WHITE PAPER

Virtualization Performance on SGI® UV 2000 using Red Hat Enterprise Linux 6.3 KVM

September, 2013

Author

Sanhita Sarkar, Director of Engineering, SGI

Abstract

This paper describes how to implement Red Hat® Enterprise Linux® 6.3's Kernel-based Virtual Machine (KVM) on a SGI UV system to consolidate many heterogeneous software environments. The resultant environment provides a complete virtual machine for each guest with integrated I/O capabilities and a large external storage pool. This paper presents benchmarks and explains best practices derived from case studies. These benchmarks show that with minimal configuration near-native performance is easily achieved on the SGI UV 2000.

TABLE OF CONTENTS

1.0 Introduction

Despite the advantages of virtualized environments, careful planning is necessary for a successful implementation. Some considerations include:

- The consolidation of multiple VMs onto one physical machine can magnify the impact of hardware failures. Careful planning is required for disaster recovery.
- Incorrect VM implementation can degrade application performance. Following best practices is critical to avoid severe performance penalties. There is a trade-off between consolidation and performance that needs to be balanced depending on the use case.
- Applying system management tools in a virtual environment is non-trivial.
- Unneeded or over-specified virtual machines in a data center waste the resources of the virtual servers' hosts.

The above considerations stress the importance of proper planning before implementing a virtualized solution. The best results will only be achieved by abiding by best practices and being aware of the specific needs of a virtualized environment.

2.0 Using Red Hat Enterprise Linux 6.3's KVM

Red Hat® Enterprise Linux® (RHEL) 6.3's Kernel-based Virtual Machine (KVM) is an excellent virtualization solution, as it solves several major problems:

- The KVM, being a kernel-level virtualization hypervisor, allows Linux Guest OSes to run with the same level of hardware support as on bare-metal with no additional guest drivers necessary.
- The KVM is actively supported by the open-source community. Any modern Linux distribution should run on top of the KVM as a guest OS without needing additional software or drivers.
- The KVM allows for many customization options for the VM. The GUI Virtualization Manager in RHEL 6.3 KVM allows for easy creation and management of VMs. For example, one can create virtual machines using a wizard and manage multiple virtual machines within a GUI interface.
- The KVM supports mature system administration tools, making it easier to administer many VMs in a data-center. For example, a command line tool, *virsh*, can be used to manage virtual machines. The *virsh* tool has an option called *vcpupin* to dynamically map virtual cores in a running VM to physical cores on the system.
- For more control and tuning capabilities, one can use the *virsh* edit command to edit the XML files that define the VM configuration. For example, the *cpuset* and *vcpupin* directives can be used to assign a VM to a particular set of CPUs and pin virtual CPUs to physical cores on the system.
- As of the date of this paper (July 2013), RHEL 6.3 KVM supports 160 Virtualized CPU cores (vcpus) per Guest OS (VM), allowing massive scalability.

A SGI UV running RHEL is perfectly suited for consolidating many applications onto a single system. It provides the largest single-system image on the market, allowing unparalleled performance and scalability.

2.1 Heterogeneous Software Environments with Virtualization

Virtualization using RHEL 6.3 KVM technology provides a complete virtual machine for each guest. A virtual machine appears to be an actual hardware system to the operating system and applications running on it. Unlike Linux Containers, KVM virtual machines require the installation of a guest operating system. This guest operating system is completely independent from the host and other installed VMs. Each guest is

installed, managed, maintained, and updated independently. Completely different versions of software can be installed on each guest. The result is an ideal environment for server consolidation.[1] Figure 1 shows the following components of a virtualized environment:

- **Host** – The physical computer on which the virtual machine is loaded.
- **Virtual Machine** – The software environment which appears to a guest OS as hardware. It consists of computing power (CPU), memory, network interface(s), and storage.
- **Virtualization Layer** – The KVM software layer (hypervisor) that makes available the physical hardware resources to the virtual machines.
- **Host OS** – The operating system installed on the bare physical computer (host).
- **Server Hardware** – The hardware components of the physical computer, shown here also attached to some external storage.

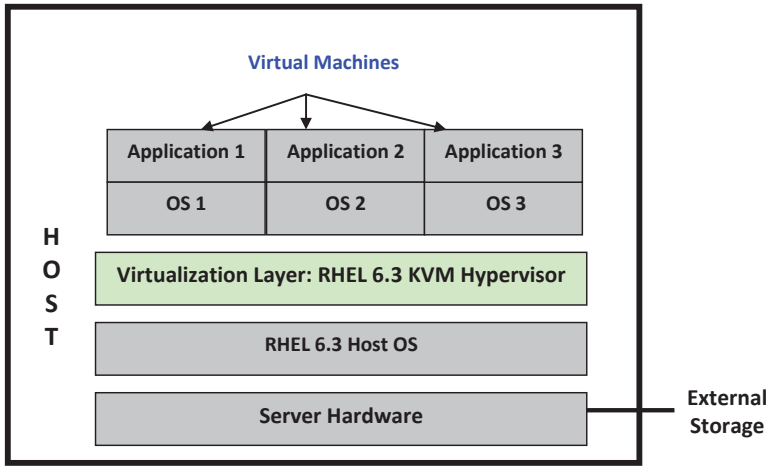

Figure 1: *Red Hat Enterprise Linux 6.3 KVM consolidating heterogeneous software environments*

3.0 Performance Benchmarks on SGI UV

This section describes the hardware and software configurations used for running a few performance benchmarks on an Intel® Xeon® E5-4600 processor-based SGI UV server platform, as well as the benchmark results.

3.1 Benchmark Configuration

The following benchmark configuration is used:

- Server:
  - 1 x SGI UV system with:
- 32 x Intel Xeon E5-4650 2.7 GHz processors
- 256 x 16 GB DIMMS (4 TB Total)
- External Storage:
  - 2 x SGI IS5500, each with 4 LUNs, each LUN with 7 x 600 GB 10K RPM SAS drives in RAID0
  - Storage is attached via 8 Gb Fibre Channel HBAs
- OS and Virtualization Environment:

1 http://www.redhat.com/products/virtualization/server/

- Host & Guest OS: Red Hat Enterprise Linux Server release 6.3 – kernel 2.6.32-279.el6.x86_64 SMP
- KVM software: libvirt-0.9.10-21.el6.x86_64; qemu-kvm-0.12.1.2-2.295.el6.x86_64; virt-manager-0.9.0-14.el6.x86_64
- Java Virtual Machine (JVM) Version
- Oracle Java HotSpot 1.6.0_23-b05 (64-bit)

3.2 Benchmark workloads

The following benchmark workloads were executed on the SGI UV in different modes – once on bare metal and then under different RHEL 6.3 KVM configurations – to show (a) the relative performance of a virtualized environment to the bare physical hardware (i.e., native performance) and (b) the benefits of using different RHEL 6.3 KVM configuration options:

- LMbench[2] 3.0, measuring the memory latency and bandwidth on UV and KVM;
- IOzone[3] v3.414, assessing the I/O capabilities of external storage on UV and KVM;
- SPECjbb®2005[4] v1.07, measuring the throughput and scalability of Java applications on the UV NUMA architecture and assessing the overall efficiency of the KVM environment with such applications.

3.3 Performance Metrics

The performance metrics for the LMbench, IOzone and SPECjbb®2005 benchmarks are, respectively:

- Memory latency (ns) and memory bandwidth (MB/s);
- I/O throughput (MB/s);
- BOPS (business operations per second).

3.4 KVM Configuration Descriptions

Four different VM configurations[5] were used for the testing:

- Default VM - The default VM as created using virt-manager. In this case, each vcpu is free to use any core on any socket (Figure 2).
- VM with NUMA nodes – This VM is created by editing the VM definition file (an XML file in /etc/libvirt/qemu) and using the cell construct. When the VM boots, the cells appear as NUMA nodes within the VM (Figure 3).
- VM with vcpu pinning – This VM is created by editing the VM definition file and using the *cpuset* and *vcpupin* constructs. These constructs assign vcpus to physical cores on the host on a 1:1 basis and pins them there (Figure 4).
- Multiple VMs with 1 VM per node (socket)[6] – Each node is assigned only one VM, with the number of VMs created determined by the demands of the application. The number of nodes sets an upper limit for the number of VMs that can be created. Each VM is thus aligned to a specific socket (Figure 5).

2 http://www.bitmover.com/lmbench/
3 http://www.iozone.org/
4 http://www.spec.org/jbb2005/
5 For visual simplicity, Figures 2-5 show a hypothetical system with 8 sockets (nodes), with each node containing 8 cores and with hyperthreading turned off.
6 A NUMA node, or node, relates to a physical socket on the system. There are 8 cores per node; enabling hypterthreading doubles this to 16 cores per node.

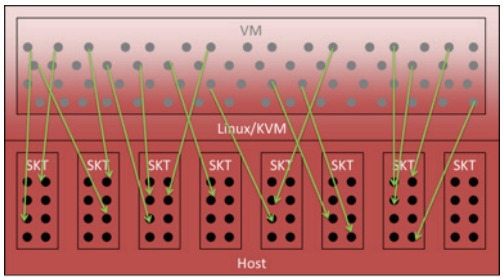

Figure 2: *Depiction of a Default VM with no vcpu Pinning*

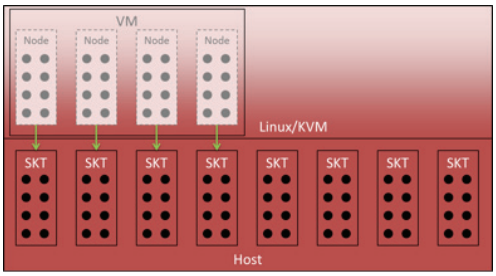

Figure 3: *Depiction of a VM with NUMA Nodes with vcpu Pinning*

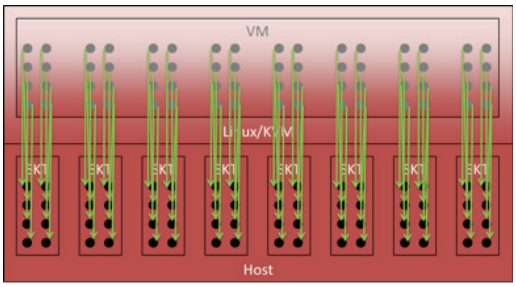

Figure 4: *Depiction of a VM with vcpu Pinning but no NUMA Nodes*

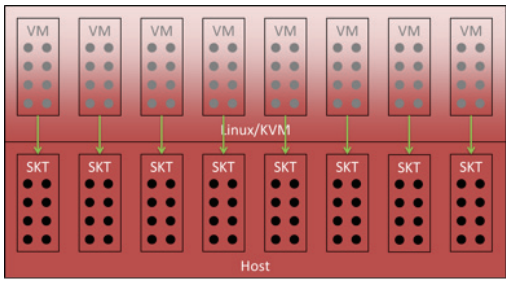

Figure 5: *Depiction of Multiple VMs with 1 VM per Node*

3.5 Benchmark Results

This section presents benchmark results for a variety of different system configurations. These results guide the creation of best practices for UV configuration.

3.5.1 Memory Latency and Bandwidth

Results from the LMBench 3.0 latency tests show that the main memory latency under the RHEL 6.3 KVM is nearly the same as Native, with random memory latency only 4% higher than Native. The KVM VM is running with 16 vcpus pinned to one CPU socket with HT enabled (Figure 6).[7]

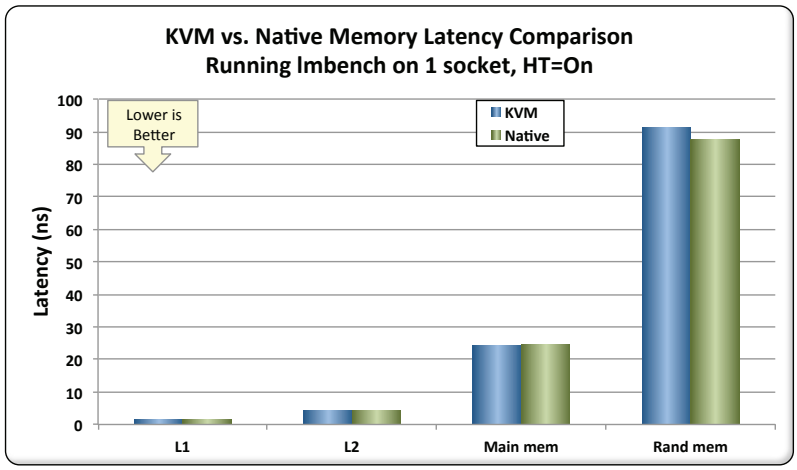

Figure 6: *LMbench 3.0 Memory Latency Tests on SGI UV 2000: RHEL 6.3 KVM vs. Native*

7 For many of the test results in this paper, the difference in performance from setting hyperthreading on or off is negligible; thus, only one case is shown.

Similarly, the LMBench 3.0 bandwidth tests (Figure 7) show the memory bandwidth for KVM as just 5% less than Native.

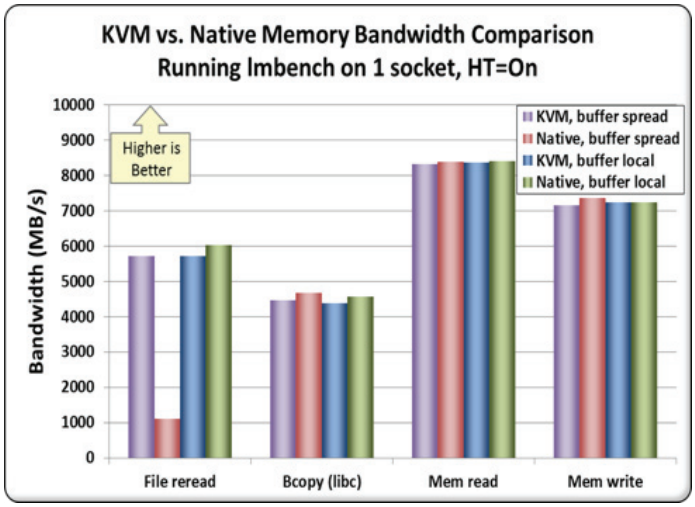

Figure 7: *LMbench 3.0 Memory Bandwidth Tests on SGI UV 2000: RHEL 6.3 KVM vs. Native*

However, the file reread test gave anomalous results, with the KVM being more than 400% faster than Native. This discrepancy is due to the kernel setting *memory_spread_page*. This setting controls how the file buffer cache is allocated. If set to 1 (buffer spread), the file buffer cache is spread among all nodes. If set to 0 (buffer local), then the file buffer cache is allocated local to the same node where the request is made to create a file. Changing *memory_spread_page* from buffer spread to buffer local (1 to 0) improved Native's file reread result by 445% without much impact to the other Native and KVM results. Changing this kernel setting does not impact KVM performance, since a VM pinned to a node automatically mimics the local cache behavior as set to buffer local. The *memory_spread_slab* setting behaves similarly for the allocation of kernel slab memory.

After setting these kernel settings to 0, all the results show that KVM performance is less than 5% below Native.

3.5.2 I/O Throughput

This section describes the I/O performance tests executed on a SGI UV directly attached to an external SGI IS5500 storage subsystem. Following are some benchmark specifics:

- The IOzone benchmark was run with Direct I/O using 8 threads writing 50 GB files to xfs filesystems on 8 LUNs on two IS5500s.
- Virtual Machine's virtual hard disk drive was created using virtio device driver with an XFS filesystem datastore and raw disk image. Within the VM, the virtual devices (/dev/vd*) were formatted with xfs filesystem.
- For Native tests, the SCSI (/dev/sd*) LUN devices were formatted with xfs filesystems.
- Note: This I/O test was done in a localized fashion, because even on bare metal non-localized I/O and (especially) buffered I/O exhibits very random behavior since the location of the buffer cache is not controllable.

Figure 8 shows bandwidth results for file reread operations that are similar to the results for memory bandwidth reread operations (see Figure 7). This is before adjusting the kernel settings for *memory_spread_page* and *memory_spread_slab* (i.e., buffer spread). That is, the KVM is outperforming Native.

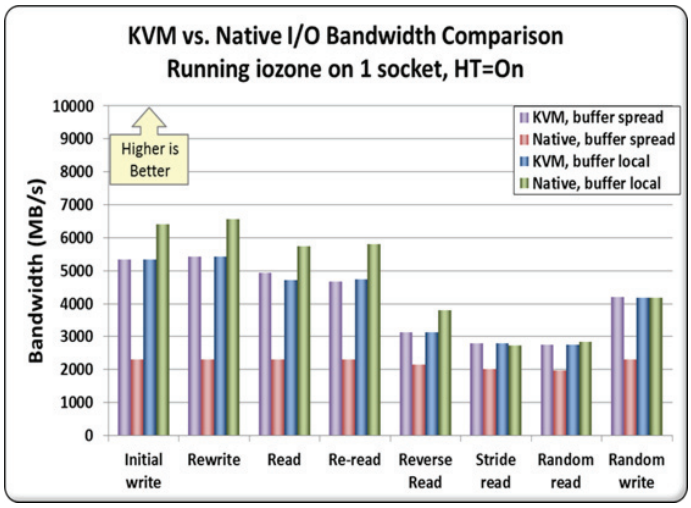

Figure 8: *IOzone Benchmark on SGI UV with Direct-attached External SGI IS5500 Storage: RHEL 6.3 vs. Native*

After adjusting the kernel settings to buffer local, Native throughput improved 35-185% while KVM throughput basically remained the same. This led to Native results being approximately 0-18% better than KVM results depending on the test.[8]

The KVM has slightly worse throughput with HT on (by about 1-17%) than with HT off (Figure 9). However, the throughput for KVM with HT off is still slightly below the throughput for Native with HT off or HT on for most tests.

For the stride read test, the KVM is barely better than Native but the difference (about 3-6%) is likely statistically insignificant.

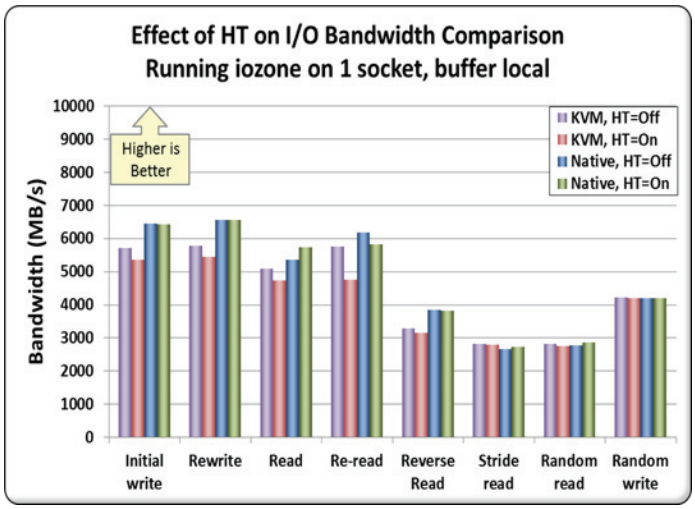

Figure 9: *IOzone Benchmark on SGI UV 2000 Comparing HT Off and On: RHEL 6.3 vs. Native*

8 The KVM drives were configured to use the *virtio* driver instead of passthrough. Using passthrough should improve performance.

3.5.3 Running SPECjbb®2005

For the purposes of these tests, SPECjbb®2005 was used in a couple different modes:

- Single JVM
- Multiple JVM
- Parallel Single JVM

The first two ways are the official methods for running SPECjbb®2005. The Single JVM method consists of using only one JVM for generating the workload. The Multiple JVM method runs multiple JVMs and sums the score for each to generate an overall score for the test. This method is often used for NUMA architectures. We assigned JVMs to specific vcpus (on VMs) or CPUs (on Native) using the *numactl* command. This allows the JVMs to be assigned to NUMA nodes, eliminating internode communication on the host.

The final mode (Parallel Single JVM) is a hybrid that is not an official mode for running SPECjbb®2005 but is used in order to drive multiple VMs. SSH is used to run multiple single JVM runs in parallel on different VMs. This exercises multiple VMs which are assigned to NUMA nodes. The score is the sum of each single JVM run and is analogous to how the Multiple JVM method score is calculated.

The Multiple JVM mode was used for Native SPECjbb®2005 runs. For KVM runs, either the Multiple JVM mode or the Parallel Single JVM mode was used depending on the purpose of the experiment.

3.5.4 KVM Configuration Comparisons Using SPECjbb – 32 cores

The following KVM configurations were used for these SPECjbb®2005 experiments:

- Default VM
- VM with NUMA nodes
- VM with vcpu pinning
- Multiple VMs with 1 VM per node

32 cores are the maximum number of vcpus that can be used in a VM when using the cell construct to create NUMA nodes within a VM. This is much lower than the maximum number of cores that can be used in a VM with RHEL 6.3 KVM (160 cores), which is covered in the next section.

Figure 10 and Figure 11 show the results for HT off and HT on, respectively. With HT off, 32 vcpus are spread across 4 nodes while, with HT on, 32 cores are spread across only 2 nodes. Native results are also included for comparison.

As NUMA awareness increases, SPECjbb throughput increases. This increase is more pronounced with hyperthreading off. With hyperthreading on a VM is using 16 cores per node (versus 8 cores per node with HT off); the resultant additional resource contention decreases the impact of additional NUMA awareness. Diminishing returns in performance take effect after using NUMA nodes on the VM with 2 nodes (~28% with HT on).

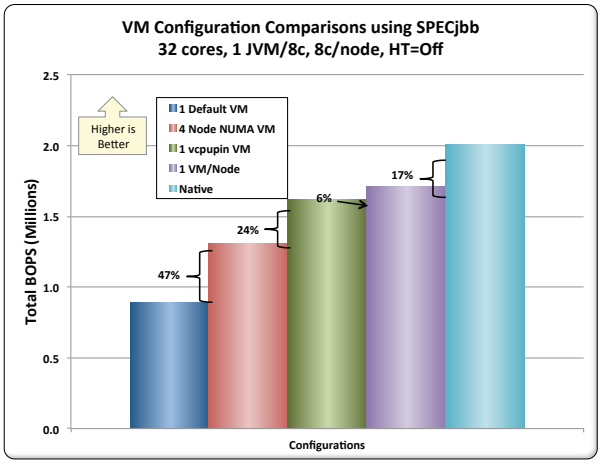

Figure 10: *KVM Configuration Comparisons Using SPECjbb®2005 – 32 cores, HT Off*

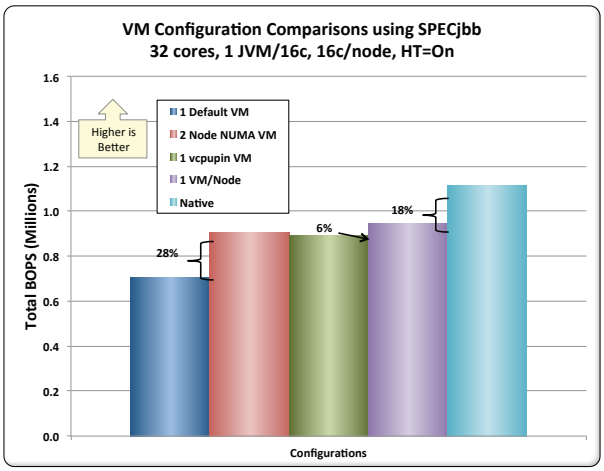

Figure 11: *KVM Configuration Comparisons Using SPECjbb®2005 – 32 cores, HT On*

3.5.5 KVM Configuration Comparisons Using SPECjbb – 160 cores

The configurations used in these experiments are unchanged from the previous section (32 cores) except not using the VM with NUMA nodes configuration, which is not supported for more than 32 cores. 160 cores is the maximum number of cores supported for a VM in RHEL 6.3 KVM which is why 160 cores were used in these tests.

Figures 12 and 13 show the results for hyperthreading off and on, respectively. With HT off, 160 vcpus are spread across 20 nodes; with HT on, 160 cores are spread across 10 nodes.

As with 32 cores, increased NUMA awareness increases throughput. Again, this effect is more pronounced with HT off. Pinning vcpus has less impact than it did on 32 cores, likely because the VMs are spread across more nodes. Switching to 1 VM per node substantially increases throughput performance; as was the case with the 32 core experiment, performance is within 18% of Native.

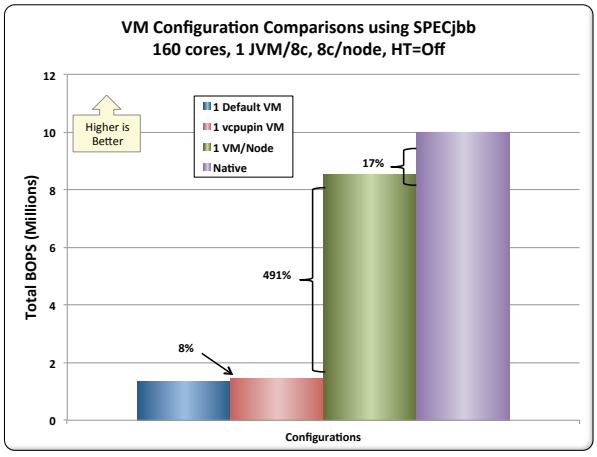

Figure 12: *KVM Configuration Comparisons Using SPECjbb®2005 – 160 cores, HT Off*

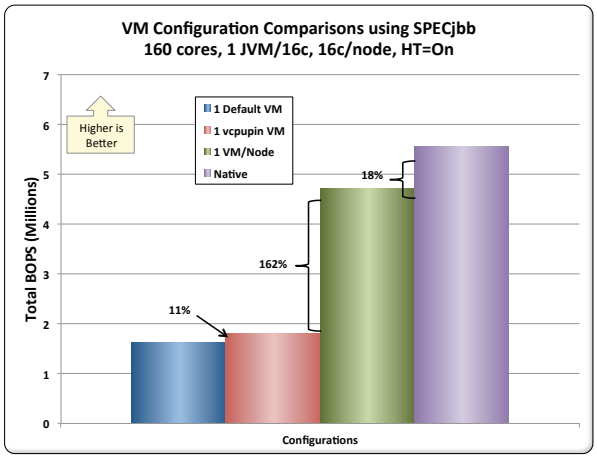

Figure 13: *KVM Configuration Comparisons Using SPECjbb®2005 – 160 cores, HT On*

These results highlight the superiority of using 1 VM per node on KVM, instead of the default VM or vcpu pinning configurations. In the 1 VM per node configuration, the system's resources are better isolated and utilized by the VMs, thus increasing throughput. Since there is a bigger increase in the number of nodes being used, the delta between the 1 VM per node case and the other cases is much larger. That is, for the 32 core experiments, the configuration change is moving from 1 VM to only 4 or 2 VMs (for HT off and HT on respectively). For the 160 core experiments, the configuration change results in going from 1 VM to 20 or 10 VMs (for HT off and HT on respectively).

3.5.6 KVM Scaling Comparisons Using SPECjbb – Running 1 JVM/node

For these experiments, the same three KVM configurations are uses as in the 160 core experiments:

- Default VM
- VM with vcpu pinning
- Multiple VMs with 1 VM per node

Figure 14 and Figure 15 (for HT off and HT on respectively) show that the default VM and the VM with vcpu pinning configurations see little benefit from using more than four nodes. This is in stark contrast to the scaling seen on the 1 VM per node and Native configurations. This further shows that running 1 VM per node offers the maximum performance with RHEL 6.3 KVM.

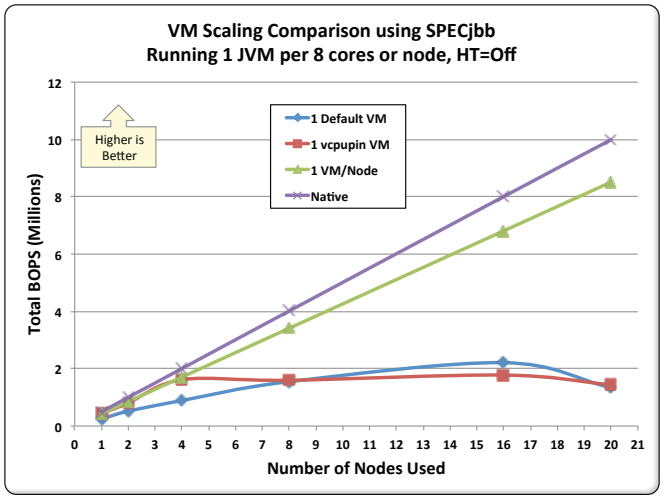

Figure 14: *KVM Scaling Comparisons using SPECjbb®2005 – 1 JVM/node, HT Off*

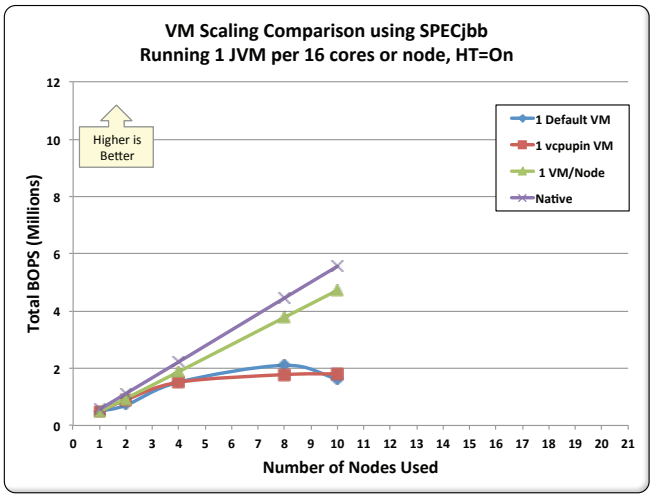

Figure 15: *KVM Scaling Comparisons using SPECjbb®2005 – 1 JVM/node, HT On*

3.5.7 KVM vs. Native SPECjbb®2005 Scaling Comparison

These tests compare the optimal KVM configuration (1 VM per node) to Native for increasing numbers of JVMs. By increasing the number of VMs (and thus JVMs), more NUMA nodes are utilized on the SGI UV system.

Figure 16 shows KVM and Native scaling linearly with the number of JVMs. Turning on HT increased performance by about 10% on both configurations. Native performance is roughly 17% better than KVM performance.

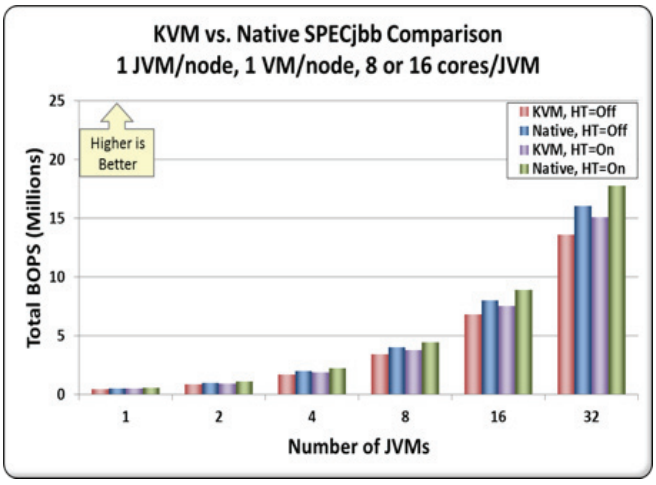

Figure 16: *KVM vs. Native SPECjbb®2005 Scaling Comparison – HT Off and On*

3.5.8 Multiple JVM Mode Comparison with SPECjbb®2005

The previous experiments used different methods of running SPECjbb®2005 to find the best KVM VM configurations. By contrast, this experiment was designed to observe the performance of SPECjbb®2005 using the same configuration on KVM and Native. The Multiple JVM method, which is optimal for the Native environment, is used. Since KVM is confined to 32 cores when configuring NUMA nodes within a VM, 32 core VMs with either 4 or 2 NUMA nodes (for HT off and on, respectively) are used for comparison. The Multiple JVM SPECjbb®2005 method is run on a single VM with multiple nodes and on Native with multiple nodes.

Figure 17 shows similar results as in previous experiments using multiple VMs:

- Both KVM and Native performance scaled linearly as the number of JVMs, or nodes, was increased.
- On both KVM and Native, having HT on increased performance by about 11%.

As a VM is configured to use more physical nodes, its performance does not improve as much as tests run on Native nodes using more physical nodes (see trend lines in Figure 16), even with vcpu pinning. Native performance is about 17% better than KVM performance for both HT off and HT on when using a single node, but Native throughput is 23-26% better than KVM when 2 nodes are used and 57% better when 4 nodes are used (HT off).

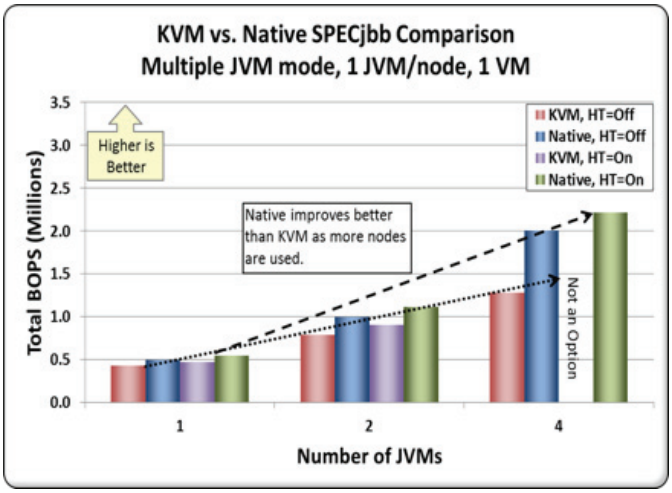

Figure 17: *KVM vs. Native SPECjbb®2005 Multiple JVM Mode Comparison*

3.6 Performance Summary

To summarize on performance:

- Measured by memory latency and bandwidth, **KVM performance is 95-100% of Native performance.**
- With Java applications, measured via SPECjbb (using 1 JVM per node), **KVM is 83% of Native performance.** Hyperthreading adds 10% more throughput for both KVM and Native.
- For I/O throughput measured via IOzone (with HT on), **KVM performance is 82-100% of Native performance.**
- Experiments show that kernel parameters *memory_spread_page* and *memory_spread_slab* play an important role on the Native I/O throughput and Native file buffer memory bandwidth. Performance improvements of approximately 35-185% and 445%, respectively, can be achieved by toggling these parameters. Changing these parameters does not impact KVM performance since a VM pinned to a node automatically sees optimal cache behavior.
- *Cpuset* and *vcpupin* do a good job of keeping vcpus pinned to cores and using memory local to those cores.
- Running SPECjbb on a 4 node VM and monitoring the memory used on the host appears to show that the memory is used on the correct physical nodes. However, the same memory utilization is seen when using *vcpupin* without the use of cells (and with better performance), so using *vcpupin* instead of cells is currently recommended. This could change if cells are updated to support more vcpus and more closely align vcpus and memory.
- For a 160 vcpu VM, the overall throughput of the system is increased by using the *vcpupin* feature, but levels off after 4 JVMs.

4.0 Best Practices

The best practices for achieving optimal performance on a virtualized SGI UV system can be described as follows:

- The size of host and guest OSes supported by RHEL 6.3 KVM today (September 2013) are 160 CPUs and 160 vcpus, respectively. It is recommended to stay within this limit on UV, even though UV can scale up to 4096 physical cores. Table 1 shows the RHEL 6.3 KVM official limits and recommendations on UV.
- For the best performance, use virtual machines with vcpu counts and memory sizes less than or equal to the physical hardware of the NUMA node it is running on. For example, if a UV system has a NUMA node with 8 cores and 64 GB of memory, a VM should be less than or equal to this size.

Figure 18 shows an example configuration of a 256 core UV with 32 virtual machines configured on each of the 32 NUMA nodes having 8 vcpus. Similarly, a 64-core UV may be configured with 8 virtual machines, each configured on one of the 8 NUMA nodes to use 16 vcpus (with HT on). One could theoretically configure a maximum of 512 virtual machines on a 512-core UV system having 4 TB of memory by restricting each VM to 1 core and 8 GB of memory.

- When configuring a VM that must span more than one NUMA node, it is important to use *vcpupin* directives to keep the vcpus pinned to physical cores. This can be done within Virt-Manager and *virsh vcpupin* or through configuration in the VM xml definition file via *virsh* edit.
- Enabling transparent hugepages (THP) on the UV host, which enables THP on the KVM, results in a better KVM performance.
- When using hyperthreading-capable CPUs on UV, it is recommended to allocate a number of vcpus to a VM which is less than or equal to the number of threads that exist on a CPU. For example, if a CPU has 8 cores/16 threads, a VM with 16 or fewer vcpus should be used. Be sure to configure the VM to use the correct CPU number associated to that socket.
- Use virtual hard disks via the *virtio* driver to attach external I/O to a VM on UV.

	Host	Virtual Machine (Official)	Virtual Machine on UV 2000
Max Cores	160/4096	160	Up to the Number of cores on a NUMA Node
Max Memory	2 TB/64 TB	2 TB	Up to the amount of memory on a NUMA Node

Table 1: *RHEL 6.3 KVM official limits and SGI UV recommendations*

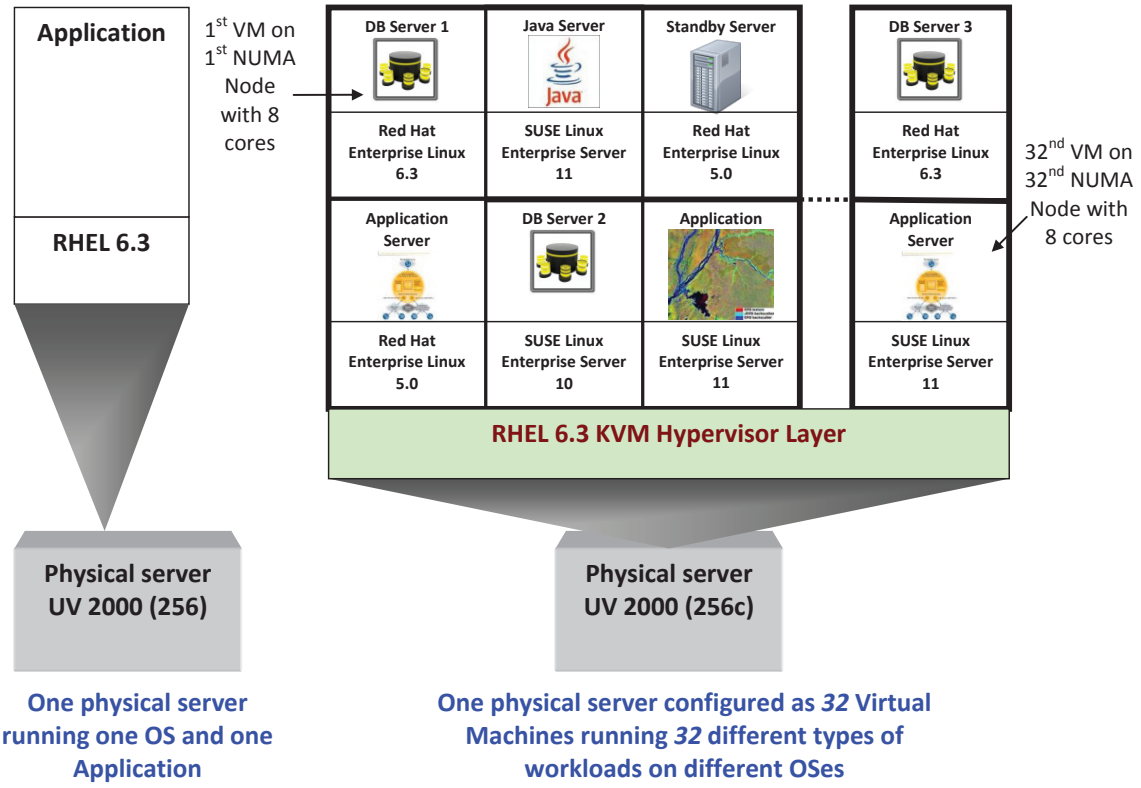

Figure 18: *Consolidation of Heterogeneous Workloads on SGI UV using Red Hat 6.3 KVM*

5.0 Summary

The SGI UV 2000 makes full use of the new virtualization features of Red Hat Enterprise Linux 6.3 KVM up to the KVM's maximum host size limits. This enables the consolidation of multiple physical servers onto one UV. This increases utilization of computer resources, eases manageability, improves energy savings and lead to a more cost-effective solution. SGI UV with RHEL 6.3 KVM enables easily manageable, high performance installations that can run multiple database deployments, collections of commercial applications, development/debugging environments, and IT infrastructures.

Global Sales and Support: sgi.com/global