS [arguments]

Example submission command-lines for the *mdrun* executable:
./submitremoteGROMACS -o a.trr -g a.log
./submitremoteGROMACS -o a.trr -x a.xtc -c aconfout.gro

Figure 3.7: The general and example command-lines for the submission of GROMACS VAA to the batch scheduler.

As we stated previously, the only requirement of the user is to submit the job from the folder that the input files reside (Figure 3.8). For example, let us say the user stores the input files in the */home/input* folder. The only requirement for the user is to go to */home/input* folder and submit her work by using our remote submission script with necessary VAA arguments.

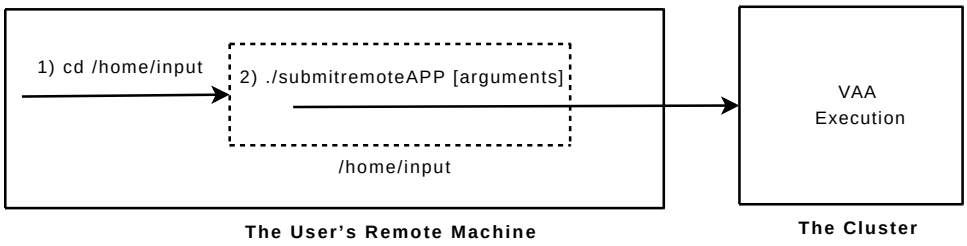


Figure 3.8: The remote submission script execution for only one cluster.

```
1 #!/bin/bash
2
3 curdirectory=‘pwd‘
4 prty=$@
5 grpath=‘which gromppLINUXPBSApp64BitRM.sh‘
6 #The submission line of the user job for the PBS/TORQUE type of schedulers
7 qsub -N $jobname -v parameteraddline=\""$prty"\" $grpath
```

Figure 3.9: A simple local submission script for the PBS and TORQUE batch schedulers

3.5.2 Local Submission Script

The local submission script submits the WHS to the batch scheduler of the cluster with a unique job name (Figure 3.9). Every batch scheduler has different notations to define the jobname and other parameters of the submission command. Therefore, this script should be specialized for different batch schedulers or should contain different batch schedulers' submission parameters.

The batch scheduler parameters are sent as arguments to the batch scheduler's submission command which is *qsub* in Figure 3.9. However, most batch schedulers allow to add these parameters to the submitted scripts. In our design, the WHS is the script that is submitted. We choose not to add the job name to the WHS. The user can assign a name to the job by using *jobname* environment variable instead of modifying more complex WHS. However, static parameters such as memory and CPU requirements of the VAA can be added to the WHS since they do not change from one execution to another. A sample parameters for the PBS-based schedulers that are embedded to the WHS can be seen in Figure 3.10 at lines 4 to 7. For example, the *#PBS -V* directive exports all the environment variables of the current shell to the submission script and the *#PBS nodes=1:ppn=4* directive requests 4 CPUs on one node. These batch scheduler directives should be added in the beginning of the script before all the other operations.

3.5.3 Wrapper Host Script

The WHS controls the operations outside of the VM environment on the execution host of the VAA (Figure 3.10). First, the WHS defines the environment variables related to the VM execution. Second, the WHS writes the SSH parameters to the parameter VM disk before the VAA starts. Third, the WHS starts the VAA. Fourth, after the VAA finishes execution, the WHS cleans up the execution folder in the SCN mechanism. In the CSM mechanism, the WHS copies the compressed output file

from the shared memory region, decompresses and executes the clean-up operations.

The preliminary operations assign paths of the VM disk images and job specific variables to several environment variables. As explained in Section 3.3, paths are the places of the VM disk images in the repository (Figure 3.10 at lines 25 to 33). However, the job specific variables are used to define the job and its requirements for a specific VMM. For example, in our implementation, we use KVM as the VMM. KVM expects different media access control (MAC) addresses (i.e. network identifiers) for the executions of the multiple jobs from the same VAA image. Otherwise, the VAAs may not have network connectivity due to conflicting MAC addresses. Therefore, one of the environment variables is a random MAC address from the KVM's MAC address range (Figure 3.10 at lines 18 to 22).

After the environment variable definitions, the WHS executes several sequential operations. These operations are explained as follows:

1. The parameter virtual disk should be copied to the execution folder before the VAA starts. In our design, we simply copy a small template parameter VM disk image from the repository to the current working directory because it should be unique for every VAA execution (Figure 3.10 at line 41). At this point, if CSM is used, two additional steps create a shared memory region and copies the compressed input file to the shared memory region (Figure 3.11). In our implementation, we develop two C programs called *createshm* and *dumpfile* for this purpose. *createshm* creates a shared memory region with the specified size and *dumpfile* runs *mmap* system call to map the compressed input file to the shared memory region.

2. A program called *parameteradd* modifies the parameter VM disk image and adds the SSH parameters (Figure 3.10 at line 46). The *parameteraddline* environment variable, which contains the SSH parameters, comes from the remote submission script if SCN is used (Figure 3.6 at line 21) and from the local submission script if CSM is used. Basically this disk image contains a file called *parameter.txt*. The *parameteradd* program seeks for the beginning of the *parameter.txt* file and writes the necessary parameters to initiate SSH calls from the VAA. These parameters are the task number, current working directory, username and IP address of the host in SCN and CSM. Additionally, only in CSM, we add the size of the input compressed file to the parameters.

3. After the preparation of the parameter VM disk, a call to the VMM executable starts the VAA. In our design, we used KVM as the VMM and its command line looks like in Figure 3.10 at lines 49 to 52. The *$KVMPATH* argument is the full path of the *kvm* executable (Figure 3.10 at line 35). The first 8 arguments after *$KVMPATH* mount the four necessary VM disks namely root, application, keys and parameter as the zeroth, first, second and third disks of the VM. The *-m* argument specifies the memory size that's going to be assigned to this VM. In Figure 3.10 at line 50, the memory size is specified as 512 MB.

```
 1 #!/bin/bash
 2
 3 # export all my environment variables to the job
 4 #PBS -V
 5 #PBS -S /bin/sh
 6 #PBS -q batch
 7 #PBS -l nodes=1:ppn=4
 8
 9
10 if [ "X$PBS_O_WORKDIR" != "X" ]; then
11         cd $PBS_O_WORKDIR
12 else
13         PBS_O_WORKDIR=‘pwd‘
14 fi
15
16 # generate a random mac address for the qemu nic
17 # shell script borrowed from user pheldens @ qemu forum
18 generate_mac() {
19         echo $(echo -n DE:AD:BE:EF ; for i in ‘seq 1 2‘ ; \
20          do echo -n ‘echo ":$RANDOM$RANDOM" | cut -n -c -3‘ ;done)
21 }
22 mac=‘generate_mac‘
23
24 #Path of the Root Image File
25 RootPath=\
26 "/usr/botha10c/unal/VMRepository/RootVMDisks/64BitRootWithGCC3-4NC.vmdk"
27 #Path of the Application Image
28 APPPath=\
29 /usr/botha10c/unal/VMRepository/ApplicationVMDisks/gromacs64BitNC.vmdk
30 #Keys diskimagepath
31 keyspath=/usr/botha10c/unal/VMRepository/ParKeyVMDisks/keys.vmdk
32 #Parameter disk image path
33 parpath=/usr/botha10c/unal/VMRepository/ParKeyVMDisks/parameter-flat.vmdk
34 #Path to kvm executable
35 KVMPath=/usr/botha10b/unal/kvm86/bin/qemu-system-x86_64
36
37 #The path of the current directory.
38 curdirectory=‘pwd‘
39
40 #Copy parameter disk file
41 cp $parpath $curdirectory
42
43 #write parameter file to disk image of the keys. 1st line task number,
44 #Note: These parameters are required \
45 # **before file args${tnumber}** is brought in via ssh.
46 ‘echo $parameteraddline‘
47
48 #Run the VAA
49 $KVMPath -hda $RootPath -hdb $APPPath -hdc $keyspath \
50        -hdd $curdirectory/parameter-flat.vmdk -m 512 \
51          -vnc :12 $RootPath -net nic,vlan=0,macaddr=$mac \
52            -net vde,vlan=0 -snapshot
53
54 #Delete current directory
55 rm -rf $curdirectory
```

Figure 3.10: A sample SCN wrapper host script (WHS) for the PBS and TORQUE batch schedulers

```
createshm $SHM_PATH $SHM_SIZE
dumpfile $SHM_PATH ${tnumber}input.tar.gz
```

Figure 3.11: The commands for creating a shared memory region on the host and copying the compressed input file to the shared memory region

The *-vnc* argument is an optional argument that denotes the VNC display number of this VM. One can view the VM's display device by connecting to this VNC display using a VNC viewer. VM's display device is equivalent to the monitor of a desktop computer. If a display is not necessary, replacing the *-vnc :12* argument with the *-nographic* argument disables the display. In this case, the user still can connect to the VM using SSH if she wants to check the status of the execution. However, she can not connect to the VM's display.

The *-net* argument determines the network options. The first *-net* argument creates a virtual network interface card (NIC) connected to the VLAN 0 with a random mac address denoted by *$mac*. The second, *-net* argument connects this virtual NIC to the VDE-based virtual network on the host.

The *-snapshot* argument tells KVM not to write any changes to the disk images. KVM writes all the VM disk manipulations to the temporary files during the execution of the VAA and after the execution, KVM removes them. This *copy on write (COW)* feature of the KVM allows us to submit multiple jobs with the same VAA without copying the VM disks for every execution and causing any corruption to the base VM disks.

The *-boot* argument specifies which device will be used as a boot device. The boot device can be a hard disk, floppy, cdrom drive or network card. If the boot device is not specified, like in our WHS example, the default boot device is the hard disk that is mounted as the zeroth virtual disk (*-hda* in Figure 3.10 at line 49). Therefore, KVM boots the VAA from the root VM disk.

Finally, if CSM is used, this command line is followed by another argument called *-ivshmem* [31]. Then, this argument is followed by the name and size of the shared memory region. Since it has already been created by the WHS, KVM skips the creation part and directly adds this file to the VM as another memory device. A sample *-ivshmem* argument that points to a 512 MB of shared memory region named as testfile as follows:

```
...  -ivshmem testfile,512
```

4. During the execution of the VAA, the VMES and FCS take their parts and the WHS waits for the VAA to stop (Section 3.5.4 and 3.5.5). After the VAA execution ends, the WHS removes the execution folder in the SCN mechanism (Figure 3.10 at line 55).

   In the CSM data movement mechanism, since the execution happens on the current working directory on the cluster, the execution folder and the current working directory are the same. Therefore, instead of removing the execution folder, the WHS issues the following additional commands to copy the compressed output file from the shared memory region on the host and to decompress the compressed output file:

   ```
   readfile $SHM_PATH ${tnumber}output.tar.gz $filesize
   ```

```
tar -zxf ${tnumber}output.tar.gz
```

Finally, the WHS of CSM removes the compressed output and VAA arguments files at the end of the WHS script. In the SCN mechanism, the decompression and clean-up operations happen inside the FCS since the WHS is not on the user's remote machine (Figure 3.5 and Section 3.5.5).

3.5.4 VM Execution Script

The VM Execution Script (VMES) controls the operations inside the VAA (Figure 3.12). The VMES starts its execution when the VAA runs and loads the guest OS and its services. For example, for the Ubuntu Linux OS, we add a line to the *rc.local* file to call the VMES. The Ubuntu OSes execute *rc.local* script right after the OS is loaded to the memory. Since we need networking, SSH and related services before the VMES starts, this is the most convenient place to call the VMES. The VMES have all the necessary instructions to complete the execution of the application or the script that is prepared by the user. At the end of the execution, the VMES shuts down the VAA automatically.

The VMES starts with defining necessary environment variables for the SSH calls and application execution. For example, in Figure 3.12 at line 4, the VMES defines the path of the GROMACS executables. At line 6, the VMES defines the maximum number of SSH calls before concluding on an error due to connection problems. Then, the VMES loads the *ssh-agent*'s environment variables at line 8, therefore, the *ssh* executable can locate the keys in the memory. The *ssh-agent* tool of SSH caches the private keys of the guest OS accounts in the VAA's memory. From line 10 to 15, the VMES reads the SSH parameters from the *parameter.txt* file on the parameter virtual disk. In the CSM mechanism, the VMES additionally reads the size of the compressed input file from the *parameter.txt* file. Therefore, in the VMES of CSM, an additional variable called *par_4* is assigned to the size of the compressed input file.

After that the VMES starts to execute several operations. They can be enumerated as follows:

1. The VMES creates an execution folder for the user's job in Figure 3.12 at line 23. This folder stores the input files. Also, the application execution occurs inside this folder. Then, at the following line, the VMES makes the execution folder the current working directory.

2. In Figure 3.12 from line 27 to line 41, the VMES attempts to copy the compressed input file from the host by using the SSH parameters that are read from the *parameter.txt* file in the parameter virtual disk. If the VMES succeeds, the operations continues, otherwise the if block, between the line 36 and 41, stops the operations because the VAA execution cannot continue without the input files. If the CSM mechanism is used, instead of an SSH call, the *readfile* program copies the compressed input file from the shared memory region as follows:

```
#!/bin/bash

#Export PATH
export PATH=$PATH:/usr/app/bin
#Maximum number of attempts before concluding on connection problems
maxtry=60
#Load agent defaults
. /etc/ssh-agent.env
#Read parameters from the parameter.txt file on parameter VM disk
i=0
while read line
do
        export par_$i=$line
        let i=$i+1
done < /parameter/parameter.txt

#par_0: task number
#par_1: current working directory
#par_2: username
#par_3: host IP

#Create folder that input files will be put and go to that directory
mkdir /usr/app/gromacs_execute
cd /usr/app/gromacs_execute

#Copy input tar files
i=0
until ssh -o StrictHostKeyChecking=no ${par_2}@${par_3} \
  input:${par_1}/${par_0}input.tar.gz > ${par_0}input.tar.gz \
     || [ $i -eq $maxtry ]
do
        sleep 0.5
        let i=$i+1
done

if [ $i -eq  $maxtry ]; then
        rm -rf /usr/app/gromacs_execute
        echo "Failed to establish an ssh connection!"
        halt -p
        exit 0
fi

#Untar input files
tar -zvxf ${par_0}input.tar.gz

#Remove input tar file
rm -rf ${par_0}input.tar.gz

#Run gromacs
if [ -f run.sh ]; then
        ./run.sh ‘cat args${par_0}‘ &> outgromacs.txt
else
        mdrun ‘cat args${par_0}‘ &> outgromacs.txt
fi

#Tar the output files
tar -zvcf ${par_0}output.tar.gz *

#Copy output files to the host
ssh ${par_2}@${par_3} output:${par_1}/${par_0}output.tar.gz \
        < ${par_0}output.tar.gz

#Remove execution folder
rm -rf /usr/app/gromacs_execute

#Shut down the VM
halt -p
```

Figure 3.12: A sample VM script for the SCN data movement mechanism

```
#!/bin/bash
python 1U3M_1_long_GromPy1g_Ber_Ber_RotRem_GenVel_XTC_VMware2.py $@
```

Figure 3.13: A sample *run.sh* contents for the execution of the GROMACS application inside the GROMACS VAA.

```
readfile /dev/ivshmem ${par_0}input.tar.gz ${par_4}
```

In Figure 3.12 at line 6, the VMES defines how many times the *ssh* program retries in case a connection problem happens. Also, one should define the duration between every try (In Figure 3.12 at line 32).

3. Before the execution of the application or the user's script, the last step is the decompression of the compressed input file to the execution folder (In Figure 3.12 at line 44). An archive utility such as *tar* can be used for the extraction.

4. The VMES is now ready to run the application or user's script. In our design, the user can just run the main executable of the application package or, optionally, can create a simple *run.sh* script to add some other operations such as external analysis of the output files. If the user decides to create a *run.sh* script, she has to call the main executable explicitly in this script. For example, in Figure 3.12 at lines 50 to 54, the VMES calls the *run.sh* script if the *run.sh* file exists. Otherwise, the VMES calls the main executable of the GROMACS package, which is *mdrun*. As discussed earlier, the user has to put *run.sh* script to the same folder that the input files reside, so that, it can be copied to the VAA along with the input files. Also, the arguments of the VAA are provided on the same lines by reading the arguments file with the *cat* command (Figure 3.12 at line 51 and 53). An example *run.sh* script can be examined in Figure 3.13. Note that, in Figure 3.13, the user calls the *mdrun* executable inside the Python script, therefore, she also adds this Python script to the current working directory. Finally, the VMES saves the output of the *run.sh* or the main executable to a file for further examination since the output is invisible to both to the user and the batch scheduler during the execution. This file is *outgromacs.txt* in Figure 3.12 at line 51 and 53.

5. At the end of the execution, the VMES compresses all the files or just the new files and modified files to send to the current working directory on the host (Figure 3.12 at line 57). In our design, the VMES simply compresses all the files. To send the compressed output file to the host, in the SCN mechanism, the VMES initiates another SSH call (Figure 3.12 at line 60), in the CSM mechanism, it calls *dumpfile* program. *dumpfile* copies the compressed output file to the shared memory region on the host. Also, the VMES writes the size of this file to the *dumpsize.txt* file and sends to the host via SSH since the shared memory region has no information on the size of the file. An example line to copy the compressed output file to

the shared memory region on the host is the following:

```
dumpsize=‘/usr/app/gromacsscript/dumpfile /dev/ivshmem \
   ${par\_0}output.tar.gz‘; echo $dumpsize > dumpsize.txt
```

6. Fnally, the VMES removes the execution folder (Figure 3.12 at line 64). Then, it shuts down the VM (Figure 3.12 at line 67). VAA execution ends at this point and control returns to the WHS.

During the VMES execution all the modifications of this script on the host is controlled by the forced command script (FCS). Therefore, the VAA cannot execute unauthorized operations on the host. We see the details of this script in the next section.

3.5.5 Forced Command Script

The FCS is placed on the host where the user submits the job with the local or remote submission scripts, and checks the VAA's operations for the validity (Figure 3.14). The FCS also decompresses the compressed output file on the host and invokes the necessary clean-up commands in the SCN mechanism. Mainly, the FCS's validity checks restrict the VAA by only allowing the data movement operations from/to the predetermined path which we call as *virtual_root*. Therefore, the FCS is called every time a VAA tries to execute a command on the host with an SSH call. The FCS's *virtual_root* concept is similar to the *chroot* command's *jail* concept [26]. However, the FCS is implemented such that it does not need root privileges to execute commands.

In order for the FCS to be called, one needs to add the VAA accounts' public keys to the user's *authorized_keys* file on the host with the forced command line (Figure 3.15). For example let us say Alice wants Bob to access Alice's computer or home folder for certain operations. If Alice adds only Bob's public key to Alice's *authorized_keys* file, then Bob can access Alice's home folder and execute any operation that Alice can execute. However, Alice has to permit only certain operations and restrict others. Therefore, Alice adds Bob's public key to her *authorized_keys* file with additional *command* word (e.g. *command=" /etc/ForcedCommandScript.sh"*) at the beginning of the public key line. Alice can put a path to a script or simply put the path of the command that Bob can execute. In our design, we need a complex control for the VAA operations, therefore, we develop the FCS and add its path to the beginning of the public key lines of the VAA accounts in the user's *authorized_keys* file on the host.

If the public key is added with the forced command line, the SSH server on the host calls the FCS whenever a VAA initiates an SSH call to the host. Then, the FCS checks the command-line of the SSH call which is passed by the SSH daemon as an *SSH_ORIGINAL_COMMAND* environment variable (Figure 3.14). We only allow data movement operations, therefore, the FCS can only accept two command-line formats. The first command-line, the input command-line, format is for moving the data from the host to the VAA:

```
#!/bin/bash

virtual_root=/home/user
#Check if the data movement is input or output operation.
input=‘echo "$SSH_ORIGINAL_COMMAND" | awk -F ’:’ ’{print $1}’‘
path=‘echo "$SSH_ORIGINAL_COMMAND" | awk -F ’:’ ’{print $NF}’‘

canon_readlink() {
   # Output a canonicalized version of what a link links to
   # Credit to Jesse Wilson @
   # http://publicobject.com/2006/06/canonical-path-of-file-in-bash.html

        OLDWD="$(pwd)"
        cd -P -- "$(dirname -- "$1")" &&
        LINK="$(readlink -- "$1")" &&
        cd -P -- "$(dirname -- "$LINK")" &&
        LINK=$(pwd -P)/$(basename -- "$LINK")
        cd "$OLDWD"
}

#Check if path is symbolic link or not
if [ -h "$path" ]; then
        canon_readlink $path
        path=$LINK
fi

#Get the folder of the file path
folder=‘echo "$path" | awk -F ’/’ ’{print substr($0,0,index($0,$NF)-1)}’‘
filefolder=$folder

#Check if folder exist or not
cd "$folder" > /dev/null 2>&1
if [ $? -ne 0 ] ; then
        echo "File does not exist"
        exit 0
elif [ -z "$folder" ] ; then
         echo "File does not exist"
         exit 0
fi

#Check if folder is rooted from virtual_root directory
filepath=‘pwd‘
folder=‘echo "$filepath" | gawk -v a=$virtual_root ’ {
      print substr($0,0,length(a))
}’‘
if [ "$folder" != "$virtual_root" ] ; then
        echo "File is not in the virtual_root!"
        exit 0
fi

#Everything is OK.
#Do the input or output data movement
if [ "$input" ==  "input" ]; then
        /bin/dd if="$path"
        #Clean up after the input files are copied to the VAA.
        rm $path
elif [ "$input" == "output" ]; then
        /bin/dd of="$path"
        tar -zvxf $path -C $filepath
        #Clean up after the output files are ready.
         rm $path
         rm $filepath/args${filefolder}
else
        echo "Access Denied: Wrong pattern in the string."
fi
```

Figure 3.14: General forced command script (FCS)

```
command="/etc/ForcedCommandScript.sh" ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA4VR9A
TsyBjZroyemlYnUIWXbzd3+4qZmK8hNRn4dwXosb1xBpginhBjVk9U4L4cO0zXxg2GKOSznGIJ1ab/3
1LA40KJVO2w81oK+CL1bVRcDw8JRvBLQZ5Mrzv67P3bFasSBiObBI1gMaTHa7/Kmeo7GCwFsicKB672
o+KKWKlJ4maeraNMMC3mrpyqnwp49hvg7dhRYZz7h7XblkK2LI1zf2ZsYUBzvxwVrhOVkY6OFtraHf8
hWv/I57qVEYSomnQXAVjw2XB5E/Bno54vJDE7N9BQt6N3Tki5CvpoUhkbFRF8gtUYOqAyICRjg+HbJb
wRXLRlvNzxz7n2ZE8PBUQ== root@172.16.202.134
```

Figure 3.15: A public key with a forced command script line

```
ssh username@HostIP input:$virtual_root/[SOMEFOLDER]/$TASKNUMBERinput.tar.gz \
                                > $TASKNUMBERinput.tar.gz
```

The above command-line tells the FCS to copy *$TASKNUMBERinput.tar.gz* file from the host to the execution folder on the VAA with the same name. As we explained before, the data movement operations can be targeted to only certain path on the host. This path is denoted as *virtual_root* in the above command-line and in the FCS (Figure 3.14 at line 3). Therefore, a valid command-line must have a path under the *virtual_root*.

The redirection part of the above command-line is $>$ *$TASKNUMBERinput.tar.gz* and it is for the input stream to be written to a file on the VAA. If the command-line passes the validity tests, the only thing that the FCS does is to read the input file from the path on the host and redirect the file's content to the standard output. The *ssh* command on the VAA can read from the standard output and with the redirection, the shell on the guest OS writes the output of the *ssh* command to the file. In our implementation, the *dd* comand reads the content of the input file (Figure 3.14 at line 54).

The second command-line, the output command-line, format is for moving the data from the VAA to the host:

```
ssh username@HostIP output:$virtual_root/[SOMEFOLDER]/$TASKNUMBERoutput.tar.gz \
                        < $TASKNUMBERoutput.tar.gz
```

The above command-line tells the FCS to copy the *$TASKNUMBERoutput.tar.gz* file from the VAA to the current working directory on the host. Inversely, the redirection part of the command-line is $<$ *$TASKNUMBERoutput.tar.gz* and tells the shell on the VAA to put the *$TASKNUMBER-output.tar.gz* file's content as an input stream to the standard input. Therefore, the FCS can issue a command and read the input stream to a file on the host . In our implementation, the *dd* comand reads the content of the standard input and writes to the file (Figure 3.14 at line 58).

The above-mentioned operations can only be executed if the command-line is in the correct format and the operations target a path under the *virtual_root*. Therefore, the FCS parses the command-line and checks the validity of the operation in several steps. In our design, the parsing and validity checks can be enumerated as follows:

1. The parser breaks up the command-line into the path of the file and operation identifier which is either input or output. In Figure 3.14 at line 5, we extract the operation identifier and assign it to the *input* environment variable. At line 6, we extract the path of the file and assign it to the *path* environment variable.

2. After parsing the command-line, before the validity checks, we examine the file path and check if it is a symbolic link or not. If it is a symbolic link, then, we convert it to the path that the symbolic link points to. In Figure 3.14 at line 22, we check if the *path* is symbolic link or not. If it is, we convert it to its canonical version with *canon_readlink* function. The *canon_readlink* function is defined between the line 8 and line 19.

3. Now we can do our first validity check which examines if the folder that the file is copied from/to exists or not (Figure 3.14 at lines 32 to 39). If it does not, the FCS rejects the operation with an error line (Figure 3.14 at line 34 and line 37)

4. The second validity check examines if the path is under *virtual_root* (Figure 3.14 at lines 42 to 49). If it does not, the FCS rejects the operation with an error line (Figure 3.14 at line 47)

5. The third and the last validity check examines whether the operation identifier is valid or not (Figure 3.14 at line 53 and line 57). It must be either input or output. If it is not one of them, the FCS rejects the operation with an error line (Figure 3.14 at line 64)

After the validity checks, the FCS executes the input or output data movement operation as explained previously and then executes the clean-up operations (Figure 3.14 at lines 53 to 65). After the input copy operation, the FCS removes the input compressed file (Figure 3.14 at line 56). After the output copy operation, in the SCN mechanism, the FCS first decompresses the output compressed file and then removes this file (Figure 3.14 at lines 59 to 61). Also, the FCS removes the arguments file which is not necessary after the VAA execution ends (Figure 3.14 at line 62). As discussed before, In the CSM mechanism, these operations are implemented in the WHS (Section 3.5.3).

3.5.6 Migration Functionality

Currently, we have not implemented the migration support in our scripts due to some instabilities in the *libvirt* library version 0.6.2 [29]. However, we performed enough experiments to evaluate the migration overheads by manually executing the *libvirt* migration operations. In Chapter 4, an evaluation of these migration overheads can be viewed. We examine the migration functionality in this dissertation to perform application checkpointing and avoid wall-time limit (Chapter 2). In this section, we remember the brief explanations of the reasons for using the migration functionality and possible extensions to our scripts.

The two possible reasons that are mentioned in Chapter 2 can be summarized as follows:

1. To avoid the wall-time limit of the cluster. In the cluster environment, it is a common practice to have a wall-time limit to prevent an application or user from monopolizing the cluster. Therefore, the wall-time limit enforces the cluster's batch scheduler to terminate the job after some time. However, some scientific applications may need more time than the specified

limit. Hence, the VAA can be saved before the wall-time has run out and restored from this state by resubmitting to the batch scheduler.

2. To recover after failover by application checkpointing. If the WHS saves several states of the VAA (i.e. checkpointing), in case a problem happens such as a power outage, file system corruption, guest OS errors and network failure, the VAA execution can be restored from one of the saved states.

The WHS and in some cases the VMES could be extended to support the migration. The WHS saves the VAA state before the above-mentioned conditions happen and then resubmits the WHS to the batch scheduler to restore the VAA from the saved state. However, in addition to the WHS extension, if the checkpointing decision comes from the VAA's internal state, an extension to the VMES is also necessary. Even if the VAA decides the conditions for checkpointing, the WHS issues the save and restore operations.

3.6 Concluding Remarks

In this chapter, we presented the design and implementation of the VAA. We proposed a security infrastructure that is transparent, portable and efficient based on the authentication, authorization and encryption mechanisms of SSH. We employ the VM initiator security policy strengthened by the SSH's forced command feature that initiates all the SSH calls from the VAA to conform to the efficiency and transparency criteria. Then, we explained the virtual disk repository (VDR) concept to store the VM disks of the VAAs in a convenient location. We proposed the secure copy over network (SCN) data movement mechanism for the remote data movement operations of the VAA. Also, we proposed a faster alternative for data movement in the local executions called copy over shared memory (CSM) data movement mechanism. The chapter continued with the explanation of five scripts that implement all these data movement, security and application execution operations. Finally, we discussed the migration functionality and how it can be implemented in our scripts to avoid wall-time limit and recover after failovers. In the next chapter, we evaluate the performance impacts of our work with several benchmarks.

Chapter 4

Empirical Evaluation

In the previous chapter, we discussed all the design principles, security infrastructure, features and scripts of our VAA solution for the cluster environment. In this chapter, we demonstrate the performance impacts of the application execution as a VAA, the overheads of our automated, scripted, secure data movement mechanisms and the overheads of the migration operations. We use three bioinformatics applications for the VAA benchmarks. The goal of the experiments is to answer two important questions: Are the VAAs competitive enough with the bare hardware execution of the same application, which we call the host execution of the application? Are the overheads of our data movement mechanisms and migration operations negligible with respect to the total run-times of the VAAs?

To answer these questions, we start this chapter by explaining the details of the scientific applications that we use. Then, Section 4.2 describes the test environment. Section 4.3 explains the details of the performance measurements. Finally, the following sections after Section 4.3 present and elaborate the results that our benchmarks provide. Also, the general conclusions and quantitative evidence from empirical evaluations are summarized at the end of this chapter in Table 4.17.

4.1 Scientific Applications for Benchmarks

We present three bioinformatics applications that we use for the benchmarks: GROMACS, GAFolder and HMMer (Table 4.1). We call these bioinformatics applications as the main applications. We also add auxiliary applications along with the bioinformatics applications to the VAA. We use these auxiliary applications in the host execution too. The auxiliary applications are for the execution of the user scripts or the execution of the tools inside the application packages. For example, the C preprocessor auxiliary application is used by the *grompp* tool of GROMACS. However, the Python script interpreter is added because we use an user script that is written in the Python language, which runs several tools from the GROMACS application package to perform simulations and analyze the output files.

The first application, GROMACS, is a molecular dynamics application package. GROMACS

Name	Description	Dataset(s)	Auxiliary Application(s)
GROMACS	A software package to perform molecular dynamics.	3 proteins: Human, Chicken, Turtle	C preprocessor and Python
GAFolder	Protein structure energy minimization software.	1 protein: Ubiquitin	none
HMMer	Hidden Markov Models (HMM) software for protein sequence analysis.	1 protein: Globin	BBS Perl Benchmark Script

Table 4.1: The scientific applications and auxiliary applications used in the benchmarks (Inside the VAAs and on the host)

represents the broad class of scientific applications that performs CPU-intensive computations. Therefore, GROMACS is an interesting application to test the performance of the VAAs. Generally, GROMACS input files are small in size, e.g. 964 KB, compared to the output files, e.g. 215 MB. *mdrun*, the main simulation program inside the GROMACS software package, is mostly computationally intensive. In GROMACS, simulations may depend on various random seeds; however, we use constant seeds and a homogeneous hardware environment to get deterministic results. Hence, we can have a fair comparison between the VAA and the host executions of the application in terms of performance. Three sets of proteins are analyzed during the benchmarks: turtle, human and chicken. Also, for the data analysis and several sequential executions of the GROMACS tools we use the Python script provided by our collaborators at the University of Alberta.

The second application, GAFolder, is a protein structure and energy minimization application developed by the University of Alberta Prion Group. GAFolder is a good candidate to investigate the class of applications which are really a collection of scripts and executables with a driver program that creates new processes (e.g. fork) for different tasks during the execution. We modify the source code to have identical outputs each time we run GAFolder. GAFolder can run in multi-threaded mode but we limit the number of threads to a single thread for both the VAA and host executions. We also eliminate all the random seed generator functions and place constant seeds as their return value. Hence, we can have a fair comparison between the VAA and the host executions of the application in terms of performance. We use an Ubiquitin protein file provided by our collaborators in our experiments. No auxiliary program is used. The input and output files are small in size, e.g. 240 KB (input) and 648 KB (output), with respect to the GROMACS ones.

The third application, HMMer, is a hidden markov models (HMM) application for protein sequence analysis. HMMer is another widely used application package. We use the BBS benchmark package [44] which includes several HMMer benchmarks. However, we only execute the *hmmsearch*-based benchmark with a globin protein file, which comes with the BBS benchmark package, to evaluate the performance of this work for the I/O intensive applications. Since this benchmark searches a sequence database with a profile HMM file, it does extensive disk reads. *hmmsearch* is also compiled as a single-threaded executable. *hmmsearch* searches 2.1 GB database, provided by

NCBI [14], during the execution. We included this database to the VAA too. We use 76 KB input files and *hmmsearch* produces 2092 KB output files.

4.2 Test Environment

We run our secure copy over network (SCN) benchmarks on the real cluster called Checkers at the University of Alberta (Table 4.2). Checkers, which is part of WestGrid, has 1280 cores powered by 2 CPUs on each of 160 nodes. Checkers also has one head node to control the job submission and the cluster's authentication. The 2.50 GHz quad core Intel Xeon CPUs are configured with 16 GB of RAM on each node. The operating system is Scientific Linux 4.7 (64 Bit) with 2.6.28.2 kernel in all the nodes. The TORQUE/Moab batch scheduler manages the job submission to the cluster. KVM is installed on all the nodes as the virtual machine monitor. The virtual distributed ethernet (VDE) and dnsmasq (dhcp and dns server) applications are configured on all the nodes for the virtual networking. Checkers also has a NFS-mounted storage system. However, to avoid NFS-related performance issues, we only allowed the applications and the VAAs to read and write to the local storage of the nodes.

Number Of Nodes	1 head node & 160 compute nodes
Number Of Cores per Node	8
Total Cores	1280
CPU Model	2.50 GHz Intel Xeon L5420 quad-core
Total RAM per Node	16 GB
Batch Scheduler	TORQUE/Moab
Operating System	Scientific Linux 4.7 (64 Bit)
Linux Kernel Version	2.6.28.2

Table 4.2: Checkers cluster configuration

We run our migration and copy over shared memory (CSM) benchmarks on four nodes of the Botha cluster in the Department of Computing Science at the University of Alberta (Table 4.3). Three of these four nodes has 12 cores powered by 2 dual core CPUs on each of 3 nodes. The fourth one is the head node to control the job submission and the cluster's authentication. 3.00 GHz dual core Intel Xeon CPUs are configured with 4 GB of physical RAM on each node. The operating system is Fedora Core 11 (64 Bit) with Linux kernel 2.6.29.5 in all the nodes. TORQUE batch scheduler manages the job submission to the cluster. KVM is installed on all the nodes. However, on Botha, KVM is compiled with shared memory support between the host and the VM. As in the Checkers cluster's configuration, the VDE and dnsmasq applications are configured on all the nodes for the virtual networking. Although Botha has a NFS-mounted storage system, to avoid NFS-related performance issues, we only allowed the applications and the VAAs to read and write to the local storages of the nodes.

As explained in Section 3.3, another component of our test environment is a virtual disk repository (VDR) which stores three sets of VM disks (Table 4.4). The repository stores the VM disks that

Number Of Nodes	1 head node & 3 compute nodes
Number Of Cores per Node	4
Total Cores	12
CPU Model	3.00 GHz Intel Xeon 5160 dual-core
Total RAM per Node	8 GB
Batch Scheduler	TORQUE
Operating System	Fedora Core 11 (64 Bit)
Linux Kernel Version	2.6.29.5

Table 4.3: Botha cluster configuration

constitute a VAA when they are used together in the right combination. The first set of VM disks, the root VM disks, contains either the base OS installation and the auxiliary applications or the combination of the base OS, the auxiliary applications and the GNU Compiler Collection (GCC). The root VM disks are for the compilation of the application or the execution of the application. The root VM disks can also be used for both purposes. For example, the *64BitRootWithGCC4* VM disk contains the GCC 4 compilers and is used for the compilations of HMMer and GAFolder. During the run-time, we use the *64BitRoot* VM disk to eliminate the GCC binaries which are unnecessary for the executions of HMMer and GAFolder. However, the *64BitRootWithGCC3* VM disk is used for both the compilation and the execution of GROMACS because GROMACS version 3.2.1 needs the GCC 3 series compiler during the compilation and the C preprocessor, which is part of GCC, during the run-time. The second set of VM disks, application VM disks, stores the application binaries after the compilation. For example, the *hmmer64bit* VM disk contains the 64 bit binaries of the HMMer application. The third set of VM disks are the parameter VM disk and the keys VM disk which are part of the VAA's security infrastructure as explained in Chapter 3.

The VM disks are created by VMware server in VMware's native *vmdk* format which are also compatible with KVM. However, we convert them to *qcow* format of KVM during the migration because migration is only supported with *qcow* type of disks. Although the VM disks have predefined maximum sizes, the VM disks are not pre-allocated, which means their sizes grow gradually as data is written on them.

Root VM Disks	64BitRoot 64BitRootWithGCC3 64BitRootWithGCC4
Application VM Disks	gromacs64Bit hmmer64Bit gafolder64Bit
Parameter and Keys VM Disks	parameter keys

Table 4.4: Virtual disk repository (VDR) structure (64 denotes that the applications or OSes are 64 bit)

When we talk about a VAA, we talk about combination of a root VM disk, an application VM disk, a parameter VM disk and a keys VM disk. Basically, KVM mounts these four VM disks when

it starts the VAA. The only bootable one is the root VM disk. The combinations of the VM disks that we use for the VAAs can be examined from Table 4.5.

VAA Name	VM Disks
GROMACS VAA	64BitRootWithGCC3 + gromacs64Bit + parameter + keys
GAFolder VAA	64BitRoot + gafolder64Bit + parameter + keys
HMMer VAA	64BitRoot + hmmer64Bit + parameter + keys

Table 4.5: The VAAs and their VM disk combinations

4.3 Details of the Evaluation Method

The aim of our benchmarks is to show that the VAAs have acceptable performance to run on the clusters and our data movement mechanisms incur negligible overhead. We also used different types of software to understand the performance of the I/O and compute-intensive workloads. We compared the results with the host execution time of the application.

We prepared three sets of benchmarks. The first benchmark set measures the performance impacts of the SCN data movement mechanism. The second benchmark set measures the performance of the CSM data movement mechanism. The third benchmark set is about the migration overheads. In this set, the time spent for the save and restore operations are measured.

We divide the total scripted VAA execution into seven stages during the execution of a VAA. Other than the *VM Boot (B)* and *VM Shutdown (S)* stages, all the stages are explicitly measured. The stages are explained as follows:

1. **Input Files Compression (IC)**: We measure the time spent for the compression of the input files into a single compressed input file with the *tar* command.

2. **VM Boot (B)**: At this stage, we consider the time spent from the start of the VM until the beginning of the VM script, which performs the data movement and application execution operations inside the VM. Note that we derive the time spent for this stage along with the *VM Shutdown (S)* stage as explained in the following performance measurements paragraph.

3. **Data In (I)**: We measure the total time spent for the transfer of the compressed input file to the VM and decompression of the compressed input file to the input files with the *tar* command. Note that we separate out the 1st stage, *Input Files Compression (IC)*, from this total.

4. **Application Execution (E)**: At this stage, the VM Script starts the application such as the *gafolder* executable or the user script such as the Python script of the GROMACS VAA as explained in Section 4.1. Therefore, we measure the time spent for the complete execution of the application or the user script.

5. **Data Out (O)**: After the completion of the application execution, we measure the total time spent for the compression of the output files into a single compressed output file with the *tar*

command and transfer of the compressed output file to the host. Note that we separate out the 7th stage, *Output Files Decompression*, from this total.

6. **VM Shutdown (S)**: At this stage, we consider the time spent from the end of the VM script operations to the end of the VAA execution. Note that we derive the time spent for this stage along with the *VM Boot (B)* stage as explained in the following performance measurements paragraph.

7. **Output Files Decompression (OD)**: We measure the time spent for the decompression of the compressed output file to the output files.

Our performance measurements are as follows. The parentheses show the corresponding components for each performance measurement from the above-mentioned stages.

1. **Data Movement Overhead (IC + I + O + OD)**: This measurement includes all the data movement stages. Therefore, it is equivalent to the total of time spent for compressing/decompressing/copying input files (IC + I) and compressing/decompressing/copying output files (O+OD).

2. **Application Execution (E)**: This measurement is the equivalent of the 4th stage. Both the host execution and VM execution have an E component. In fact, the host execution does not require any data movement and VM-related operations, therefore, it has only this component.

3. **Total VAA Execution (B + I + E + O + S)**: This measurement is recorded as a whole. In other words, *Total VAA Execution* is the time spent from booting up the VAA to shutting down the VAA. It covers the time spent for the VM Script operations and also the boot up and shutdown procedures of the VAA. Most data movement overheads are in this total too: copying/decompressing the input files (I) and compressing/copying the output files (O). After the VAA ends, the user can finally reach to the compressed output file, therefore, we used this measurement to compare with the host execution of the application.

4. **VM Boot Up/Shutdown Overhead (B + S)**: This measurement is the total time spent for booting up the VM and shutting down the VM. We referred this measurement as the *VM start/stop overhead* in the tables. We derive this total by the subtraction of in-VM data movement operations (I+O) and *Application Execution (E)* from *Total VAA Execution* (*Total VAA Execution - I - E - O*), because we do not explicitly measure the *VM Boot* and *VM Shutdown* stages.

5. **Total Scripted Execution (IC + B + I + E + O + S + OD)**: This measurement is the total of all the stages from the 1st to the 7th. In other words, *Total Scripted Execution* covers *Total VAA Execution* and non-VM data movement overheads (IC+OD). We derive this total by adding

the time spent for *Input File Compression* and *Output File Decompression* stages to *Total VAA Execution* (*Total VAA Execution + IC + OD*).

We collect all the above-mentioned performance measurements for the SCN benchmarks. However, we only collect the data movement overhead measurement for the CSM benchmarks since the only change is in the data movement mechanism. This change does not affect the application execution or the VM boot up/shutdown times. Additionally, for migration benchmarks, we collect the following time measurements:

1. **Save Overhead**: It measures the time spent for saving the VM state to the disk. We initiate the save operation in the middle of the application execution. This measurement also includes the time spent for destroying the VM process.

2. **Restore Overhead**: It measures the time spent for restoring the VM state to the memory. After this operation, the VAA starts and the application execution continues where it had left off.

4.4 Secure Copy over Network Data Movement Benchmarks

In these benchmarks, we evaluate the above-mentioned overheads and the performance measurements for the SCN data movement mechanism. In all the benchmarks, we use 64 bit binaries of the applications and 64 bit Ubuntu Jeos 8.10 OS inside the VAA. Ubuntu Jeos allowed us to have a minimal OS without extra burden. The rationale behind using 64 bit binaries is to have a fair comparison between the 64 bit environment on the host and inside the VAA. However, the user is not restricted to use a 64 bit OS or application binaries inside the VAA. KVM can also run 32 bit VAAs on the 64 bit host. We execute the VAAs locally and the network transmission between the host and VAA happens inside the cluster environment. We use only one virtual processor inside the VM. Also, the VM disks are growable VM disks which are not pre-allocated to their full sizes.

We run instances of the VAAs in 2048 MB, 1024 MB, 512 MB, 256 MB and 128 MB of VM memory sizes. We want to understand that how variations in the VM memory sizes affect the performance. We also run the applications along with their auxiliary applications and the user provided scripts on the host for the comparison purposes. We reserve the one whole node of the cluster to the host or the VAA execution of the application. Therefore, we ensure that the node contention is as low as possible. We submit the jobs to the batch scheduler each time and let the batch scheduler to find a free node. Every protein's performance numbers are the average of 20 runs for each VM memory size. Similarly, the host execution times are also the average of 20 runs.

In the *total VAA execution times normalized to the host execution times* graphs, the standard deviations are less than one percent of the total execution for all the bars, therefore, they are not shown. Also, *boot up/shutdown overhead* graphs show a simple subtraction of in-VM data movement oper-

ations (I+O) and *Application Execution (E)* from *Total VAA Execution* (*Total VAA Execution - I - E - O*). Therefore, they do not include the standard deviations which may not be accurate.

4.4.1 GROMACS VAA Benchmarks

Three proteins, human, turtle and chicken are analyzed by the GROMACS VAA and the host installation of GROMACS. The simulation time parameter is 500 ps. All the results of the GROMACS benchmarks for the SCN mechanism can be examined from Tables 4.7, 4.8, 4.9.

Figure 4.1 shows the result of the total GROMACS VAA execution times normalized to the host execution time of GROMACS. From this figure, we can conclude that GROMACS is suitable for the VAA-based execution. Most of the time total VAA execution time is close to the host execution time of GROMACS. Although we always see a performance degradation for the 128 MB VM memory sizes, the memory limit does not add an extra overhead except for the chicken protein. One of the reasons for the chicken protein's significant performance degradation may be its high memory usage due to more computations from other proteins. Also, since the VM memory sizes over 128 MB had similar performance, we can also conclude that our GROMACS simulations are not memory-intensive, therefore, GROMACS VAA has reasonable resource requirements.

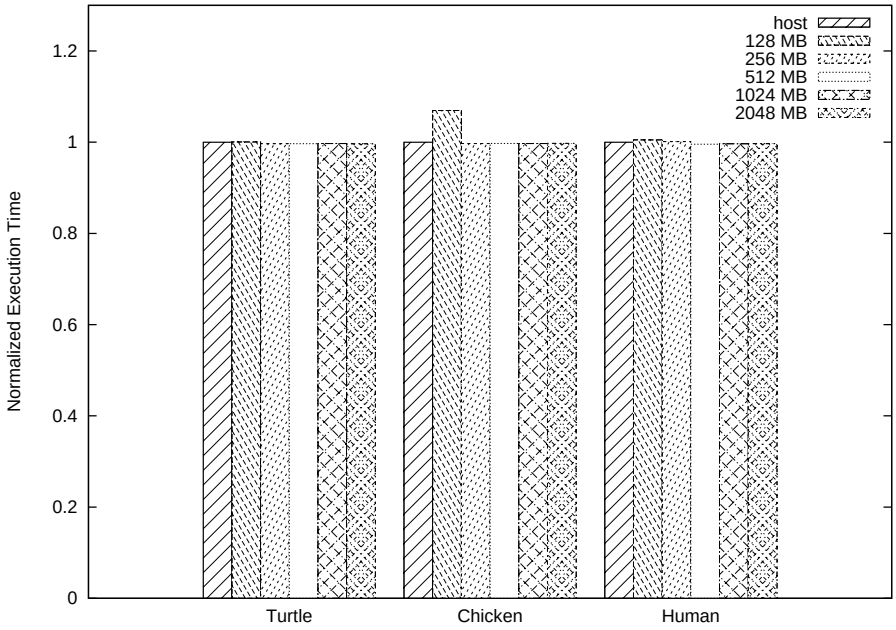


Figure 4.1: Total execution time of the GROMACS VAA for different VM memory sizes normalized to the host execution time of the GROMACS application. The GROMACS VAA achieves near-native performance for all proteins (Tables 4.7, 4.8, 4.9).

The data movement overhead is directly affected by the size of the compressed input and output files. Table 4.6 shows the input and output files total sizes and their compressed sizes. A regular GROMACS VAA has a negligible data movement overhead if we compare it with the total VAA

execution. For example, chicken protein total VAA execution is 61749.5 seconds and the data movement overhead is only 39.19 seconds with 2048 MB memory (Table 4.7). Figure 4.2 shows that the data movement overheads are less than 45 seconds in all the cases, however, total run-times are longer than 55000 seconds.

Protein Name	Input Files Total Size	Output Files Total Size	Compressed Input File Size	Compressed Output File Size
Turtle	1000 KB	202 MB	189 KB	94 MB
Human	964 KB	215 MB	175 KB	99 MB
Chicken	1064 KB	220 MB	193 KB	102 MB

Table 4.6: Total input and output file sizes of the GROMACS 500 ps benchmarks (Compressed/Uncompressed)

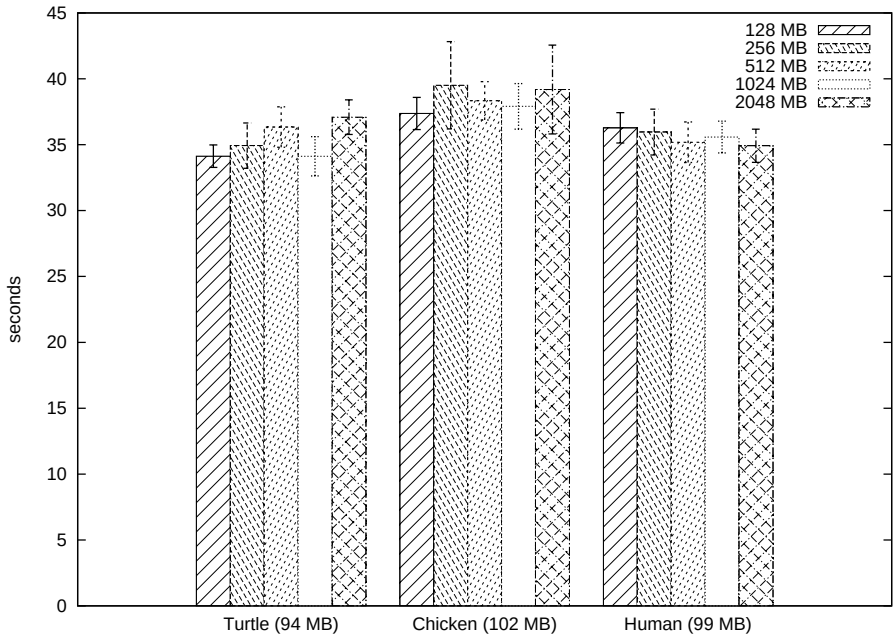


Figure 4.2: Data movement overhead of the GROMACS VAA for different VM memory sizes (The numbers inside the parentheses show the compressed output file sizes which affect the most of the data movement overhead). The data movement overheads of the GROMACS VAA are small with respect to total GROMACS VAA execution times which are more than 15 hours (Tables 4.7, 4.8, 4.9).

Another important overhead is the total of boot up and shutdown times of the VAA. In the standard OS installation, the boot up time and shutdown time may become significantly high due to extra processes such as graphical user interfaces, browsers, automounters and office applications. However, with this minimal OS it takes at most 39 seconds (Figure 4.3). Therefore, this overhead is also negligible with respect to the total execution time of the VAA.

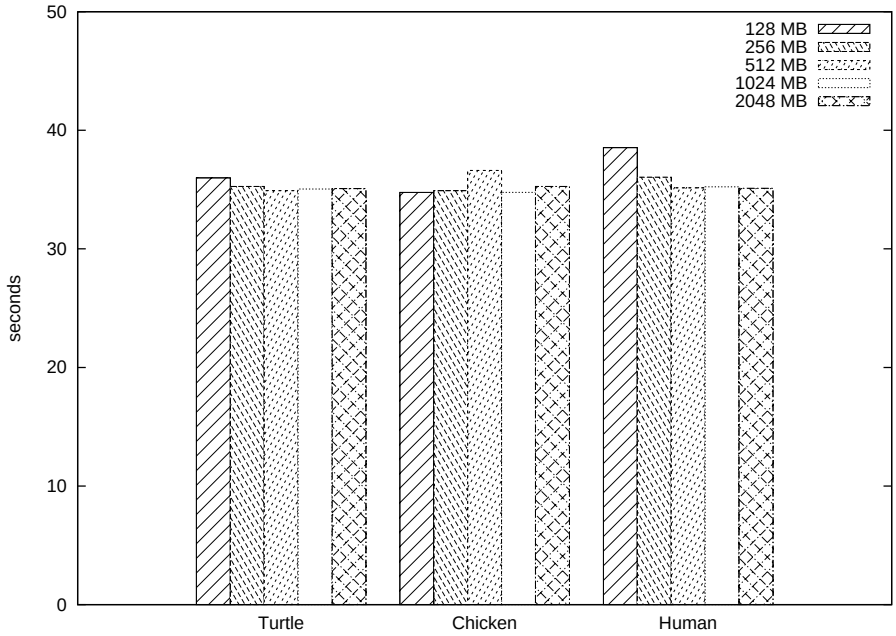


Figure 4.3: Sum of boot up and shutdown times of the GROMACS VAA for different VM memory sizes. VM boot up/shutdown overheads of the GROMACS VAA are small relative to total execution times of the GROMACS VAA which are more than 15 hours. (Tables 4.7, 4.8, 4.9).

GROMACS Chicken	**128 MB**	**256 MB**	**512 MB**	**1024 MB**	**2048 MB**	**Host**
Application Execution	66147.06	61660.13	61667.25	61651.91	61684.67	61902.89
Data Movement Overhead	37.37	39.51	38.33	37.91	39.19	N/A
VM start/stop Overhead	34.76	34.91	36.62	34.76	35.26	N/A
Total VAA Execution	66211.84	61724.86	61733.64	61715.71	61749.50	N/A
Scripted Total Execution	66219.43	61734.73	61742.66	61724.80	61759.52	N/A

Table 4.7: Checkers cluster GROMACS benchmarks: All the results for the chicken protein (in seconds)

GROMACS Turtle	**128 MB**	**256 MB**	**512 MB**	**1024 MB**	**2048 MB**	**Host**
Application Execution	55674.17	55469.56	55468.74	55487.9	55449.16	55709.37
Data Movement Overhead	34.13	34.93	36.34	34.12	37.09	N/A
VM start/stop Overhead	35.98	35.26	34.9	35.05	35.08	N/A
Total VAA Execution	55737.48	55531.46	55530.8	55549.08	55512.99	N/A
Scripted Total Execution	55744.54	55539.95	55540.2	55557.32	55521.54	N/A

Table 4.8: Checkers cluster GROMACS benchmarks: All the results for the turtle protein (in seconds)

GROMACS Human	**128 MB**	**256 MB**	**512 MB**	**1024 MB**	**2048 MB**	**Host**
Application Execution	60393.61	60166.28	59836.03	59891.28	59907.19	60160.89
Data Movement Overhead	36.28	35.96	35.19	35.58	34.92	N/A
VM start/stop Overhead	38.53	36.05	35.14	35.23	35.11	N/A
Total VAA Execution	60461.17	60230.67	59899.41	59954.29	59970.01	N/A
Scripted Total Execution	60468.85	60238.57	59906.60	59962.28	59977.41	N/A

Table 4.9: Checkers cluster GROMACS benchmarks: All the results for the human protein (in seconds)

4.4.2 GAFolder VAA Benchmarks

Ubiquitin protein is analyzed by the GAFolder VAA and the host installation of GAFolder. The simulation time is adjusted by setting the simulation iterations to 200. All the results of the GAFolder benchmarks for the SCN mechanism can be examined from Table 4.10.

GAFolder Ubiquitin	128 MB	256 MB	512 MB	1024 MB	2048 MB	Host
Application Execution	4452.81	3596.79	3595.64	3599.66	3598.70	3959.10
Data Movement Overhead	1.83	1.8	1.73	1.29	1.73	N/A
VM start/stop Overhead	36.87	36.67	36.73	36.84	36.58	N/A
Total VAA Execution	4490.74	3634.69	3634.86	3637.86	3637.17	N/A
Scripted Total Execution	4491.44	3635.57	3635.56	3638.38	3638.03	N/A

Table 4.10: Checkers cluster GAFolder benchmarks: All the results (in seconds)

The GAFolder VAA's total execution times are comparable with the host execution time of the GAFolder application (Figure 4.4). We can only conclude that GAFolder computations need at least 256 MB of VM memory to have reasonable performance. In our benchmarks, GAFolder VAA performed better than the host execution of GAFolder VAA except 128 MB of VM memory size. We cannot explain the reason behind this unusual performance with our benchmarks and it is out of this thesis's scope. However, the variety of research in the field explains the reasons as CPU instructions, memory management and input/output optimizations. For example, Adams et. al [2] discuss the CPU instructions optimizations by performing nanobenchmarks.

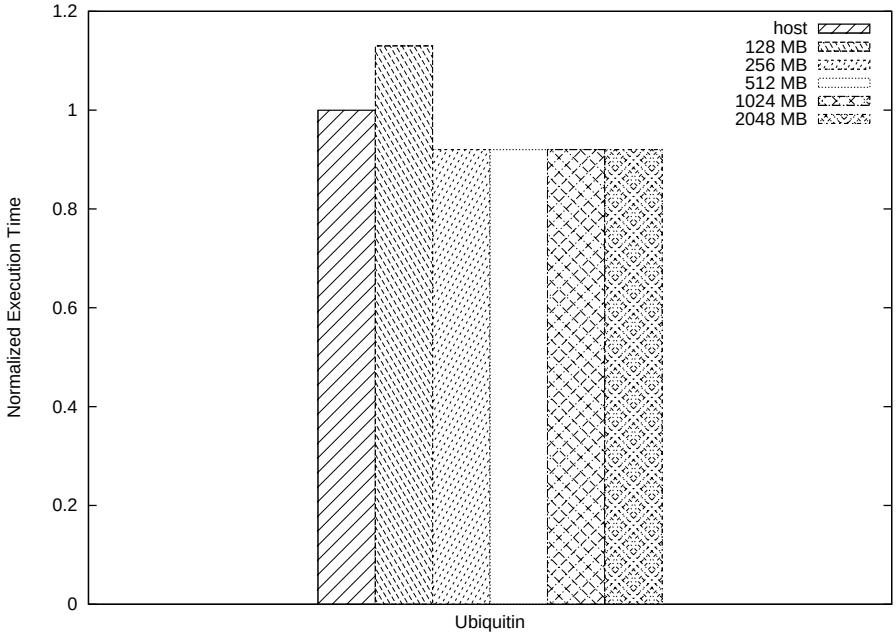


Figure 4.4: Total execution time of the GAFolder VAA for different VM memory sizes normalized to the host execution time of the GAFolder application. The GAFolder VAA's total execution times are comparable with the host execution time of the GAFolder application (Table 4.10).

Other VAA related overheads namely the total boot up/shutdown times and the data movement

overheads are not significant with respect to the total VAA execution times. Table 4.11 shows that the GAFolder VAA's compressed input and output file sizes are less than 120 KB, therefore the data movement overheads are less than 3 seconds (Figure 4.5). Also, the VM boot up/shutdown times are similar to the GROMACS VAA's totals and do not add significant overheads to the total execution times of the VAA .

Protein Name	Input Files Total Size	Output Files Total Size	Compressed Input File Size	Compressed Output File Size
Ubiquitin	240 KB	648 KB	39 KB	118 KB

Table 4.11: Total input and output file sizes of the GAFolder benchmarks (Compressed/Uncompressed)

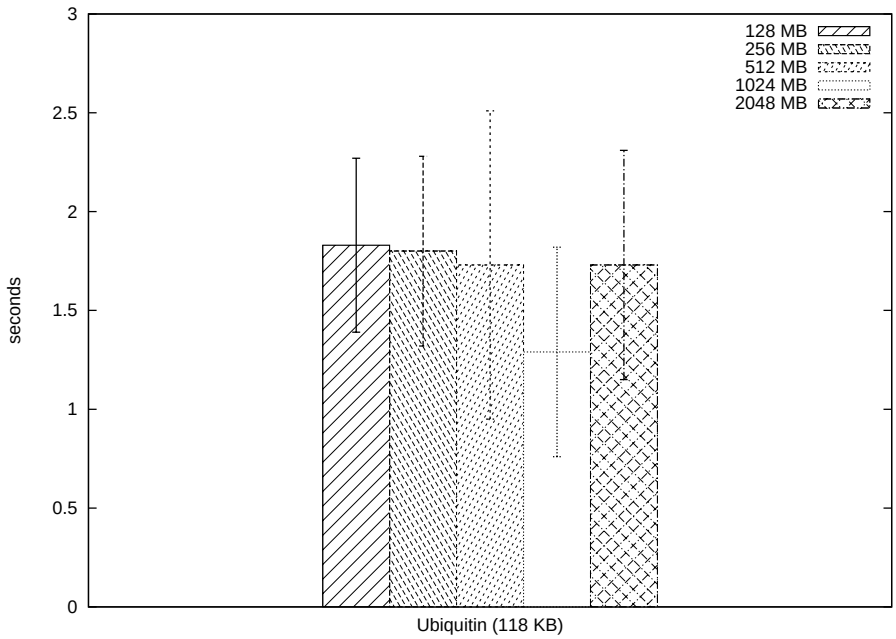


Figure 4.5: Data movement overhead of the GAFolder VAA for different VM memory sizes (The number inside the parentheses shows the compressed output file size which affects the most of the data movement overhead). The data movement overheads of the GAFolder VAA are small with respect to total GAFolder VAA execution times which are more than 59 minutes (Table 4.10).

4.4.3 HMMer VAA Benchmarks

Globin protein is analyzed by the HMMer VAA and the host installation of HMMer. The simulation time depends on the database size, therefore, we choose 2.1 GB database from the BBS benchmark suite to achieve a comparable run-time with the real life applications. All the results of the HMMer benchmarks for the SCN mechanism can be examined from Table 4.12.

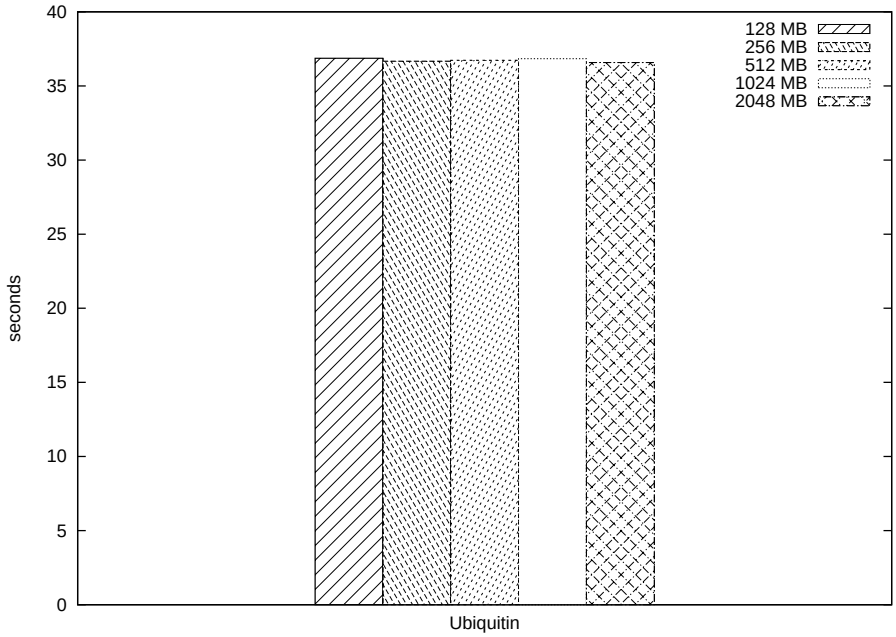


Figure 4.6: Sum of boot up and shutdown times of the GAFolder VAA for different VM memory sizes. The boot up/shutdown overheads of the GAFolder VAA are small with respect to total GAFolder VAA execution times which are more than 59 minutes (Table 4.10).

HMMer Globin	**128 MB**	**256 MB**	**512 MB**	**1024 MB**	**2048 MB**	**Host**
Application Execution	3252.14	3157.12	3159.06	3158.92	3159.23	3156.3
Data Movement Overhead	1.95	1.57	1.1	1.08	0.62	N/A
VM start/stop Overhead	37.83	38.46	42.43	40.07	38.43	N/A
Total VAA Execution	3291.92	3197.15	3202.59	3200.06	3198.28	N/A
Scripted Total Execution	3293.05	3198.39	3203.35	3200.82	3198.57	N/A

Table 4.12: Checkers cluster HMMer benchmarks: All the results (in seconds)

HMMer VAA execution times are higher than the host execution time of HMMer (Figure 4.7). However, the overhead is at most 4.4% in 128 MB VM memory tests. For a middle ground 512 MB memory size, the overhead is as low as 1.47% which makes it negligible with respect to the total search time. This benchmark shows that the I/O intensive applications can also have reasonable performance inside the VM.

Our results are different from Macdonell et al.'s identical benchmark results [32]. Denoted as the *hmmer-with-nr-1CPU* benchmark in their paper, they report that this HMMer benchmark incurs 7.7% overhead on average with respect to the host execution time of the same benchmark. They only use 2 GB of VM memory configured with Gentoo Linux kernel version 2.6. With the same amount of memory, we measure only 1.33% overhead. The main reason of this difference is that Macdonell et al. choose VMware server as the VMM which has different code base than the KVM VMM's code base. Other reasons may include some significant differences in our VAA design from Macdonell et al.'s VA design. For example, we use minimal OS optimized for the VMs, however,

Macdonell et al. use a standard Linux distribution.

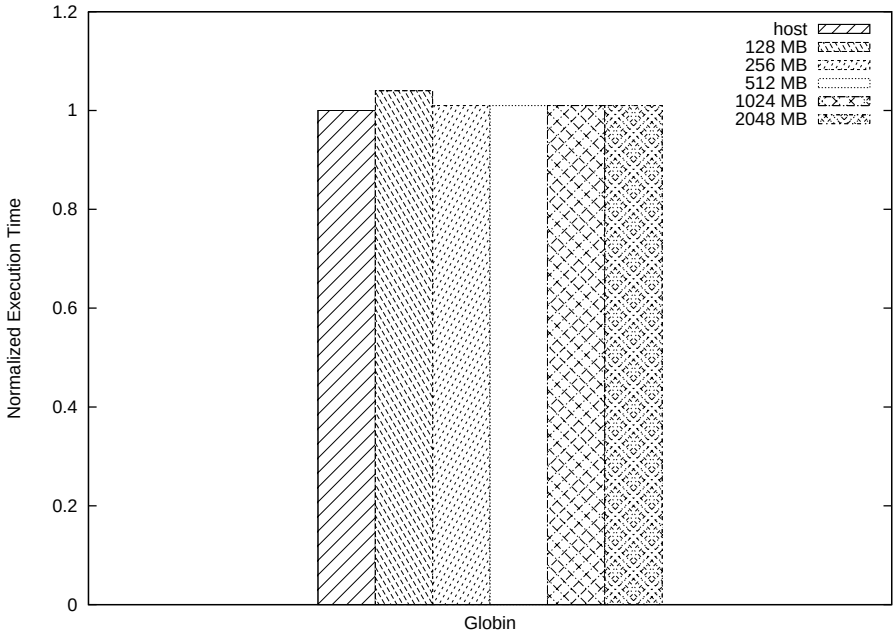


Figure 4.7: Total execution time of the HMMer VAA for different VM memory sizes normalized to the host execution time of the HMMer application. The I/O intensive applications can also have reasonable performance inside the VAA (Table 4.12).

Similar to the GAFolder VAA benchmarks, other VAA related overheads of HMMer VAA, namely the total boot up/shutdown and data movement overheads are not significant with respect to the total VAA execution times. Table 4.13 shows that HMMer VAA's compressed input and output file sizes are less than 330 KB, therefore the data movement overheads are less than 2.5 seconds (Figure 4.8). One important point is that the HMMer VAA's data movement overhead is reducing as the memory size grows which we do not encounter in other benchmarks. Also, the VM boot up/shutdown times are similar to the GROMACS VAA's and GAFolder VAA's totals and do not add significant overheads to the total execution time of the VAA . Therefore, we can conclude that, with the similar software structure other than the main application and the auxiliary applications, the total boot up/shutdown times are fairly constant for each VAA execution in all the benchmarks.

Protein Name	**Input Files Total Size**	**Output Files Total Size**	**Compressed Input File Size**	**Compressed Output File Size**
Globin	76 KB	2092 KB	12 KB	327 KB

Table 4.13: Total input and output file sizes of the HMMer Benchmarks (Compressed/Uncompressed)

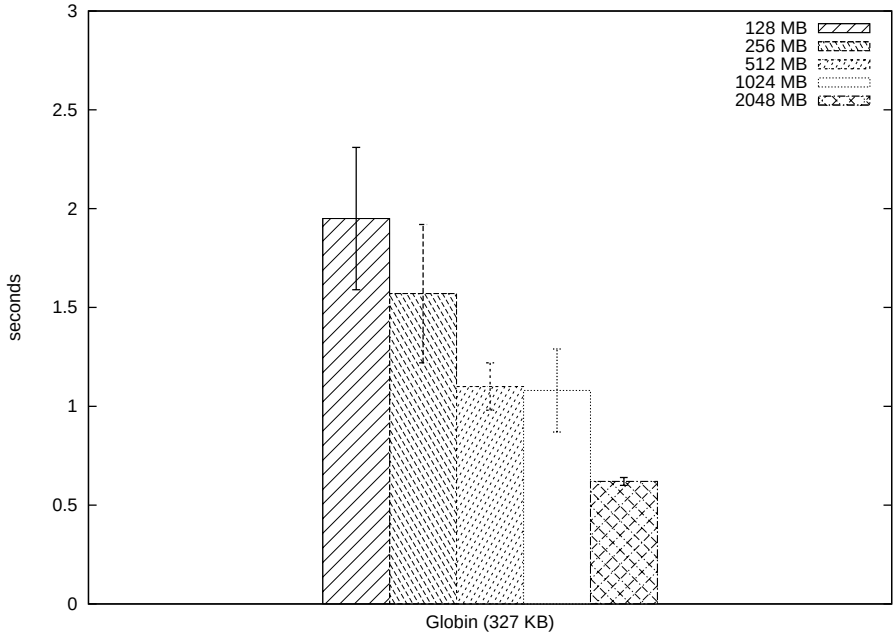


Figure 4.8: Data movement overhead of the HMMer VAA for different VM memory sizes (The number inside the parentheses shows the compressed output file size which affects the most of the data movement overhead). The data movement overheads of the HMMer VAA are small with respect to total HMMer VAA execution times which are more than 52 minutes (Table 4.12).

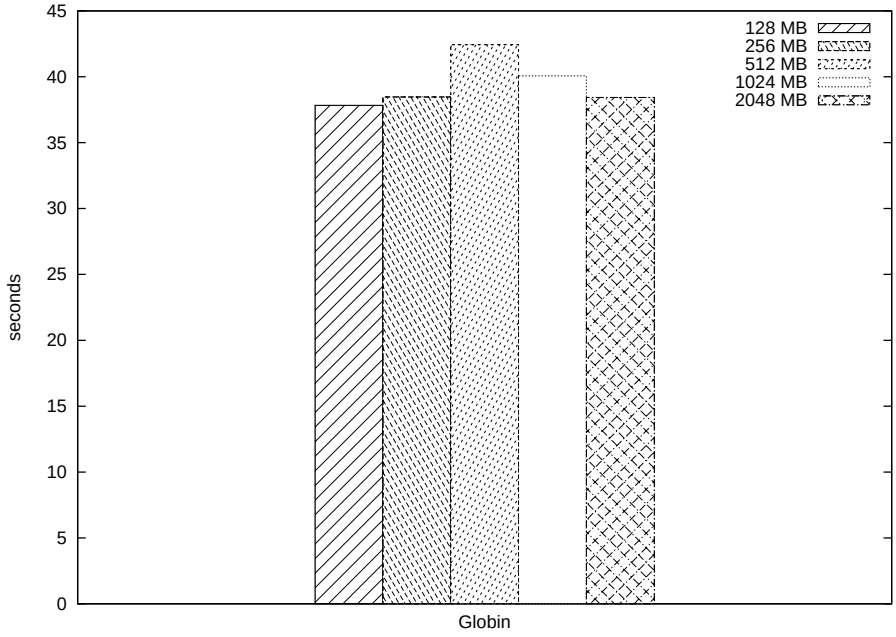


Figure 4.9: Sum of boot up and shutdown times of the HMMer VAA for different VM memory sizes. The boot up/shutdown overheads of the HMMer VAA are small with respect to total HMMer VAA execution times which are more than 52 minutes (Table 4.12).

4.5 Copy over Shared Memory Benchmarks

In these benchmarks, we compare the data movement overhead of the CSM mechanism with the SCN data movement mechanism (Table 4.14). In all the benchmarks, we use the same VAAs as in the SCN benchmarks. We only change the VM Script's SSH-based data movement parts to the CSM operations as explained in Chapter 3. Also, we add the shared memory kernel module to the guest OS and start the VAA with this driver enabled.

VAA and Protein Name (Data Size)	128 MB	256 MB	512 MB	1024 MB	2048 MB	SCN 512MB
GROMACS Chicken	16.08	16.21	16.20	16.15	16.10	32.41
GROMACS Turtle	14.20	14.66	14.49	14.23	14.51	26.51
GROMACS Human	15.60	15.87	15.16	15.17	15.15	28.24
HMMer Globin	0.14	0.14	0.14	0.15	0.22	7.49
GAFolder Ubiquitin	0.09	0.08	0.08	0.09	0.48	8.28

Table 4.14: CSM vs SCN data movement overheads: All the results (in seconds)

We execute the same benchmarks as in the SCN benchmarks except we reduce the GROMACS benchmark's run-time by reducing the simulation time parameter to 50 ps (This change also affects the file sizes (Table 4.15)). By reducing the run-time, we aim to get results faster for the GROMACS VAA since we only measure the difference between SCN and CSM. We also execute the benchmarks of SCN data movement mechanism with 512 MB of VM memory on Botha to compare the performance of our data movement mechanisms. We use the cluster nodes and batch scheduler with the same way that we use in the SCN benchmarks. Every protein's performance numbers are the average of 5 runs for each VM memory size.

Protein Name	Input Files Total Size	Output Files Total Size	Compressed Input File Size	Compressed Output File Size
Turtle	1000 KB	177 MB	189 KB	70 MB
Human	964 KB	189 MB	175 KB	74 MB
Chicken	1064 KB	198 MB	193 KB	81 MB

Table 4.15: Total input and output file sizes of the GROMACS 50 ps benchmarks (Compressed/Uncompressed)

A shared memory file is created with 512 MB size. One important point of these benchmarks is that due to the bug in the Macdonell's shared memory code for KVM, the shared memory size is added to the total memory size. Therefore, the results should be examined with additional 512 MB of VM memory. However, differentiation in memory is not significantly effective in the CSM's data movement overhead.

Our aim in these benchmarks is to show that if the complete local execution is possible the CSM mechanism is a better choice than the SCN mechanism. Therefore, we only measured the data movement overhead of both data movement mechanisms. We also calculated the standard deviations.

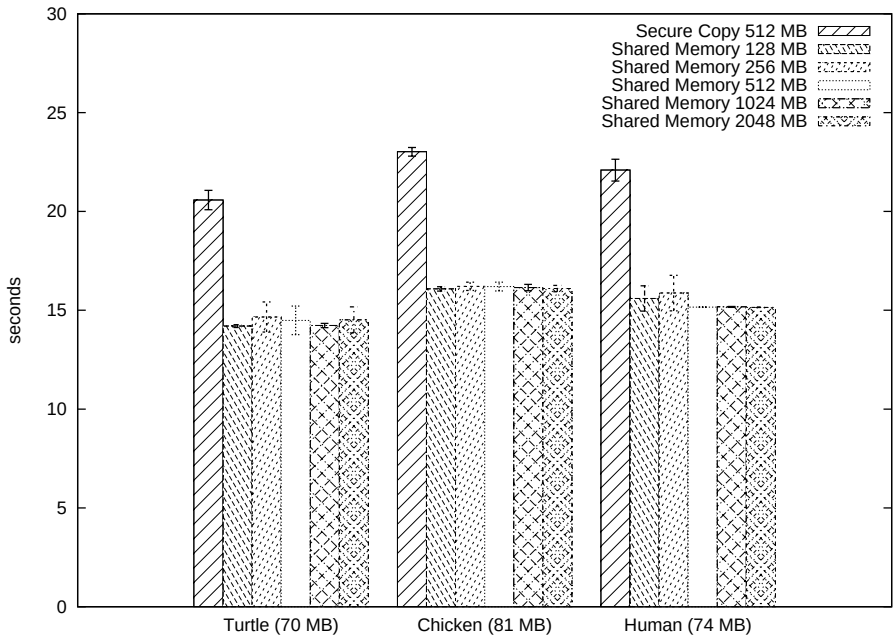


Figure 4.10: Data movement overhead of the GROMACS VAA with the CSM mechanism for different VM memory sizes vs. data movement overhead of the GROMACS VAA with the SCN mechanism for 512 MB VM memory size (The numbers inside the parentheses show the compressed output file sizes which affect the most of the data movement overhead). The CSM mechanism performs better than the SCN mechanism (Table 4.14).

Figures 4.10, 4.11 and 4.12 show the GROMACS, GAFolder and HMMer VAAs' CSM performance with respect to the SCN performance. We conclude that CSM performs at least 30% better than the SCN in all the applications. Also, we speculate that for small data transfers (Figures 4.11 and 4.12), SCN overhead is dominated by the SSH's authentication overhead which is not part of CSM. Every SSH connection starts with the key exchange protocol of SSH, which guarantees that the sender has enough credentials to initiate a data transfer to the receiver [48], [6]. Therefore, this authentication overhead is independent from the data size. We also speculate that the per byte overheads of TCP/IP and SSH encryption protocols are the prominent reasons of the performance degradation during the data transfer in the SCN mechanism.

4.6 Migration Benchmarks

We explore the migration as a mechanism to recover after failovers and avoid the cluster wall-time limit (Chapter 2). The VMM can save several states of the VAA and use one of these saved states to return to the previous execution point in case a failure happens. Also, the migration functionality can be a workaround to the cluster's wall-time limit. After the wall-time has run out, the batch scheduler terminates the user's job. However, if the user's job is longer than the wall-time, by using the migration functionality, the VMM can save the current state of the application before the wall-time has run out and resubmit it to the batch scheduler. Then, the user job can be restored from this

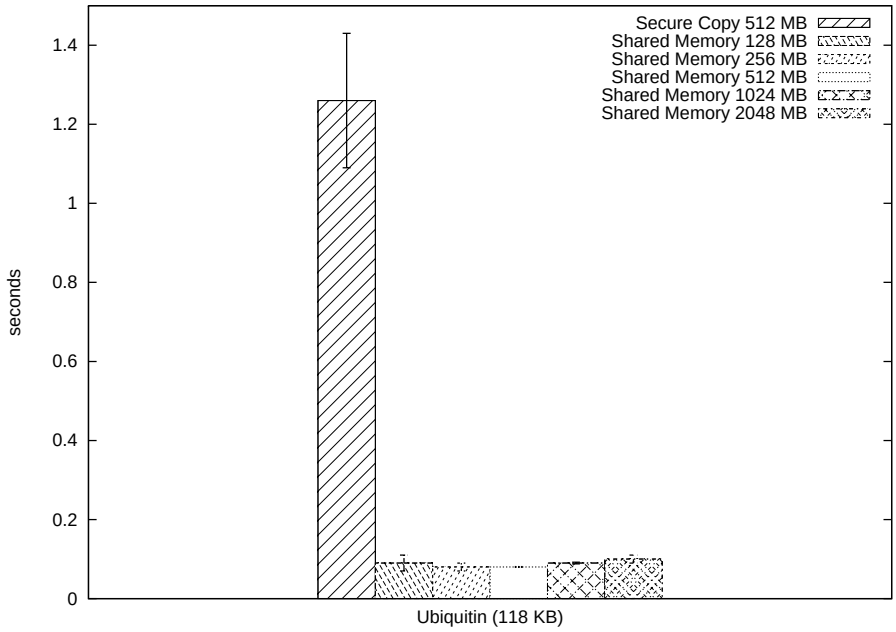


Figure 4.11: Data movement overhead of the GAFolder VAA with the CSM mechanism for different VM memory sizes vs. data movement overhead of the GAFolder VAA with the SCN mechanism for 512 MB VM memory size (The number inside the parentheses shows the compressed output file size which affects the most of the data movement overhead). For small data transfers, SCN is dominated by the SSH's authentication overhead which is not part of CSM (Table 4.14).

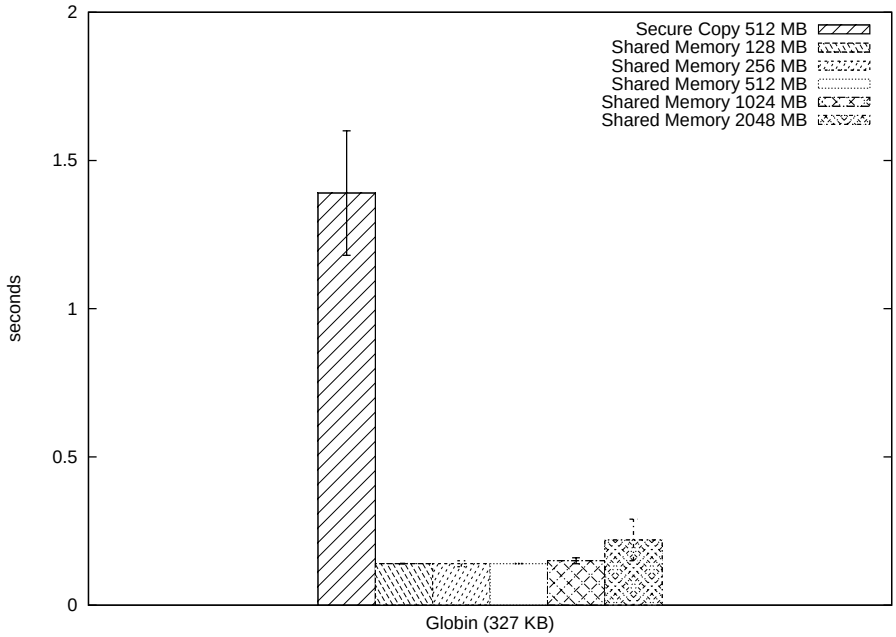


Figure 4.12: Data movement overhead of the HMMer VAA with the CSM mechanism for different VM memory sizes vs. data movement overhead of the HMMer VAA with the SCN mechanism for 512 MB VM memory size (The number inside the parentheses shows the compressed output file size which affects the most of the data movement overhead). For small data transfers, SCN is dominated by the SSH's authentication overhead which is not part of CSM (Table 4.14).

saved state and application execution continues.

In this section, we investigate if migration's two important overheads of save and restore are negligible enough to employ the migration functionality. The migration benchmarks measure the time spent for saving the VM state to a file and restoring the VM from this saved state. We use *libvirt* [29] library and its command line tool *virsh* for the save and restore operations. *libvirt* provides a virtualization API that supports several VMM platforms such as KVM and Xen. *virsh* provides several command-line options to execute the VM-related operations such as starting the VM, shutting down the VM, saving the VM state, restoring the VM from the VM state file and defining resource requirements.

We use the GROMACS, GAFolder and HMMer VAAs with 512 MB of memory. Although we execute the VAAs until completion, we only measure the save and restore times. We check that, at the end of the execution, the resultant files have correct results and the same as the non-migration executions. We execute the applications until half of their run-times to ensure that their CPU and memory usages are steady. Then, we issue the save command and save the state file to the disk. (Figure 4.13). Finally, we restore the VAA from this saved state and let the execution end (Figure 4.14). The operations are executed by submitting the VAA jobs to the batch scheduler.

```
virsh -c qemu:///session save GAFolderVAA /home/user/GAFolderVAA.save
```

Figure 4.13: The *virsh* command-line for the save operation

```
virsh -c qemu:///session restore /home/user/GAFolderVAA.save
```

Figure 4.14: The *virsh* command-line for the restore operation

One source of confusion may come from the terminology of suspend and resume versus save and restore. We use save and restore because *libvirt* uses these terms to define the operations that we execute for the non-live migration. However, instead of save and restore, some papers in the literature may refer to the same operations as suspend and resume. Therefore, they can be used interchangeably depending on the context.

The save operation creates a save file (i.e. state file) and then destroys the VM process. For example, in Figure 4.13, *virsh* saves the state of the *GAFolderVAA* VM to the */home/user/GAFolderVAA.save* file. The save file contains the resource definitions, the CPU and memory state of the VM and *libvirt*-related headers. After the save operation, the saved file can be moved locally anywhere on the host, however, the VM disk images should stay in their original paths.

The restore operation restores the VM from the state file that is created by the save operation. For example, in Figure 4.14, *virsh* restores the VM from the */home/user/GAFolderVAA.save* state file. The restore operation redirects the contents of the state file to the KVM monitor's pseudo-terminal. Therefore, KVM reads the state file from the monitor's pseudo-terminal and starts the VM from the

point that the VM is saved. In these benchmarks, the time measurement for the restore phase is until completion of *virsh*'s restore command. We test that, right after the restore command returns, we can connect to the VAA and the VAA is responsive to the user requests. Therefore, the restore times in this section guarantees the usability of the VAA after the restore operation.

All the results of migration benchmarks are shown in Figure 4.15 and Table 4.16. We conclude that as the size of the state file grows, the time spent for the save and restore operations grows. Also, even with the large state files, e.g. HMMer's 425 MB state file, that is close to the 512 MB memory size the save time is approximately 15 seconds and the restore time is approximately 5 seconds. Therefore, the migration overheads are also negligible with respect to the total execution times of the VAAs.

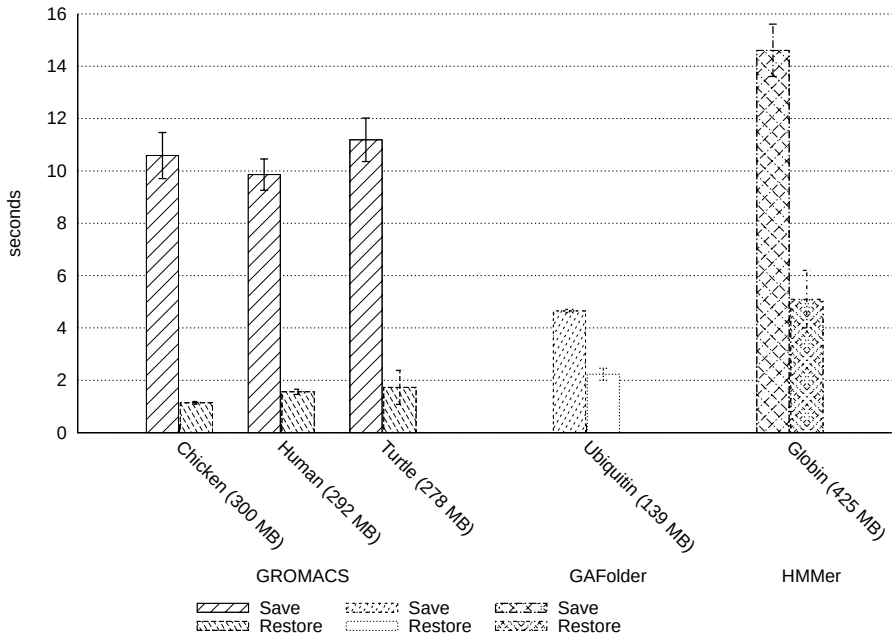


Figure 4.15: All migration benchmarks' results (The numbers inside the parentheses show the sizes of the saved state files). The migration overheads are small relative to total execution times of the VAAs (Table 4.16).

VAA and Protein Name	Save	Restore
GROMACS Chicken	10.59	1.14
GROMACS Turtle	11.19	1.73
GROMACS Human	9.86	1.56
HMMer Globin	14.61	5.1
GAFolder Ubiquitin	4.65	2.23

Table 4.16: Migration overheads: All the results (in seconds) (See also Figure 4.15)

4.7 Concluding Remarks

In this chapter, we evaluated the performance impacts of our VAA design. The general conclusions and quantitative evidence from empirical evaluations are summarized in Table 4.17. We performed three sets of benchmarks to understand the overheads caused by the VAA execution, data movement mechanisms and migration operations. We varied the VM memory sizes to examine the effect of the memory size on the performance of the VAA. We also compared the VAA execution times of the applications with their host execution times. We used the data files and simulation times that are similar to the real life applications. We executed the applications and the VAAs on the real clusters.

Through the first set of benchmarks, we saw that our SCN data movement mechanism incurs negligible overhead with respect to the total execution times of the VAAs. Further, we concluded that the total VAA execution times of the applications are very close to the host execution times of the applications. Furthermore, we saw that even the I/O intensive application of our test suite achieves near-native performance.

The second set of benchmarks were used to verify that our CSM data movement mechanism is a better choice when the complete local execution is possible. We saw that the CSM approach achieved much better results. Also, we speculated that for small data transfers, SCN is dominated by the SSH's authentication overhead which is not part of CSM.

The third set of benchmarks were performed to examine the migration overheads of the VAAs. We measured the time spent for the save and restore operations. We concluded that as the state file size grows the total time spent for the save and restore operations grows. However, even if with a 425 MB file size that is close to the 512 MB memory size of the VAA, the overhead number is in seconds and negligible with respect to the total execution times of the VAAs.

Conclusion	**Figures**	**Raw Data Tables**
The VAAs can achieve near-native performance	4.1, 4.4, 4.7	4.7, 4.8, 4.9, 4.10, 4.12
The data movement overheads are small relative to total VAA execution times	4.2, 4.5, 4.8	4.7, 4.8, 4.9, 4.10, 4.12
The VAA boot up/shutdown times are small relative to total VAA execution times	4.3, 4.6, 4.9	4.7, 4.8, 4.9, 4.10, 4.12
CSM performs better than SCN	4.10, 4.11, 4.12	4.14
For small data transfers, SCN is dominated by the SSH's authentication overhead which is not part of CSM	4.11, 4.12	4.14
The migration overheads are small relative to total execution times of the VAAs	4.15	4.16

Table 4.17: The general conclusions and quantitative evidence from empirical evaluations

Chapter 5

Related Work

Our project contributes to the research in HPC mainly in three categories: the management of clusters, adaptation of the VMs to the cluster environment and migration of VMs for easy control of the application execution. Therefore, we can discuss the related work in three categories. The first category is *Aggregated Resource Management* that focuses on managing the VMs and/or applications in the distributed systems. The second category is *Virtual Appliances* which includes the research that is related to the virtual appliances and their uses. The third category is *Migration* which presents papers from the VM migration research. The following sections discuss the related work in the above-mentioned categories.

5.1 Aggregated Resource Management

Aggregated resource management tools control the creation and/or distribution of resources. Also, these tools coordinate the applications that use these resources across the distributed systems. In this dissertation's context, we focus on the tools that adapt the VMs into the cluster environment, which we call as *Virtual Cluster Implementations*; and tools that use non-VM mechanisms to manage the resources and applications to deal with the software heterogeneity problem. In these categories, the tool can be the part of a bigger project such as Globus Virtual Workspaces [24] or the tool itself can be the center of other management components such as in Condor [30] and In-VIGO [1].

The virtual cluster implementations use the VMs as the primary unit for the management of clusters. These tools aim to integrate the VMs into the distributed systems and optimize the cluster environment for the efficient execution of the VMs. They generally virtualize all the physical components of the distributed system such as the memory, CPU and network interfaces. When we use the verb virtualize, it means we provide virtual versions of physical components to the software in the context of the VMs rather than exposing physical resources to the software as they are. From the cluster management side, virtualization gives flexibility in the management and distribution of resources.

One of the virtual cluster implementations is In-VIGO which tries to virtualize entire cluster

environment by creating a virtual cluster model and virtual interfaces. The virtual cluster model consists of virtualized versions of file systems, applications (In-VIGO's virtual applications are very similar to the virtual appliances) and networks. In-VIGO's architecture also includes a resource manager which handles the creation of virtual resources and execution of user jobs using these virtual resources. Further, In-VIGO aims to be user-friendly with its web-based user interface manager for the creation of the VMs and the management of the user jobs. Furthermore, In-VIGO can benefit from VMPlants [27] which dynamically creates virtual resources when they are needed. VMPlants gathers the information about the necessary resources from the user via a web interface. This information can be the type of OS, applications to be installed or resource definitions such as the configuration of networking with specific MAC/IP addresses, the CPU type and memory size. In-VIGO's virtual file system (VFS) implementation stays on top of NFS and redirects the RPC calls from the client to a NFS server. As a result, In-VIGO provides NFS-like file system features in a virtualized environment. The In-VIGO's authentication mechanism can use a standard password file of UNIX, LDAP or SQL database. For authorization, In-VIGO relies on user classes and application access constraints, which can also be defined by the user.

Similar to In-VIGO, the Globus Toolkit 4 [15] introduces Globus Virtual Workspaces as part of the Globus Toolkit's service-oriented architecture for the distributed computing. Services under Virtual Workspaces configure the VM as a virtual workspace (VW), deploy it to the cluster and define the operations such as starting and stopping of the VMs. In Virtual Workspaces, a VW is managed by a web service and the definition of VW is depicted in the form of an XML schema. Virtual Workspaces heavily uses other Globus Technologies such as GSI with X.509 certificates [8] for the security and GridFTP [3] for data movement.

A different implementation with similar features as In-VIGO and Virtual Workspaces is Virtual Cluster Installation System (VCIS) [19]. VCIS provides an installation request submission system that installs the necessary software and also allows the configuration of the software inside the VM. Further, VCIS uses caching techniques to accelerate the multiple VM creation.

Virtual Cluster Implementations use the VMs to deal with the software heterogeneity problem; however, BOINC [4] and Condor use different techniques for the same purpose. BOINC and Condor distribute the data and computation across the clients and aim to use the redundant cycles of various computer systems in this distributed environment. However, their approach differs in terms of handling the target systems and management of resources. BOINC targets the public computers connected via Internet across the globe and uses them as application execution platforms. BOINC users can benefit from several tools to describe the data and computations as well as to create and execute the applications in the BOINC installed computers. Condor, however, does not require special programs to be designed for the distributed environment. Condor simply aggregates several workstations and controls them with a scheduler. The user prepares a text based submission script to define the hardware and software requirements of a user job. Then, Condor chooses the best suitable exe-

cution platform from a pool of workstations. Unlike BOINC, Condor targets the computer systems across the LAN. However, Condor can also be used in the public computing. If a specific workstation becomes busy, Condor can migrate the jobs if possible or stops and restarts the jobs in another workstation. To achieve this type of migration, Condor uses the I/O redirection, checkpoint/restart and transparent process migration mechanisms. Further, Condor supports the VMs but this support is limited to locating the necessary VMM to run the VMs with their resource descriptions.

The feature comparison of the aggregated resource management tools and our design can be examined in Table 5.1. We compare our work with the virtual cluster implementations (VCIs) since we use the VMs. The main difference between our design and the VCIs is that the VCI's resource management application has to be installed on the cluster before the VCI can perform its functions. However, we benefit from the shell scripts, C programs and widely used batch schedulers. Therefore, we eliminate the need for the non-standard management software. Although they have some mechanisms for automated data movement, they leave data movement decisions mostly to the user who needs to be aware of the VMs. However, data movement back and forth from the original location is an extra burden for the user. The user also wants the output files to be organized in the predetermined locations automatically. Our secure data movement mechanisms provide this flexibility without a user interaction with the VM. Additionally, we argue that dynamic VM-creation features of the above-mentioned implementations do not add extra advantage for the user who needs extensive customization of the software environment. The best case for the user is to have her workstation or test environment virtualized. Therefore, even if the user has a VM with a compatible OS and base software installation, she has to configure the VM for her own purposes. Further, one common advantage of the dynamic VM-creation is to simplify the creation of multiple VM images, but this can also be done more efficiently by the static VM-creation. For example, after the user tests her work in one VM and decides to deploy it, she can simply copy it to multiple nodes with a simple script and run it across the cluster. Also, KVM has a snapshot option which basically executes the VM in the read-only mode and any modifications to the VM are written to temporary files (i.e. copy on write). After the execution finishes, all the changes are destroyed and the base image stays the same. With the KVM's snapshot option, the user can execute multiple copies of the same VM without any change to the base VM image; therefore the user does not need multiple copies of the same VM if the image stays on a shared file system.

5.2 Virtual Appliances

A VAA refers to a special kind of virtual appliance (VA) that runs to produce a result and stop execution after the result. We further optimized the VAAs for scientific applications with data movement mechanisms and a security infrastructure. However, the VAs are generally functional units for different distributed system management solutions. In this section, we see a complex VA-based solution to address the software heterogeneity problem and a benchmark that shows the performance impacts

Features	In-VIGO	Virtual Workspaces	BOINC	Condor	VAA
Solution for the software heterogeneity problem	Virtualization with VMware and IBM zVM	Virtualization with Xen and VMware	Plug-in architecture	I/O redirection	Virtualization with KVM
VM Creation	Dynamic	Dynamic	N/A	N/A	Static
Security - Authentication	The user manager (LDAP, passwd file and/or external SQL database)	GSI (X.509 certificates)	The BOINC manager (password authentication)	GSI, kerberos, windows, anonymous etc.	SSH public/private key
Security - Authorization	The user manager (Role-based)	GSI (Map files)	None or the project specific	User-based	Standard linux authorization and forced command restrictions
Security - Data Movement	Data movement across trusted sources, no encryption	GridFTP (GSS API extensions)	None or project specific	Optional encryption	SSH encryption
User Data movement	Manual data movement to a NFS-mounted folder	Optional automatic stage-in stage out	Automated stage-in stage-out	Automated data movement	Automated stage-in stage-out

Table 5.1: Feature comparison of aggregated resource management applications

of the VAs.

The virtual appliance concept first appears in a series of papers from Sapuntzakis et al. as the main unit of the Collective project [41], [9], [40]. The Collective project aims to achieve homogeneous software environment from heterogeneous sources via a collection of VAs. From this point of view, the main motivation of the Collective project is similar to our motivation. However, the Collective project is intended to simplify the deployment of software in general. Also, the Collective researchers add maintenance features as part of a VA solution. Unlike our model, which uses available resources across a distributed system, the Collective project proposes a new way of software deployment concept. The Collective user has different VAs for different purposes such as a VA for a firewall software, VA specialized for office applications and VA specialized for a communication application. However, the users do not install the application to the VAs and, whenever they want to use the VA, they get the up to date version of the appliance from the centralized repository. Similar to our architecture, in the Collective architecture, there are separate VM disks for each appliance, which are the data and OS VM disks. However, whenever the user wants to access a VA, she sees a unified environment. Also, the VAs of the user can communicate with each other across the network. Some of the collective system's other features are a language called CVL to define the VAs and its resource requirements, authentication system for the security and user specific management and caching mechanism for reducing the amount of data transfer over the network.

From the performance point of view, Macdonell et al. [32] provide several benchmark results with VMware-based VAAs[1]. Macdonell et al. use the BBS benchmark suite for the BLAST and HMMer VAA benchmarks. For the Gromacs VAA benchmarks, they use Gromacs benchmarking system called *gmxbench*. According to their *vmstat* results, the BLAST benchmark varies in its I/O intensity. The HMMer benchmark is a database search benchmark which is I/O intensive. GROMACS benchmarks mainly run *mdrun* from the GROMACS application suite which is compute-intensive. They found that compute-intensive jobs incur 6% or less overhead and I/O intensive ones incur 5.6% or more overhead on average. However, our results with GROMACS and HMMer (with different test sets) show a KVM-based VAA with minimal operating system can achieve near-native performance in both I/O and compute-intensive jobs (Chapter 4). The reasons may include the differences in the VMM software and minimal OS of the VM.

5.3 Migration

VM migration mechanisms can be divided in two categories: the live (online) or non-live (offline) migration. In the live migration approach, the VMs can be migrated without interrupting the VM execution. The VM state is copied to another machine while the VM is still running on its current machine. Once all the state is copied to the target machine, the identical VM on the current host is

[1] In their paper, Macdonell et al. use the term Virtual Appliances, however, we define the type of VAs in their paper as Virtual Application Appliances.

suspended and then, the modified part of the state during this suspension phase is also copied to the target host if it exists. At this point, the VM process is destroyed on the current host. Finally, the VM is restored on the target machine and continue to the execution on the new host. In the non-live migration approach, the VM state is saved to a file and then the VM process is destroyed. Later, the VM state is copied to the target machine and restored from this machine.

Non-live migration is also referred as save-copy-restore or suspend-copy-resume migration. The precedents of the non-live migration methods appears in process migration techniques which we also discuss in this section. In our design, we assume a resubmission to the scheduler; therefore, the VAA job has to be destroyed before the restore operation. If the execution can be initiated from the shared file system, there is no need for the copy phase. Although we used the term the VM state here, the dominant part of VM migration is the memory migration and others such as the migration of the network interfaces and disks depends on the configuration of the execution environment.

Clark et al. [10] proposes a pre-copy based live migration technique. In this technique, the mechanism for copying the VM state differs in the memory and local resources such as the VM disks and network interfaces. To migrate memory, Clark et al.'s mechanism iteratively copies the memory pages from one machine to another in rounds. In every round, the mechanism copies the modified pages from the previous round. To migrate the local resources, Clark et al. relies on some assumptions for the execution environment since they do not target WANs. For the networking, they simply migrate the IP address of the VM to the target host. The assumption of this IP migration is that the host and target machines are in the same LAN. For storage migration, Clark et al. assume that the execution environment has a shared file system between the host and the target system; therefore, Clark et al. do not address the migrating of the VM disks from one machine to another. This technique is integrated to Xen.

Similar pre-copy live migration approaches are also integrated into the hypervisors from KVM and VMware. KVM uses a similar iterative process and relies on the same assumptions about the local resources with Xen [25]. VMware in its *VMotion* technique adapts the pre-copy live migration approach [34]. For the migration of IP addresses and network interfaces *VMotion* relies on VMware ESX Server's virtual networking architecture. *VMotion*, similar to Xen, assumes a shared file system between the target and the host machines.

The pre-copy approach tries to minimize the downtime while the restore phase occurs on the target machine. However, since the VM on the host runs during this time, significant amount of modified pages may be resent to the target machine if the application is memory-intensive. Also, after suspending the original VM, the size of the modified pages to be transferred could be significantly high which increases downtime of the VM. Hines et al. propose a post-copy based method to overcome the limitations of the pre-copy approach in memory-intensive applications [20]. This post-copy method first copies the minimal CPU state by suspending the VM. Then, their mechanism resumes the VM and starts to copy the memory pages. Hines et al. uses 4 techniques to reduce the

downtime and pages that are resent. *demand paging* and *active push* ensures that each page is copied only once to the target machine, *pre-paging* tries to understand VM's page access patterns to reduce the VM's resume time. *Dynamic Self Balloning* prevents the migration of the unallocated pages to reduce the number of pages transferred.

Although the live migration techniques have all these advantages, they are not suitable to our design because we assume that the batch schedulers are in charge of the resource management on the clusters. Therefore, we can not assume that the batch schedulers are aware of the VM live migration as well as specific VMM's live migration capability. In fact, the main batch scheduler that we use during our experiments is TORQUE and it lacks this capability. However, the non-live migration can be achieved by just resubmitting the saved VAA to the batch scheduler that eliminates the necessity for a migration-aware batch scheduler. Consequently, we explored a non-live migration technique by using KVM's migration mechanisms with *virsh* tool from the *libvirt* library (Section 4.6).

Non-live migration techniques moves the VM state in three steps: save the VM state to a file, copy the state file to the target host (if necessary) and restore the VM from the state file. These techniques try to reduce the size of the image with a set of methods to minimize the copy overhead across the network. Generally, they are adapted for the WAN and non-shared file systems. An example of a non-live migration technique is the Collective's migration technique [42]. Collective uses a non-live migration technique because it targets low bandwidth networks across the WAN to distribute the VAs. The internet suspend/resume [43] and denali [47] projects also propose non-live VM migration techniques.

Previously studied process migration techniques are preliminaries of today's VM migration techniques. Condor's transparent process migration method is independent of the OS, however, the MOSIX OS's [22] and Sprite OS's [12] process migration techniques are integrated to the operating system. The OSes like Sprite and MOSIX are especially designed for the distributed systems. Therefore, these OSes aim to achieve load balancing and easy management of the processes across the distributed systems by the process migration. Although the process migration techniques get considerable attention from the research community, they do not gain many practical uses. The main reason is that the relative complexity of implementing these techniques due to the process dependencies such as the open file descriptors and local resources [10]. However, if we migrate the VMs then most of these dependencies are also migrated across the cluster, which reduce the overheads and failures.

5.4 Concluding Remarks

In this chapter, we reviewed some previous work in the fields of aggregated resource management for the distributed systems, virtual appliances and VM migration. We pointed that the presented resource managers install a non-standard management software to perform their functions. However,

we integrate the widely-used applications, shell scripts and C programs to achieve similar goals. We argued that the dynamic VM-creation does not add extra advantage for the users who need extensive customization for their VMs. Another advantage of the dynamic VM creation is to simplify the multiple VM creation, however, the dynamic VM-creation can not be simpler than just copying the VM to all the nodes of the cluster. In a shared file system, we stated a better solution that is the KVM's snapshot feature, which uses copy-on-write mechanism on multiple VM executions from the same base image. Finally, we stated the reason of choosing a non-live migration technique as easy adaptation to the batch schedulers that are not aware of the live VM migration.

Chapter 6

Concluding Remarks

We presented the design and implementation of our virtual application appliance (VAA) solution to the software heterogeneity problem on clusters. Our main design goal is to create a special VM, named as a VAA, for only the execution of a specific scientific application. Our VAA design puts together the main scientific software and its auxiliary applications along with the compatible OS and necessary libraries. We integrated widely available applications on the clusters with shell scripts and C programs to make the VAA execution transparent from the user. Hence, the user can run the VAA as she runs a regular application.

We scripted the execution of the VAA to make VM-related operations transparent from the user, so that, the user does not need to have VM knowledge. The user only needs to know how to execute the application not the VM-related operations such as data movement, authentication and authorization. As long as the user runs the application from the folder that the input files reside, our scripts execute the VAA and, at the end, put the output files, standard output and error contents back to the current folder. Our scripts also automate the job submission to a batch scheduler by hard-coding the necessary resource requirements for the VAA. For example, let us say the user stores the input files in */home/input*. The only requirement for the user is to go to */home/input* folder and submit her work by using our submission script with necessary arguments for her application. Furthermore, we do not restrict the user on how to use the VAA. The user can create her own scripts related to the main application execution such as analysis scripts and execute inside the VAA.

A key decision in the design of this work is the mechanism to move the data in and out of the VM automatically and securely because the VAAs provide extra layer of indirection and the VAAs and data may be in different administrative domains. We implemented two stage-in stage-out data movement mechanisms which are called the secure copy over network (SCN) and the copy over shared memory (CSM). In the SCN mechanism, data movement is established over network. Also, we used the SSH's public/private key authentication and SSH's forced command feature to establish the security of data movement. In the CSM mechanism, the VAA reads from a shared memory region and writes to the shared memory region. Therefore, the only security concern, authorization, is handled by the file access rights of the OS. SCN is the only choice for the remote VAA executions, which

files have to be copied securely over the network between different administrative domains. CSM, however, is a better choice only for the local VAA executions to eliminate the network overhead and the SSH-related authentication and encryption overheads.

After our main VAA design and implementation, we explored VM migration. VM migration allows the user to save several states of the VAA and use one of these saved states to restore the VAA in case a failure happens. Also, VM migration can be used for the resubmission to the batch scheduler after the wall-time is exceeded. If the run-time of the user's job is longer than the wall-time, the VAA state can be saved at the end of the wall-time. Then, the user or an external script can resubmit the job back to the batch scheduler. This script can restore the VAA from this state without any data loss.

We have evaluated the performance impacts of our mechanisms with widely used bioinformatics applications called GROMACS, HMMer and GAFolder. We aimed to find whether the overheads of our mechanisms and VMs are negligible enough to use the VAAs effectively on the clusters. We found that the data movement overhead and the VM boot up and shutdown overheads are negligible enough with respect to total run-times of these applications. When we compared the total VAA execution time with the host execution time of the same application, we noticed that the VAAs achieved near-native performance. Further, we saw that the CSM data movement mechanism performs at least 30% better than the SCN data movement mechanism. Also, the migration tests showed that even if more than 400 MB state has to be saved and restored, saving the VM took at most tens of seconds and restoring the VM took at most a few seconds.

Bibliography

[1] Sumalatha Adabala, Vineet Chadha, Puneet Chawla, Renato Figueiredo, Jose Fortes, Ivan Krsul, Andrea Matsunaga, Mauricio Tsugawa, Jian Zhang, Ming Zhao, Liping Zhu, and Xiaomin Zhu. From virtualized resources to virtual computing grids: the in-vigo system. *Future Gener. Comput. Syst.*, 21(6):896–909, June 2005.

[2] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM.

[3] W. Allcock. Gridftp: Protocol extensions to ftp for the grid. *Global Grid Forum*, 2003.

[4] D. P. Anderson. Boinc: a system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10, 2004.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[6] Daniel J. Barrett, Richard E. Silverman, and Robert G. Byrnes. *SSH, the Secure Shell: The Definitive Guide*. O'Reilly Media, Inc., 2005.

[7] H. J. C. Berendsen, D. Van Der Spoel, and R. Van Drunen. Gromacs: A message-passing parallel molecular dynamics implementation. *Comp. Phys. Comm*, 91:43–56, 1995.

[8] R. Butler, V. Welch, D. Engert, I. Foster, S. Tuecke, J. Volmer, and C. Kesselman. A national-scale authentication infrastructure. *Computer*, 33(12):60–66, 2000.

[9] Ramesh Chra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam. The collective: A cache-based system management architecture. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation(NSDI'05)*, pages 1–11, May 2005.

[10] Christopher Clark, Keir Fraser, and H. Steven. Live migration of virtual machines. In *NSDI '05*, pages 1–11, 2005.

[11] Mike Closson and Paul Lu. Bridging local and wide area networks for overlay distributed file systems. In *Second Workshop on Real, Large Distributed Systems*, pages 49–54, 2005.

[12] Fred Douglis and John Ousterhout. Transparent process migration: design alternatives and the sprite implementation. *Softw. Pract. Exper.*, 21(8):757–785, August 1991.

[13] Kris Buytaert et al. *The Best Damn Server Virtualization Book Period: Including Vmware, Xen, and Microsoft Virtual Server*. Syngress Publishing, Burlington, MA, 2007.

[14] National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov/.

[15] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779*, pages 2–13, 2005.

[16] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer*, 1974.

[17] Western Canada Research Grid. http://www.westgrid.ca/.

[18] University Of Alberta Prion Research Group. http://www.cs.ualberta.ca/ prion.

[19] Satoshi Matsuoka Hideo Nishimura, Naoya Maruyama. Virtual clusters on the fly - fast, scalable, and flexible installation. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 549–556, Rio de Janeiro, Brazil, May 2007. IEEE.

[20] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 51–60, New York, NY, USA, 2009. ACM.

[21] HMMer. http://www.hmmer.org/.

[22] Lau F.C. Ho R.S.C., Cho-Li Wang. Lightweight process migration and memory prefetching in openmosix. In *Proceedings of IPDPS*, 2008.

[23] VMware Inc. http://www.vmware.com/.

[24] Kate Keahey. Virtual workspaces in the grid. *EuroPar 2005*, September 2005.

[25] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, pages 37–42, 1996.

[26] Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 21–21, Berkeley, CA, USA, 2005. USENIX Association.

[27] Ivan Krsul, Arijit Ganguly, Jian Zhang, Jose A. B. Fortes, and Renato J. Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2004. IEEE Computer Society.

[28] Perl Language. http://www.perl.com.

[29] Libvirt. http://libvirt.org.

[30] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, 1988.

[31] Andrew Macdonell. PhD Thesis in progress.

[32] Cam Macdonell and Paul Lu. Pragmatics of virtual machines for high-performance computing: A quantitative study of basic overheads. In *Proceedings of High Performance Computing & Simulation Conference (HPCS'07)*, 2007.

[33] Microsoft. http://www.microsoft.com.

[34] Michael Nelson, Beng H. Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, page 25, Berkeley, CA, USA, 2005. USENIX Association.

[35] Jeremy Nickurack. MSc Thesis in progress.

[36] Padala Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, and Kang G. Shin. Performance evaluation of virtualization technologies for server consolidation. Technical report.

[37] Jordan Patterson. Jole: a library for dynamic job-level parallel workloads. Master's thesis, University of Alberta, October 2009.

[38] Christopher Pinchak, Paul Lu, Jonathan Schaeffer, and Mark Goldenberg A. The canadian internetworked scientific supercomputer. In *In 17th Annual International Symposium on High Performance Computing Systems and Applications (HPCS*, pages 193–199, 2003.

[39] Trellis Project. http://www.cs.ualberta.ca/ paullu/Trellis/.

[40] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Virtual appliances for deploying and maintaining software. In *LISA '03: Proceedings of the 17th USENIX conference on System administration*, pages 181–194, Berkeley, CA, USA, 2003. USENIX Association.

[41] Constantine Sapuntzakis and Monica S. Lam. Virtual appliances in the collective: A road to hassle-free computing. In *Proceedings of the 9th conference on Hot Topics in Operating Systems(HOTOS'03)*, pages 55–60, Berkeley, CA, USA, May 2003. USENIX Association.

[42] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[43] Mahadev Satyanarayanan, Benjamin Gilbert, Matt Toups, Niraj Tolia, Ajay Surie, David R. O'Hallaron, Adam Wolbach, Jan Harkes, Adrian Perrig, David J. Farber, Michael A. Kozuch, Casey J. Helfrich, Partho Nath, and Andres. Pervasive personal computing in an internet suspend/resume system. *IEEE Internet Computing*, 11(2):16–25, 2007.

[44] Bioinformatics Benchmark System. http://www.bioinformatics.org/.

[45] G. van Rossum. Python tutorial, technical report cs-r9526 centrum voor wiskunde en informatica (cwi), May 1995.

[46] VMware Virtual Appliances. `http://www.vmware.com/vmtn/`.

[47] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, 2002.

[48] Tatu Ylonen. Ssh - secure login connections over the internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, 1996.