Performance Evaluation of a Xen-based Virtual Environment for High Performance Computing Systems

Vaddadi P. Chandu[*], Karandeep Singh[#]
[*]Georgia Institute of Technology, Atlanta GA
vchandu@cc.gatech.edu
[#] University of Maryland, College Park MD
ksingh@mail.umd.edu

Abstract

Virtualization is becoming an increasingly popular method to achieve software-based solution for sharing hardware infrastructure in a completely isolated manner. Xen virtual machine monitor (an open source software) is becoming a popular resource to implement virtualization for managing multiple operating systems instances within one physical computing node. Current research has focused on migrating these instances between nodes during runtime. This capability is useful for numerous activities, such as fault management, load balancing, and low-level system maintenance. In this paper, we evaluate the performance of Xen Virtual Machine Monitor for high performance computing (HPC) systems and discuss its suitability for 1) easy management of applications (e.g., automatic load balancing of MPI application processes), and 2) easy management of the HPC architecture (e.g., automatic migration of OS instances away from physical resources that have been predicted to fail in the near future).

1. Introduction

Virtualization presents the illusion of many smaller virtual machines, each running a separate operating system instance on the same machine. Such a virtualized environment provides isolation, security, low performance overhead, and supports heterogeneous applications [4]. For large enterprises, where on-demand capabilities are highly desirable, such a virtualization technique is very helpful in building an ideal solution for server and application consolidations. Virtualization allows higher system utilization through better allocation and de-allocation of resources dynamically to the participating applications in a completely isolated manner. However, currently these efforts are not directed at supporting HPC architectures, SSI [8], or parallel programming models that inject dependencies into the migration (e.g., message passing with MPI, global memory accesses in UPC). In this paper, we analyze the issues that confront in using the virtualization infrastructure on high performance architectures. We use Intel Cluster Toolkit [7] for benchmarking purposes and Xen hypervisor system [1, 2, 3] for providing a virtualized environment. The rest of the paper is organized as follows. In Section 2 we briefly describe the Xen hypervisor system and the Intel Cluster Toolkit benchmarking suite. In Section 3 we discuss our experimentation methodology. In Section 4 we discuss our results and we present the conclusions and future research in Section 5.

2.1 The Xen Hypervisor

Xen is a virtual machine monitor which provides a base for running many operating system instances on a single physical machine. Virtualization is known from Mainframes but was unsupported on x86 architectures until Xen was developed. Xen hypervisor acts as a layer between the underlying hardware and various operating systems (also known as guest operating systems) that are running on the hypervisor. The job of the hypervisor is to provide transparent and completely isolated image of the underlying hardware to each one of the guest operating systems. To achieve this goal, Xen does not aim at a complete virtualization of the physical hardware, but instead, uses a technique known as 'paravirtualization' where the guest operating systems are modified accordingly. However, the modifications are restricted to the operating system core and are completely invisible to the user. Xen appears as a kernel modification to the system administrator where the host and guest operating systems both carry kernel patches.

There are four principles governing the design of Xen. First, support for unmodified binaries. Second, support for full operating systems. Third, high performance and strong resource isolation. Fourth, completely hiding the effects of resource virtualization from guest operating systems. These design principles allow Xen to be useful on a wide variety of platforms ranging from large enterprises to research solutions such as computing Grids [9].

Fig. 1 shows the Xen architecture. The usual notation for guest operating system is DomU (short for User Domain) and for hypervisor is Dom0 (Domain0, which indicates full privilege). The Xen hypervisor allows each guest operating system a definite level of authority but ensures that all critical actions go through the hypervisor. For example, paging from guest operating system has direct access to the hardware page tables, but page updates are routed through the hypervisor. Exceptions from a guest operating system are registered through the hypervisor. Segmentation cannot install segment descriptors that are fully privileged. I/O devices are virtualized and asynchronous I/O rings are used to transfer data through the I/O devices which in turn have to go through the hypervisor.

T. Sobh (ed.), Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, 475–481.

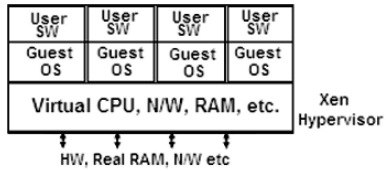

Fig. 1. Architecture of the Xen Virtual Monitor

2.1.1 Xen CPU management

The x86 architecture supports different privilege levels in hardware. It allows four different rings of protection, with highest privilege given to ring0 and lowest to ring3. Xen places the hypervisor in ring0 and the guest operating systems in ring1. Placing the guest operating system at a lower level prevents it from executing privileged instructions without the knowledge of the hypervisor. Instructions such as yielding the processor during idle cycles and installing new page table are all routed through the hypervisor. This guarantees isolation of the privileged instructions. CPU scheduling, which is another important issue in CPU management is handled through Borrowed Virtual Time algorithm [12]. This allows low-latency dispatch through virtual-time warping in which a newly woken domain is favored.

2.1.2 Xen Memory management

Xen allows guest operating systems to allocate and manage the hardware page tables. However, upon creation of each new hardware page table, a guest operating system has to register it with the hypervisor. The hypervisor exists in a 64MB section at the top of every address space which is restricted from the guest operating systems.

Direct writes to memory from a guest operating system are not allowed. All writes are required to be validated by the hypervisor to ensure consistency. Segmentation is also virtualized in a similar way, by validation to the hardware segment descriptor tables.

2.1.3 Xen I/O management

Xen presents a simple abstraction of the hardware devices. It does not perform a complete emulation of the devices. This fulfils the requirements for protection and isolation. Any I/O data is transferred to and from each domain via the hypervisor through asynchronous buffer descriptor rings. The rings are shared between the guest operating systems. Data is sent vertically through the system to allow quick validation checks and achieve high-performance communication. Event delivery to guest operating systems follows a lightweight delivery mechanism where asynchronous notifications are informed by updating a bitmap corresponding to the guest operating system. Guest operating systems can process these callbacks at their discretion.

2.2. The Intel Cluster Toolkit (ICT) benchmark suite

Now we describe the Intel cluster toolkit benchmark suite that we use to evaluate the performance of the Xen hypervisor. The ICT toolkit [7] consists of the following four modules. Intel MPI Benchmarks (IMB), Intel Trace Collector, Intel Trace Analyzer, and Intel MPI Library. For evaluating the performance of the Xen hypervisor for high performance computing systems, we are interested only in the Intel MPI Benchmarks (IMB). ICT uses a profiling methodology similar to the methodology used in [6]. In ICT, a parallel application is linked to a profiling library which adds the required time stamps to the MPI primitives and then calls the actual MPI library implementation. The time stamps are reported back to the user. ICT classifies the MPI primitives in IMB into three classes-Single transfer class, Parallel transfer class, and Collective class.

The benchmarks in the 'single transfer class' focus on measuring the performance of a single message transfer between two processes. PingPong and PingPing are two methods IMB uses to measure this. PingPong measures the throughput and startup of a single message between two processes by using MPI_Send and MPI_Recv primitives. The performance is measured as half the total time taken. PingPing also measures the throughput and startup of messages but under non-optimal conditions of oncoming traffic. To achieve this, PingPing uses a combination of MPI_Isend, MPI_Recv by simultaneously issuing the MPI_Isends, and waiting on MPI_Recv. MPI_Waits are introduced for consistency. Due to this difference, for PingPing pure timings are reported, and the throughput is related to a single message. Additionally, the numbers in PingPing, are usually between half and full of the PingPong throughput.

In the 'parallel transfer class', the benchmarks focus on measuring message passing efficiency under global load. Each benchmark is repeated with varying message lengths. The timings are measured as an average of the results in the repetitions. A periodic chain pattern [7] through MPI_Send-MPI_Recv, and an exchange pattern [7] in a periodic chain through MPI_Isend-MPI-Recv are used to benchmark the performance. In a periodic chain pattern, each process sends message to its right neighbor and receives message from its left neighbor. Due to its design, for two processes, this benchmark provides the bi-directional bandwidth of the system. On the other hand, in the exchange pattern, each process exchanges messages with both its neighbors, a pattern which occurs while computing the boundary conditions in problems like Parallel Ocean Program [10].

In the 'collective class', the benchmarks focus on measuring the performance of the collective primitives of MPI [11], such as MPI_Reduce, MPI_ReduceScatter, MPI_Allreduce, MPI_Allgather, MPI_Alltoall, MPI_Bcast, and MPI_Barrier. In this class, only the raw timing is displayed. The MPI_Reduce and MPI_Allreduce use MPI_SUM as the operator and reduce a vector of float items. In MPI_ReduceScatter, the vector is evenly reduced and split across all the processes. In MPI_Allgather, each process inputs a length of bytes and receives the gathered bytes, which is equal to the length of input bytes multiplied by the number of processes. The MPI_Alltoall functionality is same as the MPI_Allgather with the only difference that the length of the input bytes is equal to the length of the output bytes which in