Virtualizing a Multiprocessor Machine on a Network of Computers

Kenji Kaneda   Yoshihiro Oyama   Akinori Yonezawa

1 Introduction

The fundamental goal of Grid computing is to seamlessly multiplex heterogeneous resources spread over multiple computing sites. For example, the Globus toolkit [8] aims to build a middleware for harnessing thousands of machines that have a variety of different hardware/software configurations such as CPU architectures or operating systems. Another example is SETI@home [1], which searches extraterrestrial intelligence using idle CPU cycles contributed by users.

One of major issues of the deployment of computational Grids is to provide execution environments that are customizable as well as safe. A customizable platform is necessary since the heterogeneity of underlying hardware/software configurations makes it difficult to run popular programs that may depend on specific operating systems or libraries. Furthermore, it is important to offer isolation and security mechanisms complementary to operating systems because an application that runs on volunteers' machines is usually untrusted and may destroy the machines.

One approach for deploying computational Grids is to *use classic operating system concept of a Virtual Machine Monitor (VMM)* [13] as the basis for resource sharing. A VMM virtualizes the real machine at the hardware layer (e.g., processor, memory, I/O devices), and exports a virtual machine (VM), which mimics exactly what a real machine would look like. The VMM facilitates the deployment of computational Grids as follows. First, the VMM allows a user to access consistent, customized application environments that are decoupled from physical resources. Second, the VMM ensures that an untrusted user or application can only compromise their own operating system within a virtual machine, not physical resources.

Recently the research community have been exploring several approaches for virtual machines for Grid computing [4, 5, 9, 12]. For example, VMPlant [9] is a Grid service that provide homogeneous execution environments across distributed Grid resources by supporting automated configuration and creation of flexible VMs.

These existing systems are, however, not yet feasible for Grid environments with respect to execution environments for parallel applications. More specifically, these systems themselves lack a special framework for parallel computing such as job submission or job scheduling mechanism. Since the use of these middlewares breaks the abstraction level of a VMM provides and sacrifices the simplicity of resource management, a user who want to execute parallel programs on multiple hosts faces difficulties.

To address the problem above, we propose a parallel computing environment which keeps the abstraction level that a VMM provides. In this paper, we present a software layer that emulates a multiprocessor machine on a network of computers. Like existing virtual machine monitors (VMMs), this software layer takes complete control of the machine hardware and creates virtual machines, each of which behaves like a complete physical machine that can run its own operating system. In contrast to the existing VMMs, our system creates a virtual multi-processor machine on a collection of single-processor machines. For example, the system gives users the illusion of multi-processor machine with N CPUs on top of N single-processor machines that are connected by networks.

This functionality of our system greatly simplifies utilization of distributed resources. For example, it can be used to take applications that are normally installed on large multiprocessor systems and run them on a number of smaller, less expensive machines. Suppose that a user would like to utilize two single processor machines. The user easily utilizes these machines with a commodity OS by creating a virtual dual processor machine. If the user forks processes on a guest OS running inside the virtual machine, these processes are automatically allocated on multiple virtual processors by the scheduler of the guest OS, and are finally allocated onto multiple physical machines by the VMM. The prototype described and evaluated in this paper supports instances of Linux guest operating system on Intel Pentium architecture.

The remainder of this paper is organized as follows. Section 2 presents the basic design of our system. Section 3 describes the implementation of the virtual machine monitor. The final section summarizes the paper.

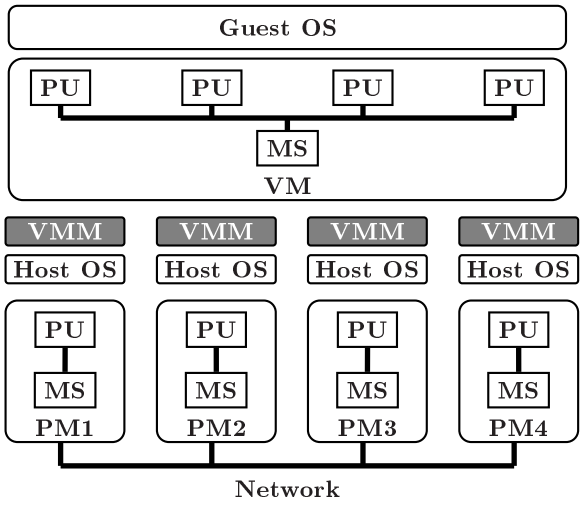

Figure 1: A virtual machine (VM) with four processors built on four single-processor physical machines (PMs). PU and MS denote a processor unit and a main memory (storage) unit respectively.

2 Basic Design

We describe the basic design of our system. First, we mentions the features of the virtualization that our VMM provides. Next, we introduce the architecture of our system and how virtual resources are mapped to physical resources.

2.1 Virtual Machine Interface

As mentioned before, our VMM creates a virtual *multi-processor* machine, in which a operating system can run. The features of the virtualization is summarized as follows:

- The virtual machine virtualizes a instruction set architecture (ISA) of prevalent Intel Pentium architecture.
- The VMM does not support full virtualization of the underlying machine but support paravirtualization of the machine. In other words, the instruction set used by a guest OS abstraction is similar but not identical to that of the underlying hardware. This promises improved performance, although it does require modifications to the guest operating system. The detail of the modification that a guest OS need to conform is described in Section 3.
- The virtual machine has the same number of processors as a collection of physical machines in which the virtual machine runs.

Note that among various processors of Intel Pentium Architectures, our design is targeted at the Pentium 4, Intel®Xeon™, and P6 family processors, which allow more relaxed memory ordering model than the other processors of Intel Pentium Architecture such as the Intel 486™processor [2].

In the current implementation, only Linux/x86 is hosted. We, however, believe that the underlying techniques described in this paper can be applied to other operating systems.

2.2 System Architecture

Our system is a *hosted* architecture [3], in which a VMM is placed on top of an underlying host operating system (See Figure 1). Like existing VMMs with a hosted architecture (e.g., LilyVM [6], FAUmachine [7]), the VMM runs as a user process of the host operating system.

This hosted architecture overcomes several technical and pragmatic hurdles around the virtualization although incurring larger overhead than the architecture where a VMM is placed on bare hardware directly [10]. First, the hosted architecture is useful for the virtualization of the Intel Pentium Architecture, which is not naturally virtualizable as shown in [11]. Second, the architecture allows a virtual machine to support a large diversity of peripheral devices with a minimal programming effort by relying upon a host operating system. Third, it allows a guest operating system to co-exist with a pre-existing host operating system.

A distinguishing feature of our system is that our systems enables virtualization of a multi-processor machine by running *multiple* VMM processes on machines connected by networks. In the existing systems, a single VMM runs simultaneously on a physical machine. In our system, on the other hand, VMMs that run on multiple machines cooperate to build a virtual machine.

The virtual resources (e.g., processors, memory, I/O devices) are mapped to physical resources in a following manner. Virtual processors map in a 1 to 1 fashion with the underlying physical processors. For example, in Figure 1, four processors of the virtual machine are respectively associated with the processor of physical machines. A memory and I/O devices are virtualized as if they are shared by one virtual machine.

3 Implementation of the VMM

This section describes how the VMM virtualizes processors, a shared memory, and I/O devices. Since the virtualization of processors is same as that in a single-processor machine and is not irrelevant to the main subject, we only give an overview of the mechanism. Since the virtualization of the other hardware resources differs from that in a single-processor machine, we explain it in detail.

3.1 Virtualizing Processors

The method of virtualizing processors is same as that of LilyVM [6]. In this scheme, a large portion of the virtual processor's instructions is executed by the machine's real processor without VMM intervention. Only some instructions that interfere with the state of the underlying VMM or host OS are not executed directly by the real processor and are interpreted by the VMM.

The instructions that require VMM intervention are called sensitive instructions. Before explaining how the VMM traps and emulate the sensitive instructions, we classify the sensitive instructions into two categories: privileged instructions and non-privileged instructions. Some sensitive instructions are called privileged instructions if the execution of these instruction at most privileged hardware domain causes a general protection exception. The instructions that do not cause a exception are called non-privileged instructions. For example, the `lgdt` instruction, which loads the value in the source operand into the global descriptor table register (GDTR) is a privileged instruction; and the `sgdt` instruction, which stores a content of the GDTR in the destination operand is not a privileged instruction.

The way how the VMM traps the execution of sensitive instructions differs whether the instructions are privileged or not. The trap of privileged instructions is straightforward. Since a virtual machine runs in user mode, the VMM only needs trap the exception caused by the execution.

On the other hand, the trap of non-privileged instructions needs a special mechanism. We statically rewrite portions of a guest OS kernel to make the execution of these instructions causes an execution. More specifically, we insert a illegal instruction before each sensitive instruction at kernel compile time. By intercepting a signal caused by a illegal instruction followed by a sensitive instruction, the VMM intercepts the execution the sensitive instruction.

This virtualization mechanism has the following benefits and drawbacks. The benefit is that we can port an operating system with small implementation cost. The drawback is that we cannot run system level binaries of which source code is not available.

3.2 Virtualizing a Shared Memory

A guest operating system that executes within a virtual machine expects a zero-based physical address space, as provided by real hardware. To virtualize such a memory in user address space, we devise the following four techniques. The first three techniques are for virtualizing the paging mechanism and same as that of LiLyVM. The fourth technique is for allowing virtual processors to assume a globally shared memory.

- The VMM maps pages to a physical memory (of the real machine) according to the page directory of a virtual machine. This mapping is updated whenever the virtual machine changes the value in the control register 3 (page directory base register), which contains the base physical address of the page directory.

- We rewrite the kernel code (the base address of the kernel address space) statically to avoid the overlap of kernel space of the guest and host operating system.
- The VMM emulates a page fault exception by intercepting the exception that is generated by the hardware of the real machine when a virtual machine requests a page not in memory.
- The VMM implements the consistency protocol of the shared memory using the virtual memory page protection mechanism (of the real machine). More specifically, the VMM uses the `mprotect` system call to control access to shared pages. Any attempt to perform a restricted access on a shared page generates a `SIGSEGV` signal. When trapping this signal, the VMM updates the content of the page by communicating with remote machines and upgrades the protection level of the page.

3.3 Virtualizing I/O Devices

The VMM prepares one central server for the emulation of I/O devices. The server keeps the state of all the devices and communicates with remote machines to I/O operations issued by a guest operating system.

For example, when a guest operating system read a value from the I/O port with the `in` instruction, the VMM emulates the instruction as follows. First, the VMM intercepts the execution of the instruction, and sends a request to the central server. When receiving the request, the server reads a value from the specified I/O port of the virtual machine, and sends the value to the VMM. Finally, the VMM copies the value to the destination operand of the instruction.

4 Conclusion

We have presented a VMM that virtualizes a multiprocessor machine on a network of computers. It provides an excellent platfrom for deploying a wide variety of parallel/distributed computing.

References

[1] SETI@home. http://setiathome.ssl.berkeley.edu/.

[2] *IA-32 Intel®Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2003.

[3] *Virtual Machines: Architectures, Implementations and Applications*, chapter 0. An Overview of Virtual Machine Architectures. to be published by Morgan Kaufmann Publishers, 2004.

[4] Amr Awadallah and Mendel Rosenblum. The vMatrix: A Network of Virtual Machine Monitors for Dynamic Content Distribution, 2002.

[5] Ananth I. Sundararaj and Peter A. Dinda. Towards Virtual Networks for Virtual Machine Grid Computing, 2004.

[6] Hideki Eiraku and Yasushi Shinjo. Running BSD Kernels as User Processes by Partial Emulation and Rewriting of Machine Instructions, 2003.

[7] Hōxer, H.-J. and Buchacker, K. and Sieh, V. Implementing a User-Mode Linux with Minimal Changes from Original Kernel, 2002.

[8] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[9] Ivan Victor Krsul and Arijit Ganguly and Jian Zhang and Jose A. B. Fortes and Renato J. Figueiredo. VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing, 2004.

[10] Jeremy Sugerman and Ganesh Venkitachalam and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, 2001.

[11] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor, 2000.

[12] R. Figueiredo and P. Dinda and J. Fortes. A Case for Grid Computing on Virtual Machines, 2003.

[13] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.