Performance of Enterprise Java Applications on VMware vSphere 4.1 and SpringSource tc Server

VMware vSphere 4.1

Enterprise-level Java applications are ideal candidates for deployment on a virtualized infrastructure using VMware vSphere 4.1. With VMware vSphere 4.1, Java applications can take advantage of the configuration flexibility and consolidation benefits of virtualization while still achieving high levels of performance. This applies to JEE applications deployed in full application-servers, as well as applications deployed on lightweight Web containers such as Tomcat and SpringSource tc Server.

In this paper we present the results of a performance investigation using a representative enterprise-level Java application on VMware vSphere 4.1. In keeping with recent trends in Java development, we use a Java application which is deployed as a Web application on SpringSource tc Server and which integrates enterprise-level functionality using the Spring framework. We examine all aspects of the performance of the application in a variety of deployment scenarios. We first look at the performance of a single instance of the application deployed both in a virtual machine (VM) and natively without virtualization. In particular, we investigate the ability of the virtualized application to maintain acceptable response-times across a range of VM configurations. We then investigate the impact on performance of various scale-up versus scale-out tradeoffs. As the load on an application increases, the controls provided by VMware vSphere 4.1 give significant flexibility in choosing between scaling-up the configuration of a single VM, or scaling-out to use multiple VMs. Understanding the performance implications of these tradeoffs is important, particularly when non-performance-related criteria, such as maintenance and software licensing costs, are major factors in configuration decisions.

Contents

1 Executive Summary

The results of the tests discussed in this paper show that enterprise-level Java applications can provide excellent performance when deployed on VMware vSphere 4.1. The application used in these tests was Olio, a multi-tier enterprise application which implements a complete social-networking Website. Olio was deployed on SpringSource tc Server, running both natively and virtualized on vSphere 4.1.

We first performed a series of tests to determine whether an enterprise-level Java application virtualized on VMware vSphere 4.1 can provide equivalent performance to a native deployment configured with the same memory and compute resources. We used response-time as the primary metric for comparing the performance of native and virtualized deployments. The results show that at CPU utilization levels commonly found in real deployments, the native and virtual response-times are close enough to provide an essentially identical user-experience. Even at peak load, with CPU utilization near the saturation point, the peak throughput of the virtualized application was within 90% of the native deployment.

We then investigated the performance impact of scaling-up the configuration of a single VM (adding more vCPUs) versus scaling-out to deploy the application on multiple smaller VMs. At loads below 80% CPU utilization, the response-times of scale-up and scale-out configurations using the same number of total vCPUs were effectively equivalent. At higher loads, the peak-throughput results for the different configurations were also similar, with a slight advantage to scale-out configurations. These results show that Java applications can be virtualized successfully in both scale-up and scale-out configurations.

2 Test Environment

2.1 Overview

This section gives an overview of the test environment used when collecting the performance results discussed in this paper. The test environment consists of the Java application and workload, the metrics used to measure the performance of the application, and the software and hardware infrastructure for the tests.

2.2 Application and Workload

2.2.1 Application

In order to collect performance results that give meaningful insight into the behavior of enterprise-level Java applications running on VMware vSphere 4.1, we needed an application that uses a variety of enterprise-level Java features, as well as a controlled workload to drive against that application. We chose to use the Olio application and workload for this project. Olio is an incubator project of the Apache Software Foundation (http://incubator.apache.org/olio/). This project has developed an application and benchmark to enable the comparison of different Web-application technologies. The Olio application implements a social networking Website which allows users to browse, post, and update information about social events. The project has developed implementations of this application in Java, PHP, and Ruby.

The Java implementation of the Olio application uses a variety of enterprise-level Java features. The front-end is implemented using Servlets and JSPs, augmented with a variety of Web 2.0 technologies and toolkits, including jMaki and Yahoo widgets. Compiled JSPs and static content are cached either in a local cache or using a distributed caching layer. Business logic is implemented in Java classes which handle requests forwarded from a Controller servlet, and persistence is managed using the Java Persistence API (JPA).

2.2.2 Workload

In addition to the Olio application, the Olio project has created a workload driver module for the Faban benchmark driver tool (http://faban.sunsource.net/). This allows a controlled and repeatable load to be driven against the application in order to collect meaningful performance results. The load for the Olio workload is controlled by setting the number of simulated users for a run. Each simulated user drives a

controlled amount of load on the application. The load takes the form of a mix of operations, selected from a set of seven representative operations performed using the application. These operations include actions such as adding a new social event, searching for events by tag, and retrieving the details about a registered user. Each operation of the Olio workload has a quality-of-service (QOS) requirement defined in terms of the 90th percentile response-time. In addition, the workload driver attempts to enforce a five second cycle-time for each operation. A run of the workload must meet the QOS and cycle-time requirements, as well as other requirements that guarantee the consistency of the workload, in order to be a valid passing run. Additional details about the operation mix and QOS requirements are given in the Appendix.

2.2.3 Extending Olio with Spring

The Olio application is a Web application which is deployed in a JEE Web container. However, it depends on transaction-management services to be provided by the container. These services are not provided natively by Tomcat or tc Server. For this project, we modified the Olio application to use the Spring framework to inject transaction management and JPA support. Spring enables a simple annotation-based method for injecting transactional behavior, and, through its dependency-injection support, simplifies the task of changing underlying service implementations at deployment time.

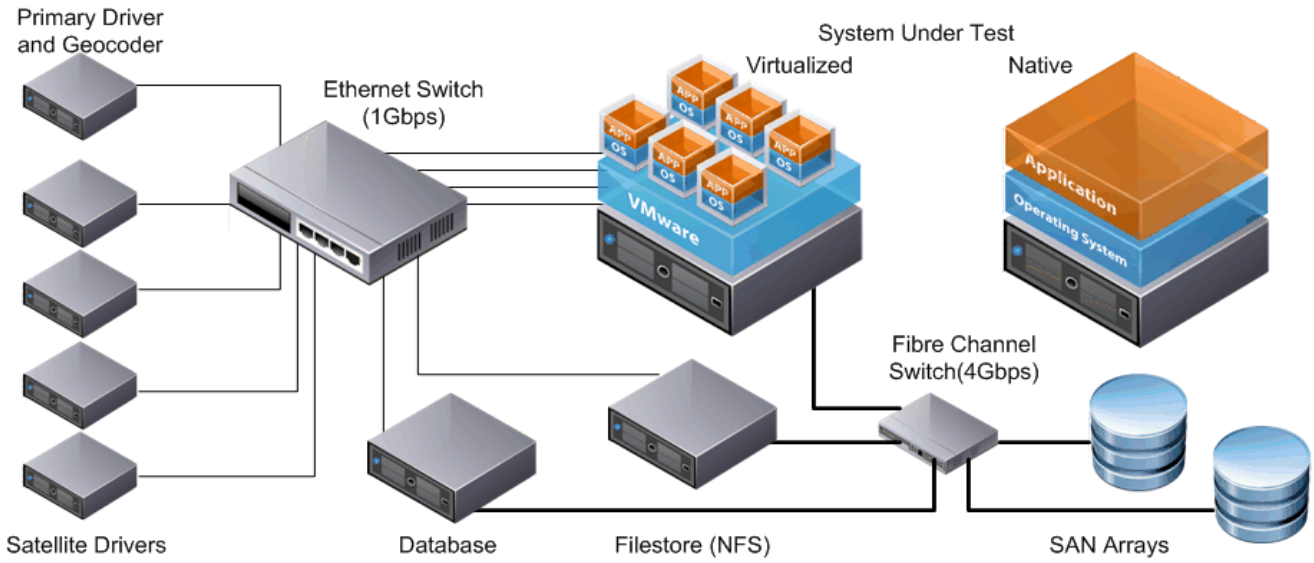
2.3 Performance Metrics

The performance achieved on a single run of the Olio workload is measured in terms of response-time and throughput. The Olio workload driver reports the average and 90th percentile response-time for each workload operation. The response-times reported in this paper are the averages for all operations, with each operation's response-time weighted by the frequency of that operation in the workload mix. The throughput of an Olio run is the number of operations per second performed during the steady-state period of the run. The throughput metric is valid only for runs in which all response-time and cycle-time requirements are satisfied. The peak throughput occurs at the largest number of users for which the application passes all requirements.

2.4 Testbed

Olio requires a number of supporting services and a robust infrastructure in order to achieve maximum performance. As a result, the testbed used in this project needed to be carefully sized and tuned. This section gives an overview of the hardware and software components of the testbed.

Figure 1. Testbed



The hardware components of the testbed are shown in **Figure 1**. The testbed consists of the following components:

- The System-Under-Test (SUT). This is the server that hosts the Olio application, whether in VMs or natively. For these tests the SUT was an HP ProLiant DL585 G5 with four AMD 8382 Opteron Processors @ 2.6GHz (quad-core, 16 total cores) and 64GB of memory.
- The primary driver system. This system runs the software to control the tests. It also hosts the Geocoder emulator, a component of the Olio benchmark used to provide geographical information to the application.
- Multiple satellite driver systems.
- The database server. This database hosts the user, event, and other tables for the Olio application.
- The filestore server. This server holds the static content used by the Olio application. It exports an NFS mount with the filestore directory which is mounted by the servers running the Olio application.
- An Ethernet switch. The network infrastructure used in these tests is 1Gbps.
- A Fibre Channel switch for access to the storage arrays.
- A SAN array which hosts the storage for the VMs.
- A SAN array which hosts the storage for the database and filestore.

Additional details about the components and their tuning are given in the Appendix.

3 Virtualizing an Enterprise Java Application

3.1 Overview

Performance is often a key concern when moving an enterprise Java application from a native server to a virtualized environment. In this section we examine the question of whether a Java application virtualized on VMware vSphere 4.1 can provide equivalent performance to a native deployment configured with the same memory and compute resources. We use response-time as our primary metric for comparing the performance of native and virtualized deployments. Response-time is the primary indicator of quality-of-service as experienced by an application's users. We also examine the peak throughput for the native and virtualized deployments.

3.2 Response-Time Results

For a user of an interactive application, observed response-time is the primary measure of performance. In order to compare the performance of native and virtualized deployments, we measured the change in response-time as the load was increased on a single Olio instance running on the server natively and in a VM. We tested configurations with 1, 2, and 4 CPUs. Beyond 4 CPUs, the network load exceeded the capabilities of a single 1Gbps Ethernet link. For the native tests, the operating system was booted into the selected CPU configuration using kernel boot parameters (for example, maxcpus=1). The same JVM settings were used for the native and virtualized tests.

Figure 2 shows the 90th percentile response-times and CPU utilizations for the 1-CPU configurations as a function of increasing load. The horizontal marks on the response-time curves indicate the highest load that passed the QOS limits. The response-times used are the weighted averages of the response-times of all operations in the workload. See "**Table 2.** Olio workload description" in the Appendix for a list of these operations.

Note: The graphs in this section show slight differences in average response-time when the load reaches the QOS limits for the VM run versus the corresponding Native run. These peak loads do not occur at the same point on the graph because an individual operation may fail its QOS requirement before the weighted average of the response-times of each operation exceeds the weighted average of the QOS limits.

The results in **Figure 2** show that at loads that keep the CPU utilization of the VM below 80% (at around 300 users), both the native and virtualized 90th-percentile response-times are well within the QOS limits. The user experience would be similar for both platforms at these load levels. It is only at extreme loads that response-times in the virtualized configuration increase more rapidly than those of the native configuration.

Figure 2. Native and Virtual Response-Times with 1 CPU

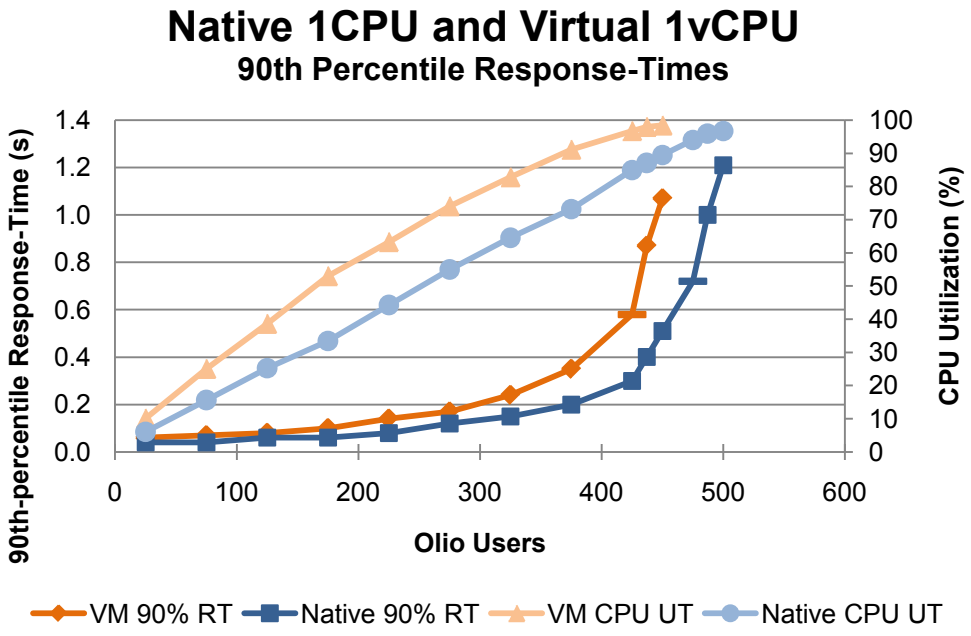


Figure 3 shows the 90th percentile response-time curves and CPU utilizations for the 2 CPU native and virtualized cases. Below 80% CPU utilization in the VM, the native and virtual configurations have essentially identical performance, with only minimal absolute differences in response-times. It is only after CPU utilization of the VM exceeds 80% (slightly under 800 users) that the response-times in the VM increase more rapidly than the native case.

Figure 3. Native and Virtual Response-Times with 2 CPUs

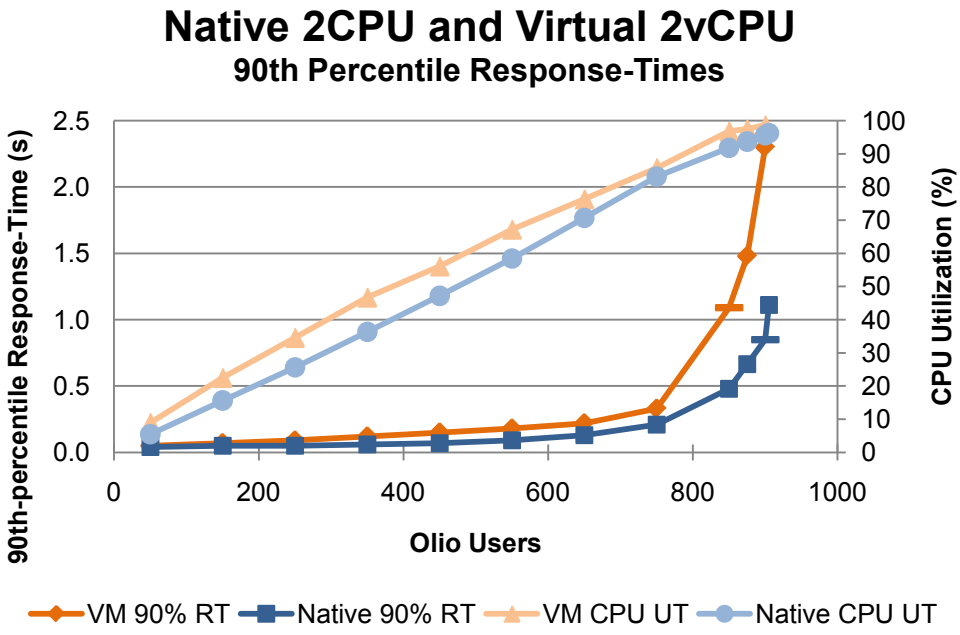
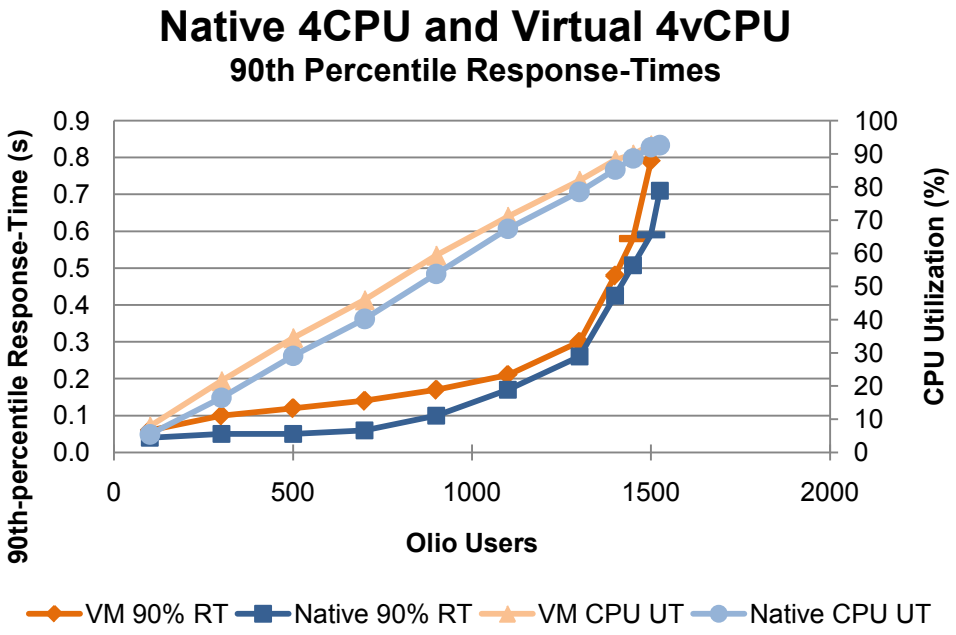


Figure 4 shows the 90th percentile response-time curves for increasing load in the 4 CPU native and virtualized cases. The native and virtual configurations have essentially identical performance across all loads, with only minimal differences in response-times. The closeness of the response-times even at higher loads can be explained by the effect of external factors, which will be discussed in Section 4.3, "Peak Throughput Scaling."

Figure 4. Native and Virtual Response-Times with 4 CPUs



3.3 Peak Throughput Results

The peak throughput for Olio is measured in terms of the number of operations-per-seconds that can be handled while still meeting the QOS requirements. The peak throughput gives an indication of how hard the system can be driven before the delay experienced by the end-users becomes unacceptable. A comparison of the peak throughput of different configurations gives an indication of the saturation points of the configurations. However, as shown in the previous sections, this does not necessarily indicate that the configuration with the highest peak-throughput will provide noticeably better response-times at more reasonable loads.

Figure 5 shows the peak throughput for a single instance of Olio running on tc Server, both natively and in a VM, with 1, 2, and 4 CPUs. **Figure 6** shows the virtual results normalized to the native results at the same number of CPUs. The application running in a VM is able to achieve 90% or more of the peak native throughput in all cases.

Figure 5. Single Instance Peak-Throughput

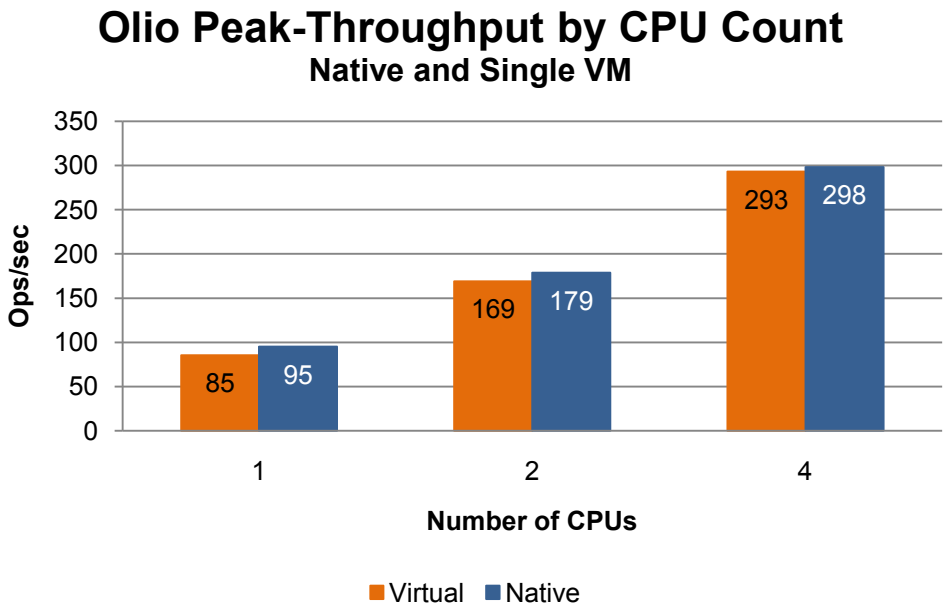


Figure 6. Single Instance Virtual Peak-Throughput Relative to Native

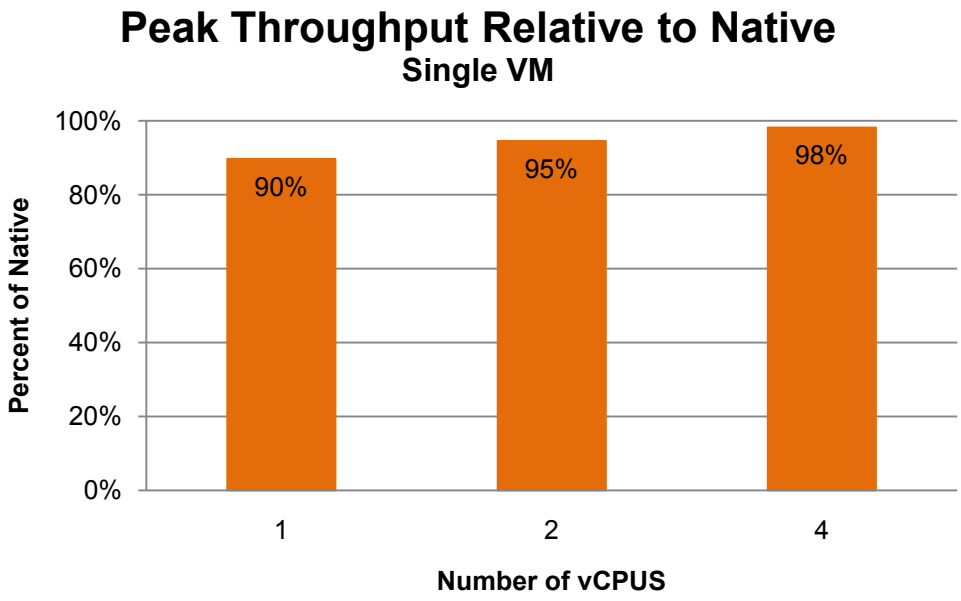
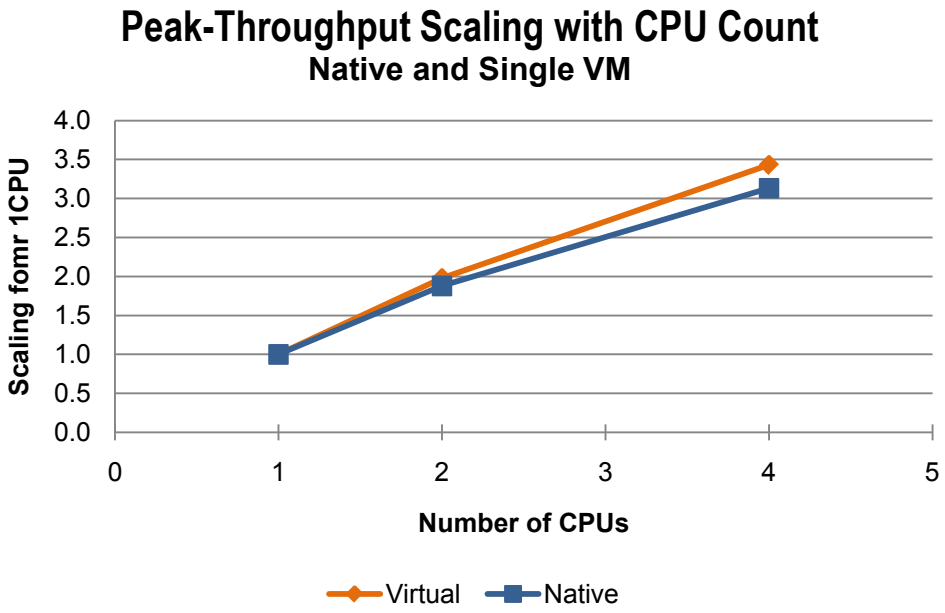


Figure 7 shows the scaling of peak-throughput as additional CPUs are added in the native and virtual cases. In both cases the peak-throughput scales nearly linearly from 1 to 2 CPUs, but drops off at 4 CPUs. This drop in scalability may reflect a number of factors in both the infrastructure and the application. Likely explanations include the effect of the high network bandwidth and throughput, and the additional load on the shared caches and TLBs within the processors. As with any workload, the scalability of the application

is limited by the capabilities of the underlying hardware. We investigate these effects in Section 4.3, "Peak Throughput Scaling."

Figure 7. Scaling of Single Instance Peak Throughput with Increasing CPU Count



3.4 Discussion

The results in this section have shown that it is possible to virtualize a Java application with performance equivalent to similar native configurations. When the CPU utilization is below 80%, the virtual response-times are effectively equivalent to the native case. This can be achieved by sizing a single VM appropriately for the expected peak loads, or, as will be discussed in Section 4, by scaling the deployment out across multiple smaller VMs. This is true across all tested vCPU counts. Even when load spikes drive the VM close to CPU saturation, the peak-throughput is very close to the native case. Furthermore, these results were achieved without requiring any special JVM tuning when virtualizing the application.

4 Deploying a Virtualized Java Application

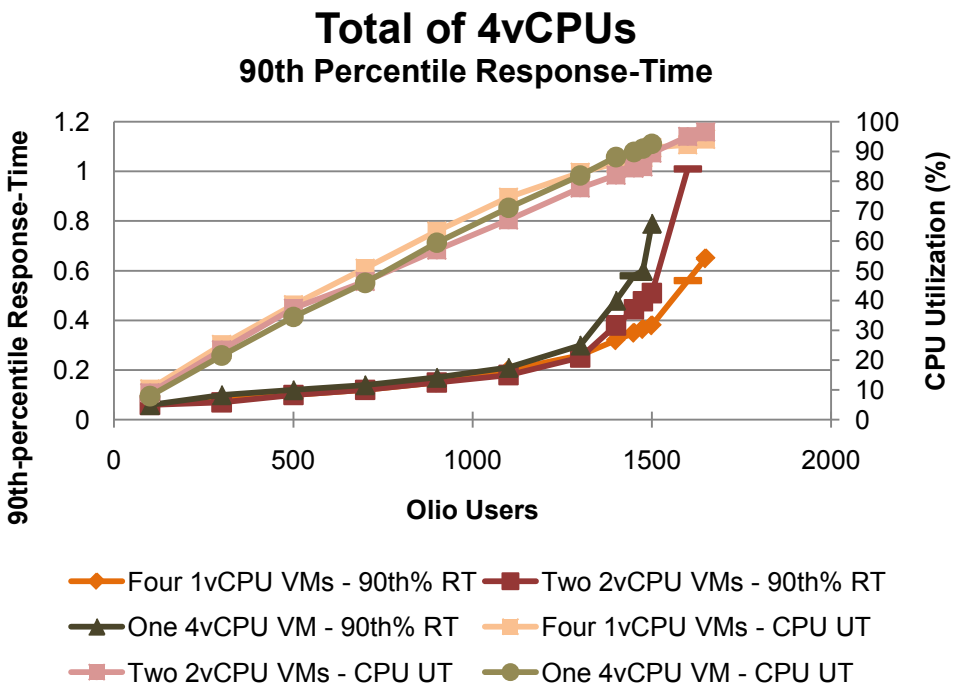
4.1 Overview

In a virtualized environment, there are many ways that an enterprise-level Java application can be deployed to provide adequate resources for the expected load. In particular, deploying in tc Server on VMware vSphere 4.1 provides great flexibility when deciding between scaling-up the configuration of a single VM (adding more vCPUs) versus scaling-out to deploy the application on multiple smaller VMs. Performance is an important factor in this decision, as are other considerations such as reliability, software licensing, and maintenance costs. In this section we examine the performance impact of different scale-up versus scale-out configuration tradeoffs.

4.2 Scale-Up versus Scale-Out Tradeoffs

In this section, we consider the performance impact of different scale-up and scale-out choices for configurations using a total of 4 vCPUs. **Figure 8** shows the change in the 90th percentile response-time with increasing load for a single 4-vCPU VM, two 2-vCPU VMs, and four 1-vCPU VMs. The response-times are essentially identical for all of these cases until the CPU utilization approaches 90%. This shows that, at typical utilization levels, the choice between the configurations can be made based on criteria other than performance.

Figure 8. Response-Times for configurations with total of 4-vCPUs



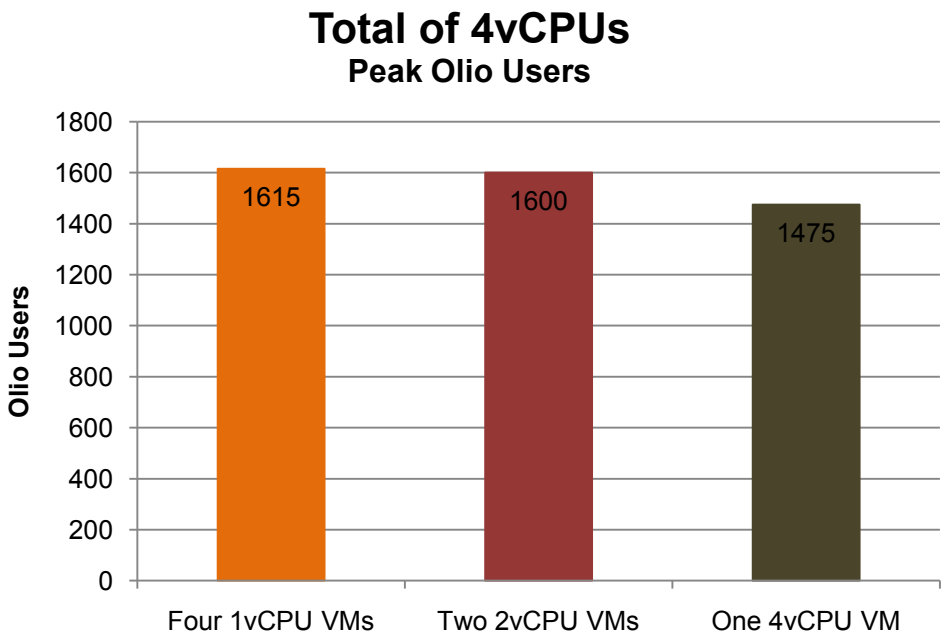
At CPU utilizations beyond 90%, the response-times curves begin to diverge. This is most likely a result of the additional memory that is available as Java heap when more VMs are used. The memory configuration of the VMs is shown in Table 1.

Table 1. Java Heap Configuration with Total of 4 vCPUs

Number of vCPUs per VM	Number of VMs	Per-VM Maximum Heap Size	Total Heap for 4vCPU Case
1	4	2GB	8GB
2	2	2.5GB	5GB
4	1	4GB	4GB

The peak throughput for these three cases is shown in **Figure 9**. The results for the 1-vCPU and 2-vCPU cases are quite similar, while the 4-vCPU case is about 8% lower. The reasons for this drop are explored in Section 4.3, "Peak Throughput Scaling," However, this difference in peak throughput does not correlate to increased response-times at lower loads. The use of a 4-vCPU VM is still a valid choice for this workload as long as the expected peak loads will not push the CPU utilization beyond 80%.

Figure 9. Peak Throughput with Total of 4vCPUs



4.3 Peak Throughput Scaling

In this section, we look at the peak-throughput scaling for configurations using from 1 vCPU up to a total of 16 vCPUs. The server on which the tests were run had a total of 16 physical cores. We used VMs with 1, 2, and 4 vCPUs. Beyond 4 vCPUs, the per-VM network load exceeded the capabilities of a single gigabit Ethernet link.

Figure 10 shows the peak throughput for configurations with increasing numbers of total vCPUs. At each number of vCPUs, this chart shows the peak throughput for configurations using 1, 2, and 4 vCPU VMs. For example, the points for 8 total vCPUs represent configurations with eight 1-vCPU VMs, four 2-vCPU VMs, and two 4-vCPU VMs. The same results are shown in Figure 11 normalized to a single 1-vCPU VM. These results show near linear scalability through eight total vCPUs at all VM sizes, and very good scaling out to use all 16 physical cores.

Figure 10. Scaling of Peak Throughput

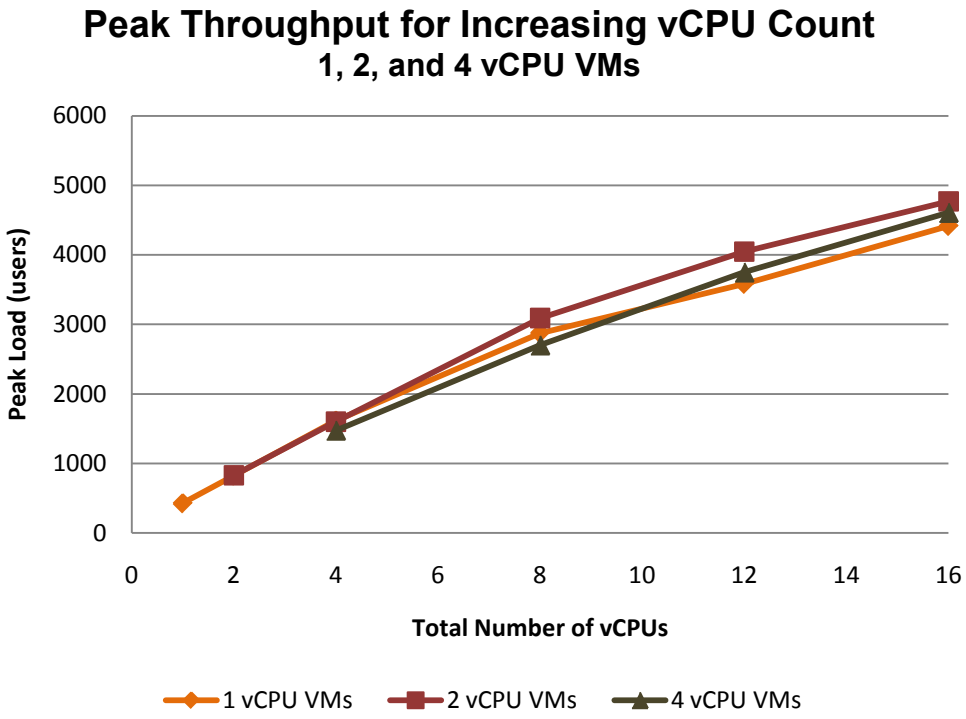
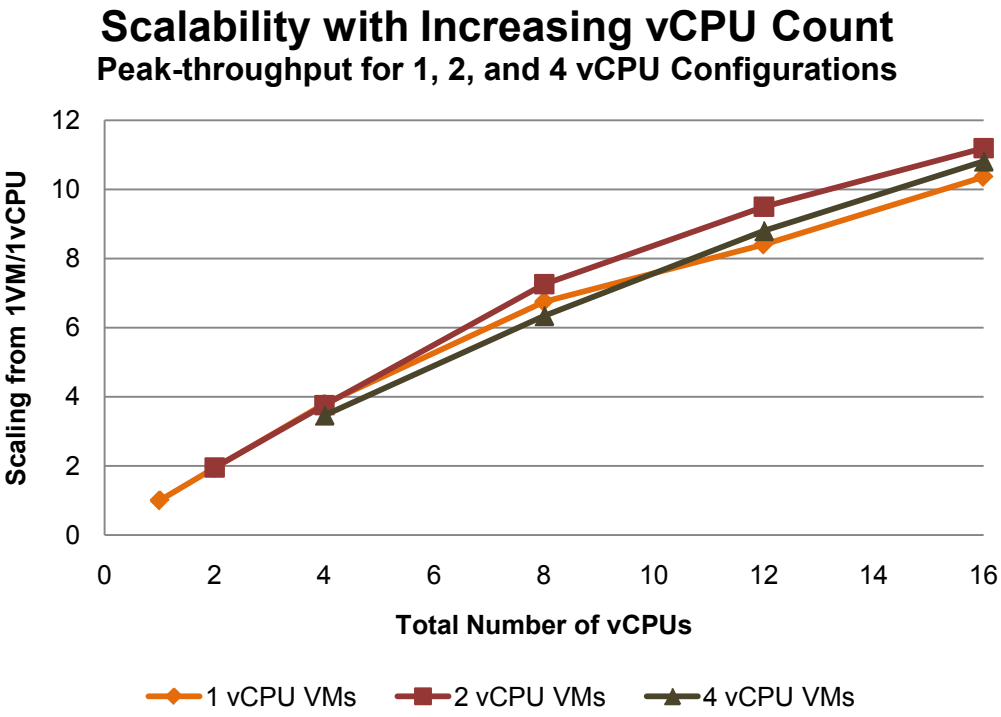


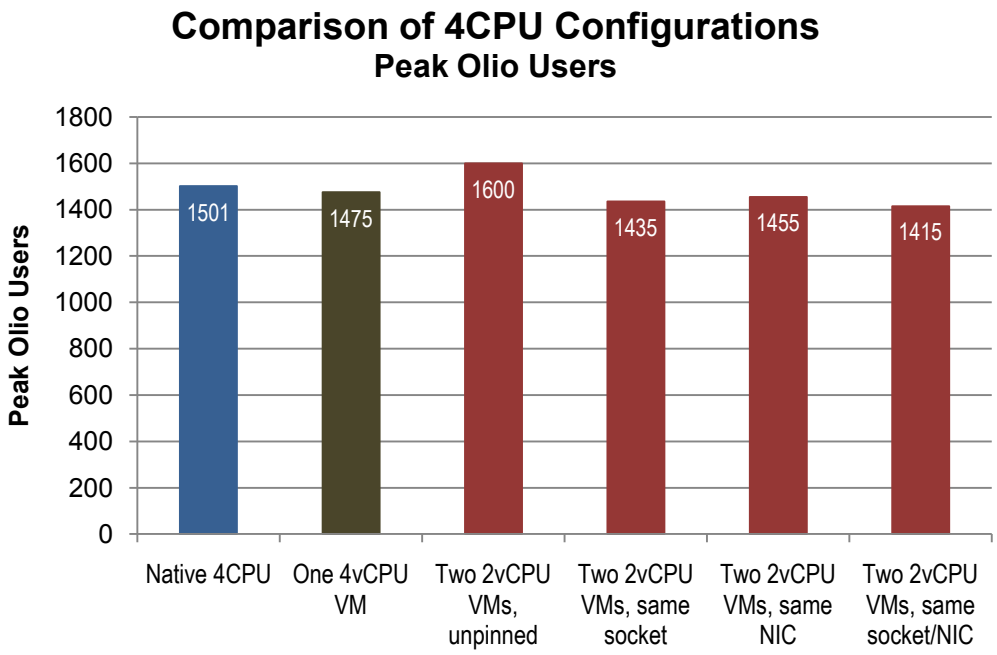
Figure 11. Scalability with Increasing vCPU Count



There are many reasons that peak throughput may not scale linearly as the number of CPUs in a single instance, or the number of instances, increases. At the most basic level, non-linear scaling is due to some resource approaching saturation as load is increased. In these tests, there are many shared components which could act as a bottleneck. For example, the response-times of the support services, such as the database and filestore, are key components of the overall operation response-times. There can also be limits imposed by the effect of the increased load on the hardware configuration. The most basic components impacted by the increase in load are the processor resources (for example, caches, TLBs, and memory controllers) shared by the individual processor cores. In these tests, the high network load also makes the shared NICs a potential bottleneck.

Figure 12 shows how the increased load on these components can impact peak-throughput in configurations with a total of 4 vCPUs. The first two points show the peak throughput for native and virtualized configurations with 4 CPUs. The third point shows the peak-throughput for two 2-vCPU VMs. In this case, each VM has its own Ethernet link, and the scheduler in VMware ESX 4.1 ensures that the VMs are scheduled on separate processor sockets. The result is that the peak-throughput is higher than for the 4-CPU cases. The remaining points show the impact of pinning the 2-vCPU VMs to the same socket or having them share the same NIC. In both cases the peak throughput drops to essentially the same level as the 4 CPU cases. This implies that both the shared processor resources and shared NIC are bottlenecks in the native and virtualized 4 CPU cases. It would be necessary to increase capabilities of both resources in order to improve peak-throughput for the 4 CPU configurations.

Figure 12. Impact of sharing processors and NICs, total of 4vCPUs



4.4 Discussion

The results in this section show that, with respect to performance, there is considerable flexibility when deciding between scale-up and scale-out configurations for Java applications. While the peak-throughput of the configurations may differ slightly, there is essentially no performance difference at more typical loads. The results also show that enterprise-level Java applications can scale-out effectively to use the available server resources. While limits to scalability can exist, they are similar in both native and virtualized deployments.

5 Conclusion

The combination of VMware vSphere 4.1 and SpringSource tc Server provides a powerful and flexible platform for virtualizing enterprise Java applications. The results of the tests discussed in this paper show that applications can be virtualized on this platform while maintaining a user experience very similar to native deployments on equivalent hardware configurations. In addition, the results show that excellent performance can be achieved on both scale-up and scale-out configurations.

There are a number of important lessons from these results that can be applied to deploying and benchmarking Java applications on VMware vSphere 4.1:

- When selecting the total VM resources for a Java deployment, including number of vCPUs and memory size, it is important to provide sufficient resources to keep the CPU utilization of the VM at reasonable levels even during periods of peak load.
- It is important to understand the scaling of demands placed by the application on the VM infrastructure when choosing between a scale-up or scale-out approach to Java application deployment. In particular, scaling-up beyond a certain point may cause the load to exceed the bandwidth or throughput limits of a VM's NICs or storage adapters.
- When virtualizing a Java application onto an ESX host or cluster supporting other applications, the shared resource effects discussed in Section 4.3, "Peak Throughput Scaling," can impact the performance of a newly virtualized application. This has the following implications:
  - Whenever possible, initial performance testing of a virtualized application should be done on an otherwise unloaded ESX host. This will eliminate the impact of shared resource effects.
  - When investigating performance issues, it is important to understand the loads on all shared resources, and not only on the VM under investigation.
- When comparing the performance of native and virtualized Java deployments:
  - Compare performance at multiple loads, including performance at loads that represent expected operating conditions for the application when deployed in production. A comparison of peak-throughput serves to uncover the saturation point of the application/infrastructure combination, but does not provide information about the user experience at more reasonable loads.
  - Always ensure that the underlying infrastructure, including server hardware, provides comparable performance. Comparing an application running natively with on a server with a certain number of CPU cores to a VM with a different number of vCPUs will give erroneous results.

6 About the Author

Hal Rosenberg is a performance engineer at VMware. His focus areas are Middleware, Java, and performance troubleshooting. Prior to coming to VMware, he had over 10 years of experience working on performance engineering and analysis for hardware and software projects at IBM and Sun Microsystems. Hal will be blogging with additional information about this paper and other performance topics at http://communities.vmware.com/blogs/haroldr.

7 Appendix

7.1 Olio Workload Details

The operations included in the Olio workload, as well as the mix percentage and response-time requirements, are described in **Table 2**. Note that for each operation there is a QOS requirement defined in terms of the 90th percentile response-time. To meet this requirement, 90% of the operations during a benchmark run must have a response-time of less than this limit. In order for the run to be considered a passing run, all operations must meet their response-time requirements, and the average cycle-time must be within a small percentage of 5 seconds. The cycle-time is the total of the time required to complete the operation plus the think-time until the next interaction. The cycle-time requirement is important because it ensures that the workload driver is placing a constant load on the application for a given number of users, even if the response-times increase.

Table 2. Olio workload description

Operation	Percent of Mix	90% Response-Time Requirement	Description
HomePage	26.15%	1 Second	Retrieves the home page including all static content and thumbnails.
Login	10.22%	1 Second	Logs into the application as a particular registered user chosen at random.
TagSearch	33.45%	2 Seconds	Searches for events with a particular tag.
EventDetail	24.68%	2 Seconds	Retrieves the EventDetail page for the specified event. The event is chosen at random from the list of events displayed on the home page.
PersonDetail	2.61%	2 Seconds	Retrieves the PersonDetail page for the specified user. The user is chosen at random from the list of registered users.
AddPerson	0.84%	3 Seconds	Registers a new user. All of the user information is generated using random values.
AddEvent	2.84%	4 Seconds	Adds a new event. All of the event information is generated using random values.

The Olio workload driver simulates the behavior of a fixed number of active users interacting with the Olio application. Each simulated user repeatedly steps through a sequence of interactions with the application. An interaction includes one of the operations from **Table 2** plus a think-time selected to maintain the constant cycle-time.

7.2 Testbed Details

7.2.1 Hardware Configuration

Table 3. Hardware configuration

Component	Details
Drivers	
System Model	Dell PowerEdge 2950
Processors	Two Intel Xeon 5160 @ 3GHz, 4 total cores
Total Memory	8GB
System Under Test (SUT)	
System Model	HP ProLiant DL585 G5
Processors	Four AMD Opteron 8382 @ 2.6GHz, 16 total cores
Total Memory	64GB
Network Controller	Intel NC364T quad-port NIC, all four ports connected to the network switch
Storage Controller	QLogic ISP2432 4Gbps Fibre Channel HBA
Storage Configuration	- Native OS and ESX 4.1 booted from an EMC CX500 Fibre Channel storage array - VMs stored in a VMFS3 datastore on an EMC CX500 Fibre Channel storage array
SUT/VM Configuration	
Number of vCPUs	1,2, 4
VM Memory Size	- 5GB with 1 or 2 vCPUs - 6GB with 4vCPUs
Virtual NIC	vmxnet3
Database	
System Model	HP ProLiant DL380 G5
Processors	Two Intel Xeon X5460 @ 3.16Ghz, 16 total cores
Total Memory	32GB
Storage Controller	QLogic ISP2432 4Gbps Fibre Channel HBA
Storage Configuration	Data stored in a 10 disk, RAID10, LUN on an EMC CX3-10 Fibre Channel storage array
Filestore	
System Model	HP DL580 G5
Processors	Intel Xeon X7350 @ 2.93GHz, 16 total cores
Total Memory	128GB
Storage Controller	QLogic ISP2432 4Gbps Fibre Channel HBA
Storage Configuration	NFS mounted on a 10 disk, RAID0, LUN on an EMC CX3-10 Fibre Channel storage array

7.2.2 Software Configuration

Table 4. Software configuration

Component	Details
Drivers	
Workload Driver	Faban 1.0
tc Server for Geocoder	tc Server 2.0
JVM Version	Sun JDK 1.6.0_19 64-bit
Operating System	Red Hat Enterprise Linux 5 Update 4 x86_64
Additional Notes	The primary driver ran the DNS server
SUT	
Operating System	Red Hat Enterprise Linux 5 Update 4 x86_64 When running natively, booted with kernel options: `mem=6144M numCPU=1`
tc Server Version	tc Server 2.0
Java Libraries	Eclipselink 2 for JPA support Spring 3.0 for transaction management and dependency injection
JVM Version	Sun JDK 1.6.0_19 64-bit
JVM Parameters	1CPU: `–Xmx2048m –Xms2048m –Xss192k –XX:+UseLargePages`
	2 CPUs: `–Xmx2560m –Xms2560m –Xss192k –XX:+UseLargePages`
	4 CPUs: `–Xmx4096m –Xms4096m –Xss192k –XX:+UseLargePages`

VMware ESX Version	ESX 4.1, Build 260247
	The Olio application has very high network demands. As a result, we used the following network-level tunings.
	`/adv/Net/vmxnetThroughputWeight=24 (default 0)`
	`/adv/Net/MaxNetifTxQueueLen=750 (default 500)`
	`/vmkernel/netPktHeapMaxSize=128 (default 0)`
	`/vmkernel/netMaxPCPUPktCacheSize=256 (default 128)`

Database	
Database	Percona Server 5.1.47 (MySQL 5.1.47 with the Percona XtraDB storage engine)
Operating System	Red Hat Enterprise Linux 5 Update 4 x86_64
Filestore	
Operating System	Red Hat Enterprise Linux 5 Update 4 x86_64
Additional Notes	Mount exported using NFS v3