

VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing

Ivan Krsul, Arijit Ganguly, Jian Zhang,
José A. B. Fortes, Renato J. Figueiredo

Advanced Computing and Information Systems (ACIS) Laboratory
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL

Abstract

Virtual machines provide flexible, powerful execution environments for Grid computing, offering isolation and security mechanisms complementary to operating systems, customization and encapsulation of entire application environments, and support for legacy applications. This paper describes a Grid service – VMPlant – that provides for automated configuration and creation of flexible VMs that, once configured to meet application needs, can then subsequently be copied (“cloned”) and dynamically instantiated to provide homogeneous execution environments across distributed Grid resources. In combination with complementary middleware for user, data and resource management, the functionality enabled by VMPlant allows for problem-solving environments to deliver Grid applications to users with unprecedented flexibility. VMPlant supports a graph-based model for the definition of customized VM configuration actions; partial graph matching, VM state storage and “cloning” for efficient creation. This paper presents the VMPlant architecture, describes a prototype implementation of the service, and presents an analysis of its performance.

1 INTRODUCTION

The need for flexible, efficient sharing of distributed resources is at the core of the design and implementation of computational “Grids” [11]. While existing Grid environments typically support resource sharing by relying on mechanisms found in operating systems, the use of virtual machines (VMs [13]) as the basis for resource sharing can provide improved resource security and user isolation, customized execution environments, and simplified resource administration [10]. To realize these advantages, it is key that the functionality to manage virtual resources that are dynamically created and destroyed be provided by middleware. This entails support for flexible and automatic VM environment customization from high-level user specifications, the ability to “clone” and instantiate a VM environment efficiently, and to monitor VM instances at run-time. This paper presents the VMPlant middleware for flexible VM customization and efficient cloning through a service-oriented architecture.

VMPlant handles virtual machine creation and hosting for “classic” virtual machines (e.g. VMware [23]) and user-mode Linux platforms (e.g. UML [8]) via dynamic cloning, instantiation and configuration. In combination with a virtual machine “shop” service (described in Section 3.1), VMPlants deployed across physical resources of a site allow clients (users and/or middleware acting on their behalf) to instantiate and control client-customized virtual execution environments. The plant can be integrated with virtual networking techniques (such as VNET [24]) to allow client-side network management.

The key design goals for the architecture described in this paper are configuration flexibility, support for multiple virtualization techniques and fast VM instantiation, scalability, resilience to failures, and interoperability. Flexibility is achieved via the use of directed acyclic graphs (DAGs) to describe actions to configure and customize a VM, and a VM cloning process that can be applied to various VM technologies. It is architected such that the process of virtual machine management is not tied to a particular middleware solution, and follows Web and Grid service frameworks [12][18] for interoperability. Fast instantiation is achieved via a cloning mechanism that: 1) allows for partial matching of configuration DAGs, and 2) leverages techniques available in existing hosted VMs for committing changes to virtual disks and file systems at the end of a session. Results show that efficient cloning allows a VMware-based VMPlant prototype to achieve VM creation in 17 to 85 seconds.

The VMPlant architecture enables Grid users and resource providers to use services that are unique to a VM-based approach to distributed computing. Users can define customized execution environments (where Grid applications and their preferred environments are encapsulated), which can then be archived, copied, shared (with other users) and instantiated as multiple run-time clones. Resource providers can decouple the configuration of their physical machines from that of user's execution environments and associated idiosyncrasies; administration becomes simpler by requiring from providers only the underlying support for virtualization and the VMPlant service. In combination with complementary middleware for virtual networking and user, data and resource management, the functionality enabled by VMPlant allows for problem-solving environments to deliver Grid applications to users with unprecedented flexibility.

For instance, in the context of In-VIGO [1], it complements virtual applications, file systems, and a Web-based interface to enable the dynamic instantiation of "virtual workspaces", where users have access to a VM for managing their files through a Web-based graphical user interface that presents a full-fledged X11 session to the user via VNC [21]. On-going work considers the use of the described services to implement mechanisms whereby virtual workspaces allow users to install customized applications (including unmodified binaries) and make them available to other users and collaborators via the In-VIGO portal. Furthermore, once defined, such an application-centric environment can be instantiated without requiring the installation of applications in the physical resources, facilitating application deployment across an existing setup, and also when new resources are added.

The rest of this paper is organized as follows. Section 2 presents motivations for the design of a service for the management of virtual machine Grid execution environments. Section 3 describes the architecture of VMPlant, and Section 3.2 describes in detail its components. Section 4 presents results from a performance analysis of a VMPlant prototype that consider the latencies involved in dynamic VM creation and configuration. Section 5 discusses related work, and Section 6 presents conclusions.

2 VIRTUAL MACHINES FOR GRID COMPUTING

"Classic" VMs [23][13] present the image of a dedicated operating system while enabling multiple O/S configurations – completely isolated from each other – to share a single machine. This is an effective mechanism for resource consolidation, and a key reason for the renewed interest and popularity of VMs. They also provide a flexible, powerful execution environment for Grid computing, offering isolation and security mechanisms complementary to operating systems, customization and encapsulation of entire application environments, and support for legacy applications [10]. The software layer introduced by virtual machine monitors allows users to access consistent, customized application environments that are decoupled from physical systems administered by resource providers, addressing a fundamental goal of Grid computing – flexible resource sharing [11].

A "classic" design has the flexibility of supporting O/Ss of different types, and the efficiency of supporting VM time-sharing of a physical resource. It is not, however, the only virtualization technique with potential usefulness for Grid computing. Techniques including user-level O/S translation [8][15], para-virtualized VMs [25][3], and inexpensive machines that can be remote-booted and "scrubbed" [2][7] can deliver systems that may lack the flexibility and time-sharing of classic VM environments but are desirable for cost and/or performance reasons.

Supporting such a variety of VMs in a Grid environment poses a challenge of dealing with VM-specific idiosyncrasies. Nonetheless, while different VM technologies present different interfaces for their configuration and control, core mechanisms on top of which middleware services can be layered are identifiable. First, VM environments can be encapsulated as data (e.g. VMware virtual disks, UML and Xen file-systems, z/VM minidisks, and COD [7] configuration databases). Second, instantiation can be implemented by a control process (e.g. via VMware host O/S APIs; UML host scripts and virtual console; Xen's "domain 0"; a z/VM Linux machine with communication channels established with the VMM; and COD remote boot "trampolines"). The VMPlant architecture, as described in the following sections, builds upon these mechanisms to support an application-centric model for the management of various types of Grid VMs.

3 ARCHITECTURE

The overall VMPlant architecture is depicted in Figure 1. It is based on services designed to operate within Web and Grid frameworks [12][18]. The client (user or middleware) interacts with the service via a

front-end virtual machine “shop”. The service can use standard mechanisms for dynamic or static discovery (e.g. Universal Description, Discovery and Integration – UDDI) and for obtaining the service’s binding and location description (e.g. as a Web Service Description Language document). It can then use messaging mechanisms (e.g. Simple Object Access Protocol – SOAP) to request the creation of a new virtual machine instance, and to query or collect (destroy) an active VM instance.

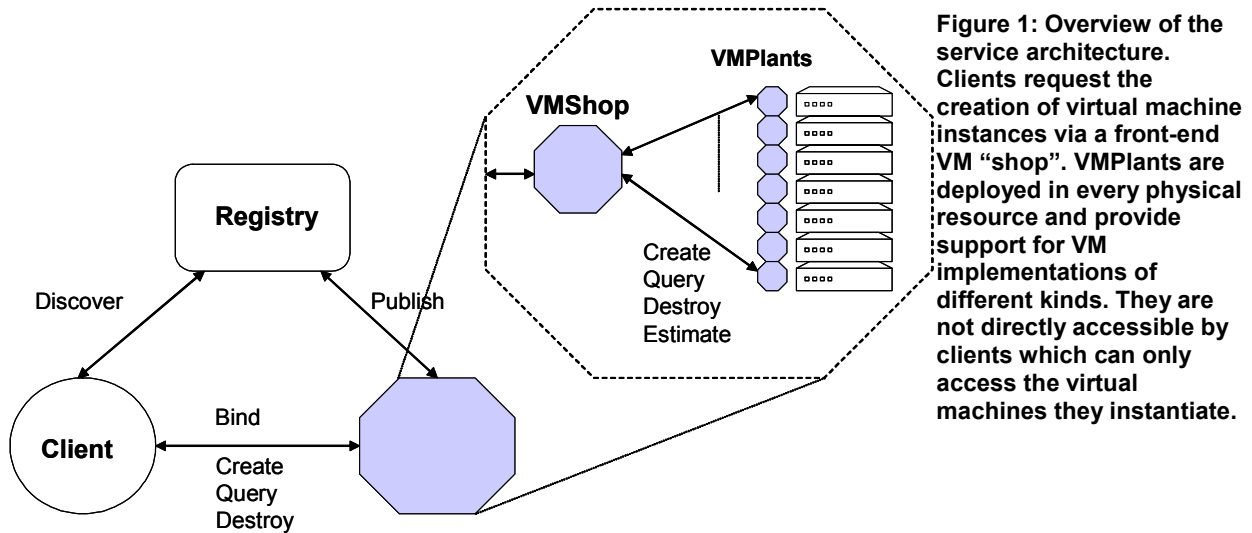


Figure 1: Overview of the service architecture. Clients request the creation of virtual machine instances via a front-end VM “shop”. VMPlants are deployed in every physical resource and provide support for VM implementations of different kinds. They are not directly accessible by clients which can only access the virtual machines they instantiate.

A key motivation for the service-oriented approach presented in this paper is the leveraging of emerging standards and implementations that can facilitate its integration and interoperability with other systems. The focus of this paper, however, is not on the underlying mechanisms to support Web or Grid services, such as messaging, discovery, publishing, authentication, or lifetime management. Rather, the focus is on the mechanisms that are particular to virtual machine management – namely, on techniques that allow flexible application-centric specification of VM configurations and efficient run-time VM instantiation – and on a model that allows provisioning of these mechanisms as services.

3.1 VMShop

From the client’s perspective, the VMShop service performs tasks analogous to those handled by system administrators to accommodate user requests for new computational resources in a network. For example, typical tasks of a system administrator include: securing access to resources that meet cost and specification constraints (hardware and software, such as memory size and O/S configurations) through acquisition or reuse; configuring networks; providing users with resource access information (host and user names); and also handling of queries about the status of the resource.

Similarly, VMShop provides a single logical point of contact for clients to request three core services: create a VM instance, query information about an active VM instance, and destroy (collect) an active VM instance. Requests for virtual machine creation received by VMShop contain specifications of hardware, network and software configurations. Hardware specifications are used to determine appropriate resources that match requirements such as the VM’s instruction set, memory and disk space, while software specifications are used to configure the VM once it is started. The latter process requires a flexible means for users to provide configuration information: it may involve a number of actions that are specific to a VM instance (e.g. installation of libraries, setup of user identities and machine credentials, application startup) and that may require action ordering and error handling.

For flexible configuration, the architecture uses direct acyclic graph (DAG) representations to allow clients to specify software configuration specifications for the creation process. The DAG represents configuration actions by nodes, and ordering is established by directed edges. Except for special start/end nodes, action nodes of the configuration DAG may encounter errors during configuration. For

error handling, a special error node is implicitly associated with each action node, and the client can also explicitly configure custom error-handling sub-graphs for action nodes. Nodes in the configuration DAG may be associated with actions to be performed within a virtual machine's "guest" (e.g. setup of a user account) or by a virtual machine's "host" (e.g. setup of a virtual device, such as a CD-ROM ISO image or a network interface card).

Configuration DAGs are generated by clients and are sent as part of XML-based VM creation service requests. If creation is successful, the client obtains in return a classad [20] with (attribute,value) pairs storing information that includes a VMShop-assigned unique identifier for the virtual machine (VMID), as well as configuration-specific data resulting from the output of action DAG nodes. The classad returned to the client allows it to request query and collection through the service, and also to access an active VM's "guest" directly, for instance with physical or virtual IP network addresses and SSH keys or X.509/GSI certificates setup during its creation.

VMShop is responsible for selecting a VMPlant for the creation of a virtual machine. This process is implemented through a communication API and a binding protocol that allows VMShop to request and collect bids containing estimated VM creation costs from VMPlants (directly, or indirectly through VMBrokers). The protocol uses XML-based requests and allows a VMShop to select from multiple plants one that can satisfy a request based on its analysis of response bids. Costs are generically represented as numbers; a variety of models can be conceived to associate costs with resource utilization policies.

The classad of an active virtual machine is maintained by its corresponding VMPlant, but it is not part of the state that needs to be maintained by VMShop, thus facilitating service restoration in the presence of failures. VMShop may, however, cache classad information in the information system to speed up queries and bidding requests.

3.2 VMPlant

The VMPlant implements a Production Process Planner (PPP, Figure 2) to plan the process of VM instantiation, based on the DAG specification of a virtual machine. In addition to supporting flexibility of VM configuration, the DAG aids the implementation of an efficient VM creation process by supporting partial matches of cached VM images. When the PPP receives a production order, it searches the VM Warehouse to find a suitable match – a "golden" machine. The golden machine must match the client machine specification in terms of memory, disk, the operating system installed and (fully or partially) the

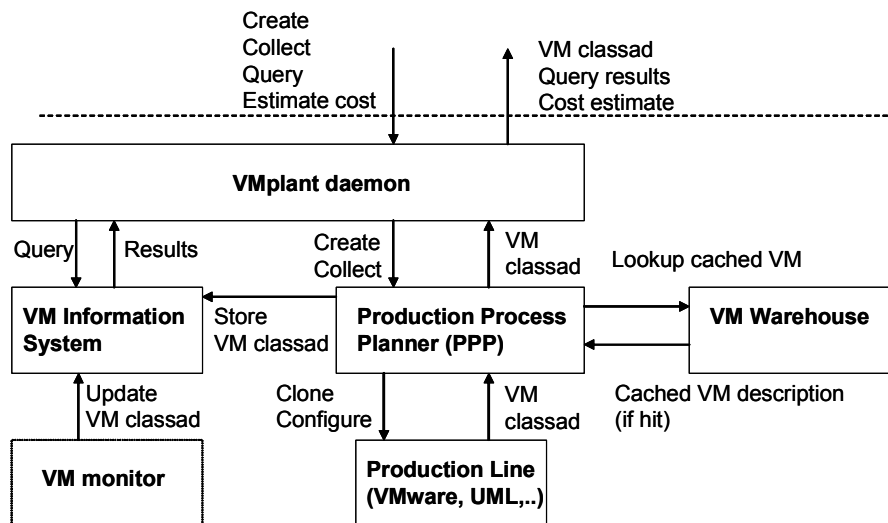


Figure 2: Internal architecture of a VMPlant. The PPP matches a creation request specification against VMs available in the warehouse. The production line controls processes for "cloning" and configuring a machine. The VM information system maintains state about currently active machines (including dynamic information gathered by a VM monitor)/

DAG configuration actions. The DAG enforces a partial order between the actions; the cached image may also have some configuration actions performed on it. The matching criterion requires these operations not to conflict with those required on the requested machine in terms of the partial order and the type of operation. Figure 3 illustrates an example of the matching process in context of In-VIGO virtual workspace. For each cached image, the following conditions are checked:

- **Subset Test:** The operations that have been performed on the cached image are only a subset of those required on the requested machine. This means that the cached image should not have any operation performed on it that is not required by the requested machine.

- **Prefix Test:** The operations that have been performed on the cached machine must only be a prefix of the configuration DAG for the desired virtual machine. If an operation “A” in the DAG is preceded by a number of other operations, the cached image should have “A” performed on it only if it has all the preceding operations performed.

- **Partial Order Test:** The operations that have been performed on the cached image obey the partial order as specified by the configuration DAG for the requested machine. If the DAG specifies operation “A” to be performed before another operation “B”, and the cached machine has both these operations performed, they should obey the order required by the DAG.

Once a golden machine has been found, the PPP requests the VM Production Line to clone the machine, and then parses the DAG to perform a series of configuration actions on the new machine. The configuration actions are a set of commands that have to be executed on the guest machine. It uses the Production Line to execute these scripts inside the guest machine. Once a machine is created, the configuration process returns a classad describing the machine, which is then stored into the VM Information System maintained by the VMPlant.

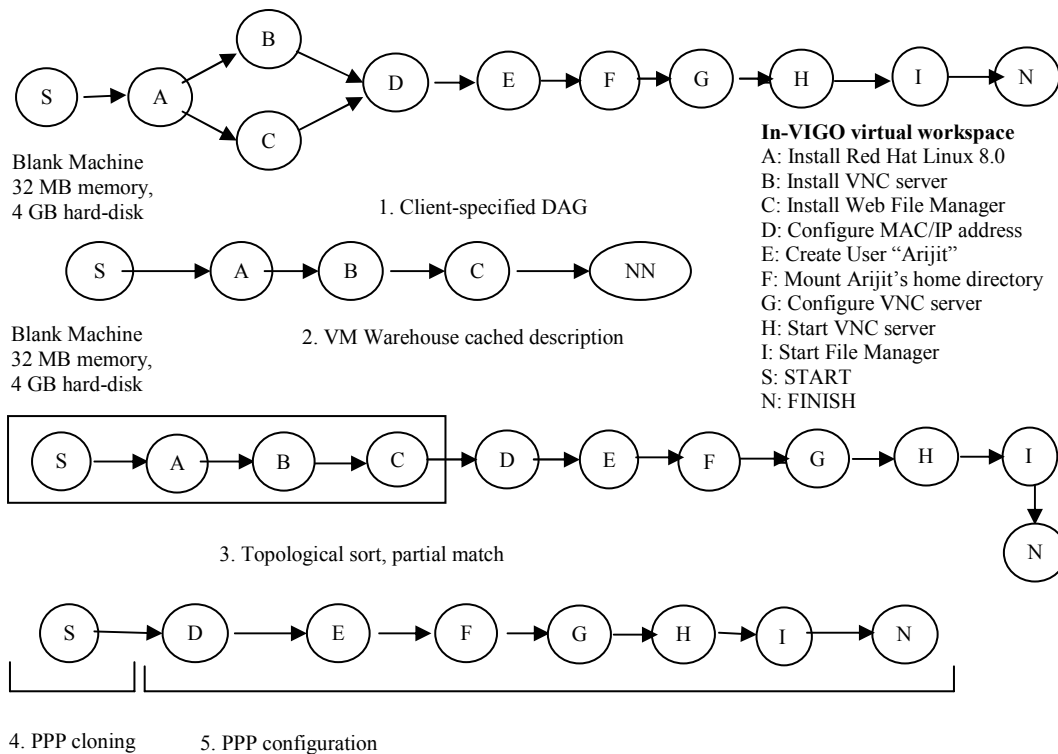


Figure 3: DAG matching process. The client-specified configuration DAG (1) matches a VM warehouse cached description (2) once it is topologically sorted (3). The process of creating the VM is then coordinated by the PPP and includes cloning of the cached sub-graph (4) and configuration scripts for the remaining sub-graph (5).

The process of cloning and configuration is thus based on the copying of a machine’s state from its golden image to the cloned image, instantiation of the machine and execution of configuration actions. The copying process contributes significantly to the creation time, especially when large amounts of state (e.g. from disk or suspended memory) must be handled. The VMPlant can efficiently clone VMs that support storage commits at the end of a session (e.g. non-persistent VMware disks, copy-on-write UML file systems) through the use of links rather than file copies.

The start node of a configuration DAG refers to a “blank” virtual machine with no operating system or software installed. Subsequent nodes define actions to be taken to bring the VM to a desired state. While

it is conceivable to construct actions to control a VMPlant to automate tasks such as an O/S installation from a CD/ISO image, the current implementation supports the off-line definition of “golden” virtual machine images for subsequent cloning and configuration. The VM Warehouse stores “golden” images of not only pre-built images with typical installations of popular operating systems, but also images that are set up and customized for an application by providing VM installers with the capability of publishing a VM image to the Warehouse, for subsequent instantiations through VMPlant. Although the VMPlant can use different implementations of VM-Warehouse, one possibility under on-going investigations is based on virtualized distributed file systems [26].

The mechanisms used by the VMPlant prototype implementation are described in detail in Section 4; for instance, the VMware-based In-VIGO virtual workspace “golden” machine is checkpointed with a setup consisting of Linux RedHat, a VNC server and a Web file manager server. After cloning from this state, the virtual workspace is configured with an IP address and an In-VIGO’s user name and encrypted password, and with a mount point for the user’s distributed virtual file system [9].

3.3 Support for virtual networks

In a realistic deployment scenario, there can be a number of network domains providing VMPlant services, and the client may contact any of these domains for a VM. The client may want to assign to the VM an IP address from its own domain. The client may also want to be able to run software licensed to its domain on the remote VM. Moreover, resource providers also need to isolate the VM from other machines. VMPlant can leverage virtualization techniques developed in the context of Virtuoso/VNET [24] to circumvent these issues. VNET provides a TCP/SSL bridge that operates at the Ethernet layer, and bridges the remote VM to the client’s network. For instance, with VNET it has been possible to run an In-VIGO [1] back-end on a host at Northwestern University, assign it an IP address from a University of Florida domain (and use typical LAN services such as NIS/NFS), and also run licensed software (Matlab) remotely¹. This turns a remote host into a provider of compute cycles, which can be configured and used by a client.

As described in [24], the idea behind VNET is to create the client VM in a host-only network, in order to provide for isolation of other remote hosts and the VM itself. Such host-only networks correspond to statically installed “vmnet” switches for VMware and “tap” devices with a switch daemon for UML [29], which are dynamically assigned to client domains. The assignments must ensure that VMs from different client domains are never created inside the same host-only network. A VNET server runs on each VMPlant, and on a host (called the Proxy) in client domain. The client attaches to its VM request, credentials for uniquely identifying its domain, and also the IP address and port on which the Proxy is running.

The necessary requirements for virtual networking can be encapsulated behind a virtual network service. The front-end VMShop becomes a client to this service, and uses it to dynamically set up and tear down VNET handlers. The administrators within a domain can define local policies for the accessibility of VMPlants. The scenarios that have been considered include VMPlants operating inside a private network and not directly accessible from outside (but only through VMShop running on a Gateway host), or only SSH connections allowed to the VMPlants. An implementation based on the first scenario and using statically established SSH tunnels between public ports on the Gateway and the ports where the VNET servers are running on VMPlants is currently being pursued.

3.4 Cost-Function

As described earlier, a front-end VMShop uses cost-bidding to achieve resource utilization and load balancing. In the current implementation, the focus has been on the compute resource and the host-only network resource. The host-only network resource has to be utilized very judiciously as it imposes a limit on the number of client domains that a VMPlant can service. Two situations that can lead to under utilization of resources are (1) VMPlants running out of host-only networks, but still having compute power for more VMs; and (2) VMPlants running out of compute power, but still having some unutilized host-only

¹ Depending on specific-application licensing policies, applications may or may not be allowed to be executed in this model. If the licensing model is based on network/IP addresses and does not pose constraints on the geographical location of a resource, the described scenario is possible.

networks. The current implementation splits the VM creation cost into “compute cycles cost”, and the “network cost”. The first component is proportional to the number of VMs already operating on the VMPlant, and is an estimate of the load on the plant. The second component is a one-time charge for a host-only network, required only when a free host-only network is allocated to the client domain. Subsequent VMs for the same domain can re-use the allocated host-only network, and do not incur the network cost.

The effectiveness of the cost function becomes clearer from the following illustration, where the “network cost” is 50; and the “compute cycles cost” is 4 *times* the number of VMs operating on the plant. Consider a scenario with two VMPlants A and B, each having 4 host-only networks, and capable of hosting at most 32 client VMs. Initially neither plant has VMs running. When asked to make bids, both report a cost of 50 that corresponds to the network cost. The VMShop picks one plant at random (say A), and instructs it to create the VM. Next time a client from the same domain requests a VM, the plant A reports a cost of 4 (corresponding to the “compute cycles” cost), but the plant B reports a cost of 50 (corresponding to the “network cost”). The VM shop picks plant A, and the new VM is created in the same host-only network. Initially, the network cost on B dominates the computation cost on A, and VMShop keeps picking A for the same client. Eventually, the computation cost on A dominates the network cost on B. This happens when the client has requested as many as 13 VMs from the VMShop and all of them end up being created on the plant. At that point, the shop would pick plant B; assign another host-only network to the client and instantiate a VM.

4 IMPLEMENTATION AND PERFORMANCE ANALYSIS

This section describes the implementation of a prototype that supports the core functionality of VMPlant and presents an analysis of its performance. The goal is to demonstrate feasibility of the design and determine expected times required to create VMs based on the proposed configuration/cloning techniques and existing cluster and VM technologies. The focus of the analysis is on creation times; analyses of the run-time performance of VM systems have been covered in related publications [3][25][23][10].

4.1 Implementation status

In the prototype, the VMShop and the VMPlant have been implemented in Java. The communication between the VMShop clients, the VMShop and the VMPlants is based on Berkeley Sockets and the messages are serializable Java objects. Services requested by VMShop clients are specified as XML strings. The Create VM service specification contains the DAG of configuration actions. The VMShop parses the XML string and converts it into an in-memory Java object. Based on the service type, the VMShop makes service requests to the VMPlants. In the prototype, the bidding protocol uses a cost model that is based on the amount of host memory available for cloned VMs.

The VM Warehouse and the Information System have been implemented using the host file system. Golden machines are stored as files in sub-directories of the VM Warehouse; each golden machine is specified by a configuration file, and virtual disk and memory files. XML files are used to describe such cached images in terms of their memory sizes, operating system installed, and the configuration actions that have already been performed in the cached machines. Such cached images are suspended VMs with non-persistent virtual disks, allowing multiple clones to share the base virtual hard disk of the golden machine (avoiding copying of large files), and write all changes to private (and smaller) redo log files. Hence, the Production Line uses soft links for the virtual hard disk, and replicates the VM configuration file, memory state, and base redo log for each clone.

The Production Line has been implemented for VMware GSX server and UML virtual machines. The VMware Production Line consists of a set of Perl Scripts (with a Java wrapper) for cloning and configuring virtual machines, and uses the Perl API provided by VMware for virtual machine control. The Production Line uses these mechanisms as a basis for executing configuration commands specified by a topologically-sorted configuration DAG. The DAG actions are converted into Perl scripts, and the Production Line writes each such script to one or more CD/ISO images that are then connected to the cloned VM as virtual CD-ROMs. Once a CD-ROM is connected to the guest, a daemon running within the VM mounts the CD-ROM and executes the configuration scripts. Outputs are provided back to the

Production Line, from which error and classad information are extracted and stored in the VM information system.

The UML production line has been implemented in a similar fashion in the way of configuring the virtual machine from a virtual CD-ROM image and sharing read-only Copy-On-Write virtual disk files. The main difference is that the current UML production line boots the virtual machine after cloning, instead of resuming it from a checkpoint. With checkpointing techniques such as SBUML, it is possible to clone virtual machines from the corresponding snapshots and resume them without a full reboot [28].

4.2 Experimental setup

Experiments have been conducted on an 8-node set of an IBM e1350 xSeries Linux-based cluster. Each physical node is configured as follows: dual 2.4GHz Pentium-4, 1.5GB RAM, 18GB SCSI disk. Each node runs a VMPlant that supports cloning and configuration of VMware GSX 2.5.1 (build 5336) virtual machines. The VMShop client runs in a Linux workstation (1.80GHz Pentium-4, 512MB RAM) and the server runs in a cluster node. Experiments consider creation of VMs from “golden” machines configured as follows: Linux Mandrake 8.1 workstation with memory sizes of 32MB, 64MB and 256MB. The VM warehouse is accessible from each cluster node via a network file system (NFS) mount served by a dual Pentium-3 1.8GHz, 1GB RAM, 500GB SCSI RAID5 storage server. All physical host servers run RedHat Linux 7.3. The cluster nodes are interconnected by an Ethernet gigabit switch, and are connected to the NFS server and VMShop by a 100Mbit/s switched Ethernet network.

Golden VMs are cloned, resumed and configured from state checkpointed at a post-boot stage. The configuration includes setup of the VM’s network interface and of a user ID within the VM guest. Experiments consider a series of requests, in sequence, for virtual machine creation through VMShop (128 requests for 32MB and 64MB VMs, and 40 requests for 256MB VMs).

4.3 Results and analysis

Figure 4 shows the distributions of measured end-to-end VM creation times. The normalized occurrence distribution (y axis) is obtained from the creation time recorded for each VM successfully created (121, 124 and 40 VMs with 32MB, 64MB and 256MB, respectively). Two important observations can be drawn from these results: first, the DAG-based configuration and the cloning mechanism allow VMs to be instantiated, on average, in 25 to 48 seconds. Second, the results show that creation times are larger for machines with larger memory sizes.

Creation times are significantly influenced by the mechanisms used by the VMPlant to clone a machine. A closer look at the distribution of cloning times results in the distribution shown in Figure 5. The use of golden images and symbolic links for on-demand access to virtual disk state results in considerably smaller cloning times than explicit copying. For instance, the virtual disk of the golden machine in this experiment occupies 2GBytes of storage (spanned across 16 files) and takes 210 seconds to be fully copied – around 4 times slower than the average cloning time of the 256MB VM. The memory state is currently copied by the VMPlant implementation during cloning, hence the larger creation latencies for VMs with larger memory capacities².

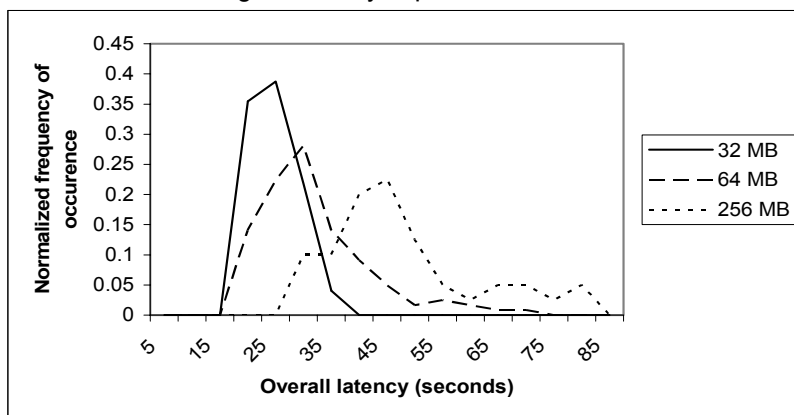


Figure 4. Distribution of overall VM creation latencies for golden machines of different memory sizes. Latencies shown in this graph are measured from client request to VMShop response

² The memory state (*.vmss files) needs to be copied because of an implementation-dependent restriction imposed by VMware GSX and is not fundamental to the cloning approach.

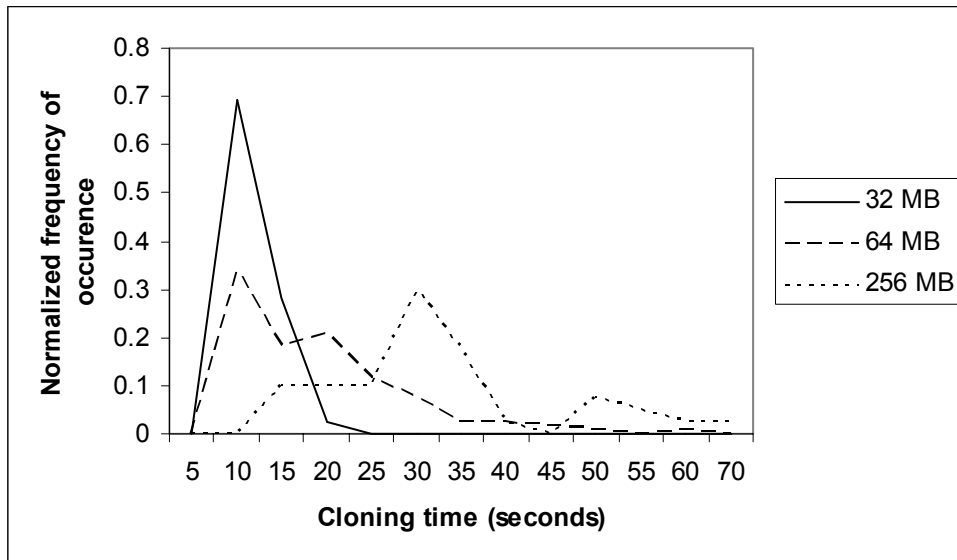


Figure 5: Distribution of VM cloning latencies. Latencies shown are measured from the time the PPP requests cloning to the completion of the VMware resume operation on a cloned machine.

The distribution summarized in Figure 5 shows larger variances in cloning times for the larger-size VMs. A closer look at the profile of cloning time versus the order in which VM clones are requested (Figure 6) shows that cloning times tend to increase when the VMPlant hosts a large number of VMs. This behavior is most noticeable in the 64MB and 256MB cases, where each of the 8 VMPlants hosts up to 16 64MB clones or 5 256MB clones, requiring an aggregate of more than 1GB memory of host memory.

Experiments have also been conducted to analyze the performance of the UML production line. For a 32MB UML VM that is instantiated via a full reboot, the average cloning time is 76s. The performance of the UML production line using checkpointed VM state is the subject of on-going experimental studies. This overhead incurred for dynamic instantiation can be amortized over the running times of an application; latency-hiding optimizations such as speculative pre-creation of VMs can be conceived, but have not yet been investigated.

Once instantiated, the run-time overhead of using virtual machines for CPU-intensive applications has been shown to be small in related work. In [3], the SPEC INT2000 benchmark was used to evaluate VMware, UML and Xen VM monitors, and the overheads relative to a physical machine are very small – 3% for UML, 2% for VMware and negligible for Xen. In [10], the SPECseis and SPECchem (serial) benchmarks showed a 6% overhead running under VMware. Grid technologies have emerged to benefit scientific and technical computing applications [11], which typically fall in this application domain. Application domains involving more of I/O and system activity however may incur a higher performance overhead. Experiments with the parallel Light Scattering Spectroscopy (LSS) application in [19] that involves frequent database accesses demonstrate an overhead of 13%, but even here the benefits from using virtualization by far exceed the overheads incurred.

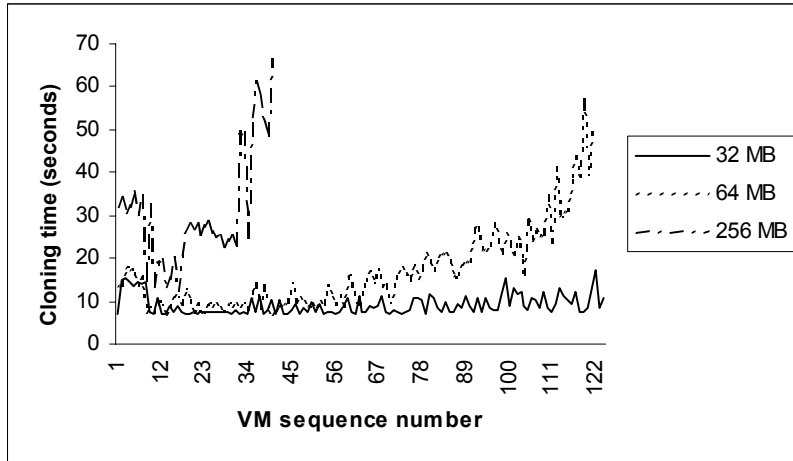


Figure 6: Cloning time as a function of virtual machine sequence number. The sequence number is obtained by VMShop in the experiments from the sequence of client requests for VM creation (total of 128 requests for 32MB and 64MB golden machines, and 40 for 256MB golden machines).

5 RELATED WORK

The key differentiators of the approach described in this paper from related work reflect the design decision of supporting flexible, application-centric VM environment configurations that can be instantiated dynamically through replication. These have motivated the design of DAG-based configurations, partial matching, and a cloning mechanism for “golden” images that can be applied to different virtualization technologies.

Oceano [2] and COD [7] are related projects that allow for on-demand provisioning. The instantiation process is based on a remote boot technique rather than cloning, and configuration templates are used instead of DAGs. Through remote boot, a physical machine can be allocated on-demand for a task; however, a physical resource cannot be time-shared between O/Ss running concurrently.

The commercial product VMware GSX (version 3.0 and above) [28] supports booting VMs over the network and installation of guest operating systems from a PXE server; VMware Virtual Center [29] supports virtual machine provisioning based on server templates. However, these products do not support the use of DAGs and partial matching.

The architecture described in [14] bears similarities with VMPlant: the XenoCorp is related to VMShop, and the XenoServer is related to the VMPlant. However, configuration is based on machine descriptions that are matched against available resources, and does not support ordering of configuration actions and partial matching. The SODA [16] architecture is geared towards creation of long-lived VM servers that are then used to handle services, rather than a service to handle VMs with flexible application configurations. The design uses configuration files and RPM-packaged services and does not support cloning based on partial matching. The Collective system [22] provides the ability to instantiate complete virtual networks of long-lived, generic virtual appliances with the goal of facilitating software deployment and administration. In contrast, the goal of VMPlant is to provide application-centric, short-lived sandbox VM environments for running Grid computations – possibly executing “clones” in parallel for high throughput. The choice of using a DAG model reflects this focus – it allows for finer-grained machine descriptions, matching and cloning, as opposed to Collective Virtual Appliance Language (CVL) [22]. Another related project (GridVM [27]) also relies on VMs to simplify software administration by enabling users to submit jobs from their desktops to different remote clusters.

It should be noted that the current focus on supporting unmodified applications (and their preferred environment) is reflected on the choice of whole system VMs running their own O/S kernel, as opposed to other virtualization approaches like the Software Pot [17], the Entropia Virtual Machine [6], Harness Virtual Machine [4]. The Out-Of-Order Virtual Machine described in [5], is a client-server based computing model motivated towards high-performance computing on parallel architectures.

6 CONCLUSIONS AND DIRECTIONS FOR FUTURE WORK

The main contribution of this paper is the definition of a framework for the management of Grid virtual machines that includes novel techniques for representing VM configurations in a flexible manner, for efficient instantiation of VM clones, and for composition of services to support large number of VM “plants”. This paper also characterizes the performance of the virtual machine instantiation based on a prototype implementation. The performance results encourage the use of this technique for on-demand provisioning of Grid VMs, showing that these flexible execution environments can be dynamically cloned often in less than a minute.

This paper focused on the core mechanisms for machine configuration and instantiation. There are other aspects of the VMPlant infrastructure that motivate further research, including: the use of a VMArchitect to instantiate customized virtual machines with router and tunneling capabilities to establish virtual networks that seamlessly span across distinct domains; cost and bidding models for VMPlant selection; migration of active VMs across plants, and speculative pre-creation of VM clones.

7 ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. EIA-9975275, EIA-0224442, ACI-0219925, EEC-0228390 and NSF Middleware Initiative (NMI) collaborative grants ANI-0301108/ANI-0222828. The authors also acknowledge a gift from VMware Corporation and a SUR grants from IBM. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, IBM, or VMware.

8 REFERENCES

- [1] S. Adabala, V. Chadha, P. Chawla, R. Figueiredo, J. Fortes, I. Krsul, A. Matsunaga, M. Tsugawa, J. Zhang, M. Zhao, L. Zhu, and X. Zhu. “From Virtualized Resources to Virtual Computing Grids: The In-VIGO System”, to appear, *Int’l J. Future Generation Computing Systems*, special issue, Complex Problem-Solving Environments for Grid Computing, David Walker and Elias Houstis, Editors.
- [2] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalanantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. “Oceano – SLA Based Management of a Computing Utility”, in *Proc. 7th IFIP/IEEE Int’l Symp. Integrated Network Management*, May 2001.
- [3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the Art of Virtualization”, *Proc. ACM Symp. Operating Systems Principles (SOSP 2003)*, Oct. 2003.
- [4] M. Beck, J. Dongarra., G. Fagg., A. Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulos, S. Scott, and V. Sunderam. “HARNESS: A Next Generation Distributed Virtual Machine,” *Int’l J. Future Generation Computer Systems*, volume 15, numbers 5-6, pp. 571-582, 1999.
- [5] G. Bosilca, G. Fedak, and F. Capello. “OVM: Out-of-order execution parallel virtual machine”. *Int’l J. Future Generation Computer Systems*, FGCS, 18 (4) (2002), pp. 525-537.
- [6] B. Calder, A. Chien, J. Wang, and D. Yang. “*The Entropia Virtual Machine for Desktop Grids*”, CSE technical report CS2003-0773, October 28, 2003, Dept. of Computer Science and Engineering, Univ. California, San Diego.
- [7] J. Chase, D. E. Irwin, and L. E. Grit, J. D. Moore, and S. E. Sprenkle. “Dynamic Virtual Clusters in a Grid Site Manger”, *Proc. 12th IEEE Int’l Symp. High Performance Distributed Computing (HPDC 2003)*, June 2003.
- [8] J. Dike, “A User-mode Port of the Linux Kernel”, *Proc. 4th Ann. Linux Showcase and Conf.*, USENIX Association, Atlanta, GA, Oct. 2000.
- [9] R. Figueiredo, N. Kapadia, and J. A. B. Fortes. “The PUNCH Virtual File System: Seamless Access to Decentralized Storage Services in a Computational Grid”, *Proc. 10th IEEE Int’l Symp. High Performance Distributed Computing (HPDC 2001)*, Aug. 2001.
- [10] R. Figueiredo, P. Dinda, and J. Fortes, “A Case for Grid Computing on Virtual Machines”, *Proc. 23rd Int’l Conf. Distributed Computing Systems (ICDCS 2003)*, May 2003.

- [11] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", *Int'l J. Supercomputer Applications*, 15(3), 2001.
- [12] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "*The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*," Technical report, Open Grid Service Infrastructure WG, Global Grid Forum, June 2002.
- [13] R. Goldberg, "Survey of virtual machine research", *IEEE Computer Magazine*, 7(6):34-45, 1974.
- [14] S. Hand, T. Harris, E. Kotsovinos, and I. Pratt, "Controlling the Xenoserver Open Platform", *Proc. 6th Int'l Conf. Open Architectures and Network Programming (IEEE OPENARCH '03)*, San Francisco, CA, Apr. 2003.
- [15] H. Höxer, V. Sieh, and K. Buchacker, "UMLinux - A Tool for Testing a Linux System's Fault Tolerance", *Proc. LinuxTag 2002*, Karlsruhe, Germany, June 6-9, 2002.
- [16] X. Jiang and D. Xu, "SODA: a Service-on-Demand Architecture for Application Service Hosting Utility Platforms", *Proc. 12th IEEE Int'l Symp. High Performance Distributed Computing (HPDC 2003)*, Seattle, WA, June 2003.
- [17] K. Kato and Y. Oyama. "*SoftwarePot: An Encapsulated Transferable File System for Secure Software Circulation*", Volume 2609 of Lecture Notes in Computer Science, Springer-Verlag, Feb. 2003.
- [18] H. Kreger, "*Web Services Conceptual Architecture*", White paper WSCA 1.0, IBM Software Group, 2001.
- [19] J. Paladugula, M. Zhao, and R. Figueiredo. "Support for Data-Intensive, Variable-Granularity Grid Applications via Distributed File System Virtualization – A Case Study of Light Scattering Spectroscopy", *Proc. Challenges of Large-Scale Applications in Distributed Environments (CLADE 2004)*.
- [20] R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing," *Proc. 7th IEEE Int'l Symp. High Performance Distributed Computing (HPDC 1998)*, Chicago, IL, July 1998.
- [21] T. Richardson et al, "Virtual Network Computing", *IEEE Internet Computing* 2(1), Jan./Feb. 1998.
- [22] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. Lam, and M. Resenblum. "Virtual Appliances for Deploying and Maintaining Software", *Proc. 17th Large Installation Systems Administration Conf. (LISA 2003)*, pages 181-194, Oct. 2003.
- [23] J. Sugerman, G. Venkitachalan, and B. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor", *Proc. USENIX Ann. Technical Conf.*, June 2001.
- [24] A. Sundararaj and P. A. Dinda, "Towards Virtual Networks for Virtual Machine Grid Computing", *3rd USENIX Virtual Machine Research and Technology Symp.*, May 2004.
- [25] A. Whitaker, M. Shaw, and S. Gribble, "Scale and Performance in the Denali Isolation Kernel", *Proc. 5th Symp. Operating Systems Design and Implementation*, Dec. 2002.
- [26] M. Zhao, J. Zhang, R. Figueiredo, "Distributed File System Support for Virtual Machines in Grid Computing", *Proc. 13th IEEE Int'l Symp. High Performance and Distributed Computing (HPDC 2004)*, June 2004.
- [27] Naregi Project, <http://www.naregi.org/data/SC2003Miura.pdf>
- [28] SBUML, http://sbuml.sourceforge.net/contents/quick_start.html, 13th Apr. 2004.
- [29] User-Mode Linux, <http://user-mode-linux.sourceforge.net/networking.htm>, 13th Apr. 2004.
- [30] VMware Corporation, http://www.vmware.com/pdf/gsx31vm_manual.pdf, 13th Apr. 2004.
- [31] VMWare Corporation, http://www.vmware.com/pdf/VC_Users_Manual_11.pdf, 13th Apr. 2004.