Best Practices for Enterprise Java Applications Running on VMware

vmware®

Table of contents

Introduction

Executive Summary

The purpose of this document is to provide guidelines for deploying enterprise Java applications on VMware vSphere. The recommendations in this document are not specific to any particular set of hardware or to the size and scope of any particular implementation. The best practices in this document provide guidance only and do not represent strict design requirements due to the fact that enterprise Java application requirements differ greatly. However, the guidelines presented in this document may help to form a solid foundation for customers to successfully virtualize enterprise Java applications.

This document provides information about best practices for deploying enterprise Java applications on VMware, including key considerations for architecture, performance, design and sizing, and high availability. The information presented should help IT professionals to successfully deploy and run Java environments on VMware vSphere™. For specific Java best practices, always refer to a vendor documentation for specific Java runtime engines.

NOTE: It is important to realize that virtualizing enterprise Java applications does not require a major change neither in architectural design nor in Java programming. Furthermore, any performance enhancements done on physical environments are transferrable to the vSphere-deployed instances of Java applications. Simply take existing Java applications and move it over to a VM without a need for any changes. In this document VMware is sharing with customers several best practices developed over the years in running virtualized Java applications.

Intended Audience

It is assumed that the readers of this document possess basic knowledge and understanding of VMware vSphere along with understanding of common knowledge of enterprise Java applications.

Architectural staff may utilize this document to gain an understanding of how a system works while the design and implementation of various components take place.

Engineers and Application Platform teams may also find this document useful as a catalog of technical capabilities.

Primary Topics

Architectural Best Practices for Enterprise Java Applications running on vSphere and best Practices for Enterprise Java Applications running on vSphere. The following topics are overviewed:

- Design and sizing of VMs,
- Guest OS recommendations,
- Fine tuning of CPU, memory, storage, networking,
- Useful JVM (Java Virtual Machine) tuning parameters,
- Various high availability features in vSphere ESXI host clusters,
- Resource pools (horizontal scalability and vertical scalability),
- VMware Distributed Resource Scheduler (DRS).

Enterprise Java applications on vSphere platforms

More than 50% of enterprise applications are written in Java and were deployed in the last 5-25 years. Today, the JVM deployment environments still exist and thriving. Typical enterprise Java application are multitiered and multi-organizational;

therefore, a change in one tier often causes a ripple effect across all tiers. The decisions related to these changes affect multiple organizations of an enterprise.

A highly scalable and robust Java application has all of these tiers running in VMware vSphere in order to reach full benefits of scalability features offered by vSphere.

Enterprise Java applications are made of four main tiers. These tiers are the following:

- Load Balancers tier
- Web Servers tier
- Java Application Server tier
- DB Server tier and/or In-memory DB tier and/or NO-SQL DBs

Figure 1 depicts a multi-tier fully virtualized enterprise Java applications architecture running on VMware.

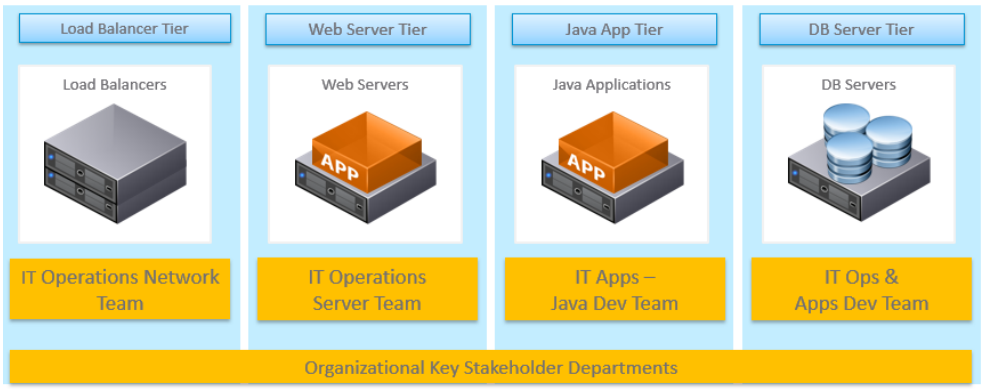

Figure 1. Multi-tier Virtualized Enterprise Java Application Architecture

Key architectural attributes of each tier include:

As depicted by Figure 1, each one of these tiers is running in a VM that is managed by **VMware vSphere**, which forms the key building foundation. Best practices are discussed in this document for vSphere features such as VMware HA, DRS, VMware vMotion™, resource pools, hot plug/hot add, networking and storage.

Load Balancer tier– Increasingly feature-rich load balancers are available that provide various load balancing algorithms and API integration with VMware vSphere. This allows the enterprise Java application architecture to scale on demand as traffic bursts occur.

Web Server tier– the web server tier must be appropriately tuned, with the right number of HTTP threads to service your anticipated traffic demands. Typically, this tier includes servers such as Nginx, Apache and others.

Java Application Server tier– many of the commonly used application servers have mechanisms to tune the Java engine to meet traffic demands. If the available Java Threads is already tuned - JDBC configurations and various JVM and GC parameters on physical machines, the tuning information is transferrable as is for the vSphere deployment of enterprise Java applications.

DB Server tier– critical to meeting the uptime SLA of enterprise Java applications is having an appropriate high availability architecture for the DB server. DB servers can benefit from running on vSphere.

Java Platform Categories

In general, Java platforms can be divided into three categories. If the presented categories are ignored, then either SLAs are missed and/or the infrastructure is excessively overprovisioned. Frequently, this overprovisioning means twice or three times waste of the resources.

NOTE: It is important to remember, that virtualizing enterprise Java applications does not require a major change in Java coding paradigm. Furthermore, any performance enhancements done on physical environments are transferrable to the vSphere-deployed instances of Java applications.

Category 1: 100s to 1000s of JVMs. (Web applications and microservices)

This Category 1 is distinguished by its large number of JVMs, many different types of enterprise Java applications/services accommodating various lines of business; for example, the JVM (or set of JVMs representing one application) act as a LOBs tenancy construct. In this Category 1, hundreds to thousands of JVMs (think of them as application tenants) are deployed on the Java platform. These JVMs typically function within a system that might be servicing millions of users. The Java virtual machine (JVM) is a virtual "execution engine" instance that executes the bytecodes in Java class files on a microprocessor.

Here are the general considerations for Category 1:

- In Category 1 to reduce VM sprawl you may have to run by using multiple JVMs on each VM. When doing so, please ensure that different apps/JVMs from separate LOBs are not deployed on the same VM. Ignoring this best practice would imply that potentially two or more JVMs for different LOBs are deployed on the same VM. For example, if a VM needs to restart or to be updated, it would cause downtime for both LOBs.
- Multiple JVMs per VM are acceptable; however, proceed with caution as multiple JVM instances will compete for the same set of CPU resources.
- Use cluster or shared clusters with some Resource Pools' construct to separate Category 1 from Category 2.
- Use Resource pools to manage different LOBs.
- Consider small JVM standard at 4GB heap in order to eliminate the sprawl of many 1GB JVMs, 4GB heap implies 4.5GB Java process, and 5GB for VM.
- vSphere hosts with <512 GB RAM may be more suitable, as many JVM instances are stacked - it is possible to reach CPU boundary before all the RAM is consumed.
- Consider using 4 socket servers with more cores. Since there are many more JVM process instances in Category 1 (more than in Category 2) - it is more likely to reach CPU contention vs. being memory-constrained. Therefore, think about purchasing a cost-optimal host hardware that has more CPU cores. It prevents purchasing excessive amount of memory that is most likely won't be completely utilized by an ESXi host. In short, save money on memory and buy more CPU cores. However, it's always possible since it depends on the ESXi clusters configured a certain way, such as a dedicated cluster resources.
- Understand the NUMA architecture; NUMA building block size becomes the divisor for all calculations. It's important to establish a well-tested building block VM and JVM. There may be one building block per application group, where some may have 3 building blocks, small-medium-large etc. To calculate a NUMA optimal building block VM, use the following formula:

*NUMA Optimal building block VM Size = (0.85 * Total RAM on Host) / 4 (number of sockets)*

*NOTE: The headroom factor of **0.85** is a practical approximation for the reserved capacity for ample vSphere headroom of 15%, tested for mission-critical applications with deterministic performance such as trading systems* (PKS on the VMware SDDC, 2019).

For example, if a vSphere host with 512GB RAM and with 4 sockets is chosen with 10 cores per socket, then:

*NUMA Optimal building block VM Size = (0.85 * 512GB) /4 = 108.8 GB ~108 GB*

Therefore, the VM size will be 108GB of RAM and 10 vCPUs, assuming 1 vCPU = pCPU depending on the workload and, if it is suitable for overcommitment. On this 4-socket host, it is possible to place 4 of such VMs (or have 8 VMs or more, but with a total aggregate memory of 108GB across all VMs in each NUMA node). In order to fully utilize all the RAM for each VM, the number of JVMs would be:

108GB/5 GB =21.6 ~ 21JVMs, which result in 2.1 JVMs per pCPU.

Notice that 2.1 JVMs would clearly reach the CPU NUMA boundary, with an assumption that there are 10 pCPUs running per socket – resulting in 2.1 JVMs per core that would most likely cause CPU contention and slow response times for an application. To remediate the problem, relocate JVMs to other hosts; unless, there is a way to consolidate these JVMs into larger JVMs, especially if they are of the same application type.

In some rare cases, if the JVMs are not very busy, having 2.1 JVMs against one pCPU core may be suitable, especially for non-production workloads. For production workloads it is recommended to have at least 2 vCPUs per JVM as a starting point - if the testing has been performed to confirm the desired performance. If the tests conclude that the total pCPU consumption of the ESXi host is still low, further overcommitment of vCPUs with more JVMs may be possible. To compare the peak CPU cycles of the of the bare metal deployment, measure *JVM-per-CPU-cores-of-bare-metal* (per GHz) and compare that to the intended vSphere configuration. Start with the *JVM-per-CPU-cores-of-bare-metal and if there are ample ESXi CPU cycles, then increase the number of JVMs per CPU beyond this starting ratio.*

NOTE: In this example, a busy JVM is the one that would have reached at least 80% of the heap memory and used at least one vCPU.

For additional information on CPU and memory sizing refer to the NUMA Local Memory section of this document for further considerations.

Category 2: Up to 20 of very large JVMs. (In-memory databases)

This Category 2 is distinguished by a fewer number of JVMs (1-20 JVMs of the same application or application cluster) but with a large heap size (8GB to 256 GB or higher).

NOTE: In this category, Garbage Collection (GC) tuning becomes critical. Please refer to the GC Tuning Considerations section of this document for further considerations.

General considerations for Category 2 are the following:

- 1 JVM per VM. For example, if this an in-memory DB, then it would be one cache node per VM.
- Fewer JVMs less than 20. Always vertically scale to consume the amount of the available NUMA memory and then consider scaling out. See the NUMA Local Memory section of the document.
- Very large heap sizes - 8GB to 256GB. Examples are in-memory databases, messaging systems and backing data services.
- Choose 2 socket vSphere hosts, and install ample memory 128GB to 1024GB.
- Always deploy 1 VM per NUMA node and size to fit perfectly. To determine optimal number of JVM/Cache nodes in an in-memory cluster, take the total memory in the in-memory-cache cluster and divide it by the largest NUMA bound building block VM that fits.
- Dedicated vSphere cluster is preferred, but if such cluster is not feasible then leverage a resource pool and reservation to ensure the in-memory database cluster has the needed resources.

Category 3: Category-1 JVMs accessing data from Category 2 JVMs

Category 3 is a combination of the Category 1 and 2, where perhaps thousands of JVMs run enterprise applications that are consuming date from the Category 2 types of large JVMs in the back end.

General considerations for Category 3 are the following:

- Many smaller JVMs accessing information from fewer large JVMs
- Category 3 is a golden category. Every enterprise has one or will eventually have one.

VM Memory Sizing

To understand how to size memory for a VM it is also important to understand the memory requirements of Java and various memory segments of the JVM.

JVM Memory Areas

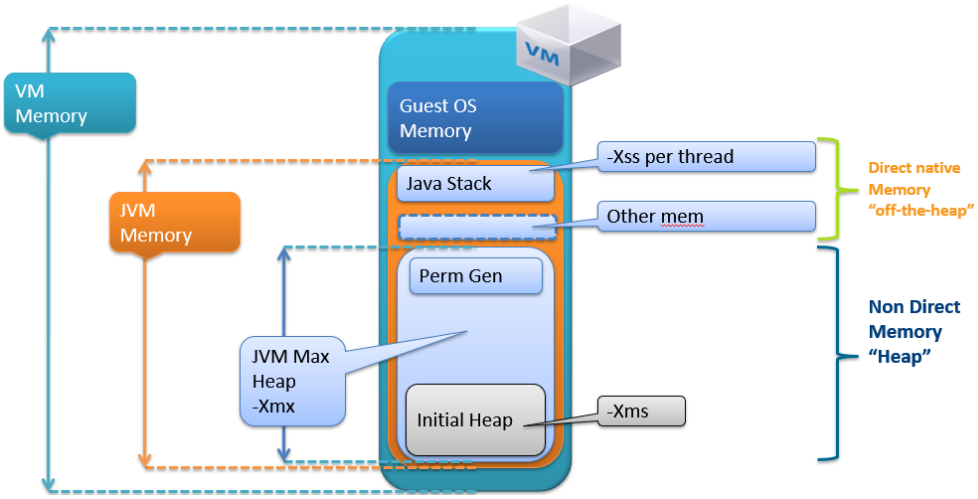

Figure 2. HotSpot JVMs on VMware vSphere

Figure 2 details the following information on separate memory areas in JVMs:

- In general, Java heap is where Java objects reside. Such objects include live, dead and free memory. When JVM starts, JVM heap space is equal to the initial size of heap specified by **-Xms** parameter, as application progress more objects get created and heap space is expanded to accommodate new objects.
- **-Xmx** is the maximum heap value. **-Xmx** value prevents the heap from growing too large.
- **-Xss** is the Java thread stack size, which typically equals to 512KB. The default is OS- and JVM-dependent, and it can range 256KB to 1MB. Each allocated Java thread will use up to "**-Xss**" worth of memory that is not part of the Java heap, i.e. it does not get deducted from the available **–Xmx** value. Instead, it is allocated natively and therefore the memory is provided by the OS. If there is an N number of concurrent threads, **Nthreads * -Xss** is the memory burden on the Guest OS.
- **PermGen** is an area that is not Garbage Collected (GCed) because it contains class-level information. Typically, this is a small area but it can be 256MB-512MB at times.

- "**Other mem**" is additional memory required for NIO buffers, JIT code cache, classloaders, socket buffers (receive/send), JNI, Garbage Collection (GC) internal info.
- **Guest OS Memory** is greater or much higher than 1GB. It depends on the OS type and other processes. Perform an analysis of all the processes and their memory consumption at peak times to determine total memory needed by the guest OS.

Let's review the following formulae to better understand the information presented by Figure 2:

VM Memory = Guest OS Memory + JVM Memory

*JVM Memory = JVM Max Heap (-Xmx value) + NumberOfConcurrentThreads * (-Xss) + "other mem"*

If multiple JVMs are present (N JVMs) on a VM, then consider the following formula:

*VM Memory = Guest OS memory + N * JVM Memory*

If JVM Heap (-Xmx) is Y, then JVM Memory is (1.1 to 1.25) * Y.

For example, if heap is 4GB then JVM Memory = (1.1 to 1.25) *4= 4.5 to 4.8GB

NOTE: To precisely size the memory, a load test is needed for a Java application for additional memory requirements that may be allocated due to NIO buffers, JIT code cache, classloaders, and verifiers. For instance, examine Figure 2 that displays Other Mem where NIO buffers, JIT code cache, classloaders, and verifiers reside. The content of a direct buffer is allocated from the guest operating system memory instead of the Java heap, and non-direct buffers are copied into direct buffers for native I/O operations. Some Java applications may be using NIO buffers, which can have massive additional memory demands. NIO buffers is memory allocated outside of heap space. If the application is using direct buffers that are allocated outside of the heap but still within a Java process, then use monitoring tools to measure the actual Java process memory. Use load testing to appropriately size the effect of these buffers.

Four Gigabyte is the New 1GB

About 25 years ago when 32-bit JVMs would only allow up to 2GB (varies with OS versions), many systems were written and this JVM level restriction caused excessive scaling out of JVM processes in order to accommodate more memory across large applications. For example, if an application needed 100GB of heap in total to accommodate normal business activity, there may be 100 JVM processes each of 1GB heap because of the 32-bit heap size restrictions. Of course, with 64-bit JVMs that we currently have that same application in the example could be easily modeled as 25JVMs of 4GB heap each for far better performance and scale. Since the 32-bit days business applications have been improved due to 64-bit capabilities and newer JVM versions, but yet the number of JVMs instances hasn't been consolidated. 4GB is a unique number in 64-bit JVMs which is a natural 2^32: even though app is running in 64-bit JVM, its memory throughput is optimized/compressed to 32-bit footprint.

Note: There is a historical legacy trend from the 32-bit JVM days that continues to linger the industry. It is the year 2020, and no one should be using 32-bit JVMs. When JVM vendors introduced 64-bit JVM, the operand was double in size in 32-bit operand versus the newly added 64-bit. That potentially meant that the same application running in 32-bit happily under 1GB heap could now use more memory because of the 64-bit operand representation. However, JVM vendors were aware of this issue and created compression algorithms that compress the 64-bit operand down to 32-bit operand when possible, even though it is a 64-bit JVM. This compression algorithm would have some CPU cost. Anything less than 4GB is a natural 2^32 bit and there is zero cost to the compression algorithm (with JVM option ***-XX:+UseCompressedOops****). Beyond 4GB and up to 32GB (64GB in some cases) there is a variable cost, but mostly not very significant (for actual cost, always refer to the official JVM vendor documentation).*

Here are some unique facts about 4GB JVMs:

- At almost no additional cost a JVM can grow from 1GB to 4GB quite easily. This will allow the rationalization of the number of JVMs along with consolidation to a more manageable total number of JVMs. For example, if there are 1000 JVMs of 1GB each all servicing the same application, then **the same application can run within 250 JVMs of 4GB each** effortlessly and with substantially lesser CPU usage, while improving response time performance.
- If going to 4GB eliminates many unnecessary 1GB JVM instances, then always consider eliminating unnecessary 1GB JVM instances. In many cases, 1GB JVMs could be a part of many of the memory pressure and performance issues. For more information on this topic, refer to the Ten Rules of Virtualizing Enterprise Java Applications section of this document, specifically, Rule 8 and Rule 9.
- The 4GB JVM will also have 3 to 4 times much larger Young Generation leading to much more improved response time performance. And that is to contrast the 1GB heap space which has a very small 100MB to 300MB Young Generation space.

Oracle Sun HotSpot JVM and the IBM J9 JVM have compression algorithms to compress code operands within 64-bit JVM down to 32 bits. The compression is as follows:

- 2^32 bit, i.e. 4GB its automatic.
- In Java 8 the limit goes up from 32GB to 64GB.
- Beyond 64GB it is off (except in java 8 in some advanced modes it could be higher).

Practical limits of JVM memory sizing

Consider the following suggestions on practical limits for JVM Memory sizing:

- High JVM vertical scalability limit, 100s of GBs of heap sizes are practical with even higher theoretical limits.
- When required, an effective scale out approach.
- More than 50% of enterprise applications are written in Java in the last 5-25 years.

On Figure 3, the practical limits of JVM Memory sizing are depicted.

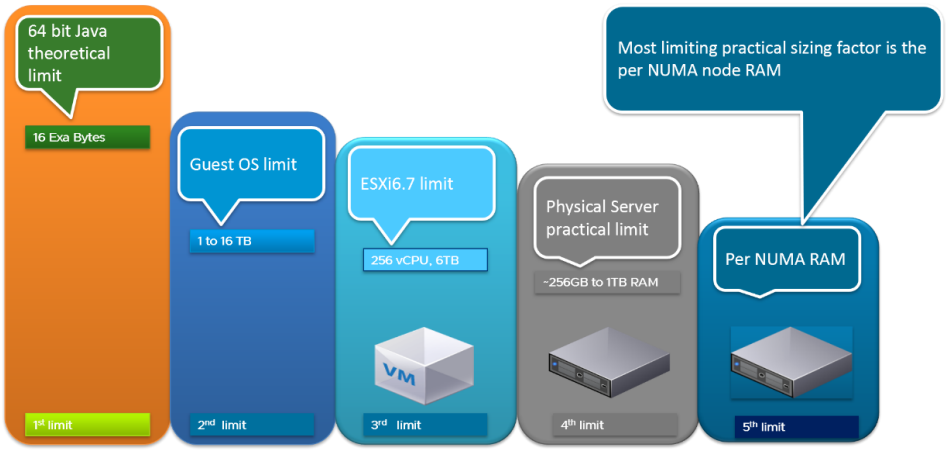

Figure 3. Practical limits for JVM memory sizing.

GC Tuning Considerations

Over the years, VMware has helped many customers with their GC tuning activities, even though GC tuning on physical platforms is not different from tuning on virtual platforms.

JVM tuning may take many weeks of discussions on theory and practice. When opting to increase the size of the JVM by increasing the heap space/memory, GC tuning knowledge becomes very important for handling large JVMs. GC tuning for different JVM sizes may need expert knowledge as the size of JVMs increases. Many considerations come with sizing very large JVMs, such as GC tuning complexity and knowledge needed to maintain large JVMs. Review Figure 4 for GC tuning requirements of different JVM sizes.

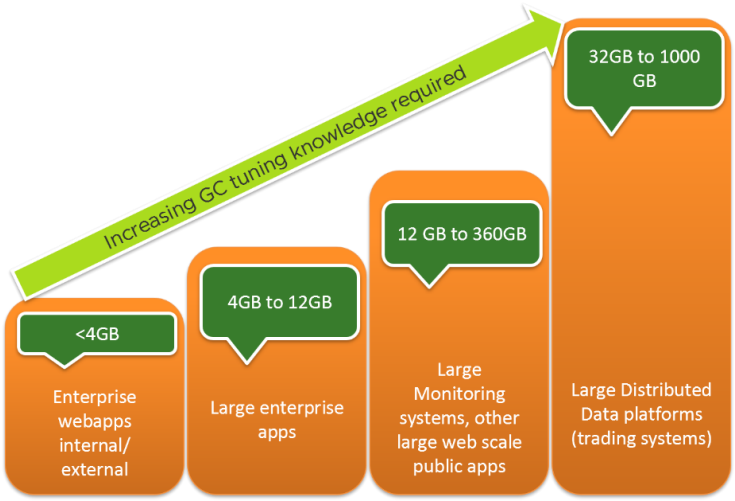

Figure 4. GC tuning knowledge for each JVM size

Most JVMs within VMware customer base are in the vicinity of 4GB of RAM for the typical enterprise web application, or what has been referred to in this document, the Category 1 workloads. However, larger JVMs exist (such as Category 2), and VMware has customers that run large-scale monitoring systems and large distributed data platforms (in-memory databases) on JVMs ranging from 4GB to 320GB and even higher; where total cluster size can be 1TB to 3TB in-memory.

With large JVMs comes the need to have an expert knowledge of GC tuning - typically using a combination GC such as:

(-XX:+UseConcMarkSweepGC and XX:+UseParNewGC) or G1GC. With either case, some of the largest financial trading platforms in the world are ran on configurations that are implemented by VMware.

Ten Rules of Virtualizing Enterprise Java Applications

Enterprise Java applications are highly customizable, but performance testing should be done to establish best sizing. There are several rules that are associated with Distributed Java Platforms.

NOTE: This section of the document presents the rules in no particular order; each rule should be treated with equal importance.

Rule 1 - Understanding memory sizing

- Always use memory-based sizing. For example, if a system needs 400GB made from 100 JVMs of 4GB each, then this is the available capacity. If migrating from one system to another, once machines have been sized for an adequate memory-

based on Java heap configured across all JVM instances, ensure that peak CPU availability is ample by comparing the current JVM to CPU ratio of existing system as the starting benchmark for CPU consumption; additionally, if the underlying VM and ESXi host CPU is plentiful (<80% CPU) then be more aggressive on the JVMs to vCPUs ratio.

- Always set initial heap (-Xms) equal to max heap (-Xmx). In production environments, set the minimum heap size and the maximum heap size to the same value to prevent wasting VM resources used to constantly grow and shrink the heap.

NOTE: It is not recommended to set -Xms initial heap at much lesser value the max heap value. Almost always this practice is not recommended since the max heap (-Xmx) value would have to be pre-configured ahead of time, against the VM's memory total size. If the memory has been pre-configured, and available to the VM, then why not offer it to the JVM from the beginning by setting -Xms = -Xmx? Additionally, if the -Xms is not equal to -Xmx, any time the initial heap window needs to be resized, it will trigger a full GC activity which may impact performance.

Rule 2 - Multi-tier sizing rule decisions in one tier impacts the next

- Sizing of each tier has impact on next tier, scaling up and/or out of one tier has downstream impact on next/all tier(s).
- Expand each tier proportionally to avoid bottlenecks or throttling.
- Use separate load balancer pools to isolate functional application groups.
- Count the total number of threads of each application container and determine the best the ratio of the healthy threads to the size of the DB connection pool.

Rule 3 - Understanding various workload platform categories

- Category 1 is the most common in the world presently, suffers from fragmentation of JVMs.
- Category 2 in-memory DBs, avoid too many JVMs.
- Category 3 is a golden category, every enterprise has one, or will eventually have one.

Rule 4 - Understanding how to build a scalability building block

- It's important to establish a well-tested building block VM and JVM. There may be one building block per application group, where some may have 3 building blocks, small-medium-large etc.
- Build new system based on the building block VM/JVM/Container/Pod VM
- Minimize the number of JVMs and avoid the costly administration of JVM sprawl
- All JVM/VM configurations should be based on the building block.

Rule 5 - Big VMs and big JVMs are real; don't ignore NUMA

- For Category 2 platforms, try to achieve 1 JVM per VM per NUMA node.

Rule 6 - Understand when to scale up vs. scale out

- Scaling out can be costly. In Category-1, sometimes far too many JVM instances are used to represent the same functional code which may lead to an inferior performance and scalability by fragmenting the compute space. For example, one application may have 16 JVM instances with each instance of 1GB heap; however, a preferred approach would be to use 4 JVMs of 4GB each where the net heap services memory is still 16GB, but it is achieved with just 4 instead of 16 JVMs. This approach would substantially reduce CPU, and JVM management costs.

Rule 7 - Avoid mixing workloads with different behavior in the same JVM

- Example: Web-App should not be deployed in the same JVM as a batch job or a reporting engine. This avoids the noisy neighbor issues and someone running heavy demand reports which may impact the response times of the web application deployed in the same heap space.

Rule 8 - Do not limit heap size to 1GB only

- Remember – 64-bit JVMs do not have a 1 GB heap limitation.
- Systems that were written and deployed under the 32-bit restriction of small heaps, had the code moved to 64-bit standard, but the deployment pattern, and the number of JVM instances didn't change.
- 1GB JVMs are bad for performance since frequently, 100-300MB is used for Young Generation that leads to a performance degradation for production-grade apps.
- If there are many identical, scaled-out copies of 1GB heap JVMs, consider consolidating those JVMs with larger heap memory sizes.

Rule 9 - The 4GB is the new 1GB

- 4GB is a unique number in 64-bit JVMs. This is a natural 2^32; even if an app is running in 64-bit JVM its memory, throughput is optimized/compressed to 32-bit footprint.
- The 4GB JVMs have 3 to 4 times much larger young generation, leading to much better response time and performance.
- Both Oracle Sun HotSpot JVM and the IBM J9 JVM have compression algorithms to compress code operands within 64-bit JVM down to 32 bits.

Rule 10 - GC knowledge is important as heap increases

- It is acceptable to have large JVMs with a proper GC tuning knowledge.

HotSpot JVMs on Docker Containers and vSphere

One of the best approaches in sizing Docker containers in vSphere environments is to allocate 1 JVM to 1 container to 1 VM/POD. With container orchestrators, such as Kubernetes, this approach becomes the de facto approach to allocate CPU and memory.

NOTE: In Java 10 and later versions, there is an automatic detection of CGroup limits.

Prior to Java 8 update 131, Java did not recognize CGroups, which meant the following:

- Multiple containers on an OS each with a JVM inside of them would see all of the compute space since one container is not aware of the presence of the other container. That often leads to many premature JVM Out of Memory (OOM) errors.

Java8u131 and Java9:

- **-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap** to adhere CGroup memory limits.
- **-XX:ParallelGCThreads** or **-XX:CICompilerCount**, set these to limit CPU consumption.

Figure 5 depicts the HotSpot JVMs on Docker Container and vSphere.

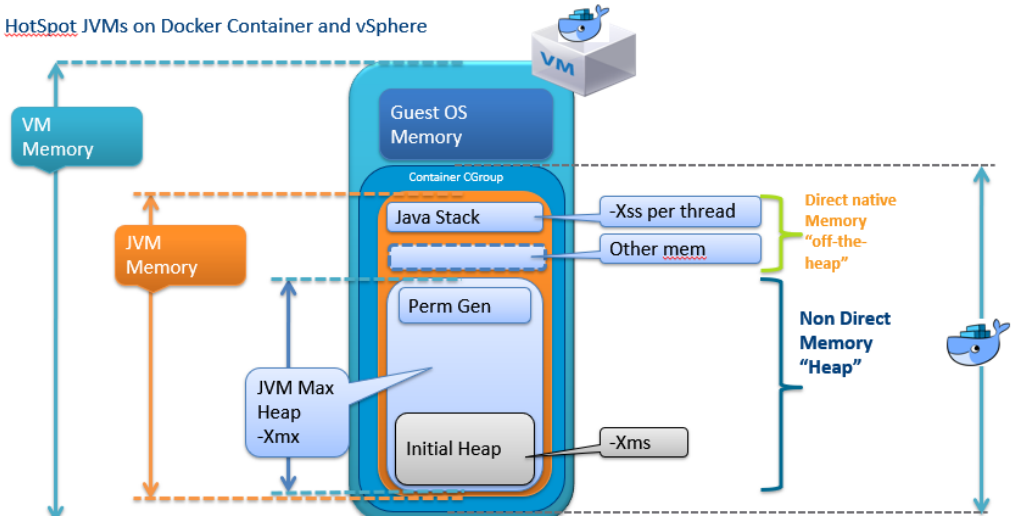

Figure 5. HotSpot JVMs on Docker Container and vSphere

For further information containerized environments, please refer to the *Best Practices for Enterprise Kubernetes on the VMware SDDC*.

Category 1 JVM Sizing Example

Figure 6 depicts the sizing considerations for Category 1 JVMs. Each memory area should be sized appropriately to have optimal performance.

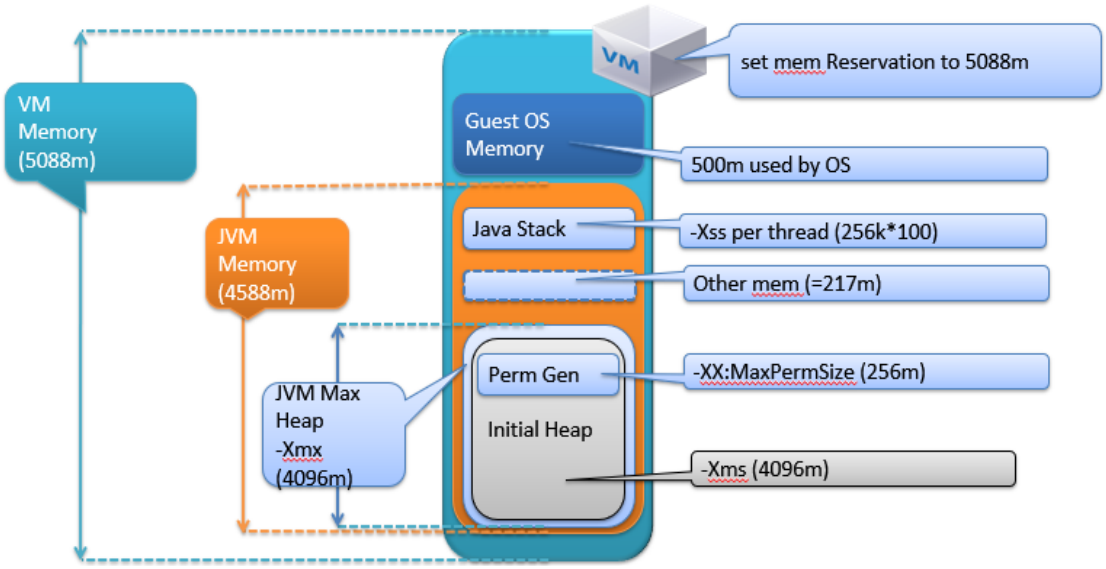

Figure 6. Category 1 JVM sizing example

Figure 6 presents the following information:

The Java Stack, -Xss value. Default values may vary between JVMs but mostly it is 512K, and typically it is being tuned down 128-192K. Each allocated Java thread will use up "-Xss" worth of memory that is not part of the Java Heap, i.e. it does not get deducted from the available –Xmx value; instead it is allocated natively and therefore the memory is provided by the OS. If there is an N number of concurrent threads, **Nthreads * -Xss** is the memory burden on the Guest OS.

NOTE: NIO Direct buffers are not mentioned here; they should rely on load testing to understand acceptable utilization rates.

1 VM and 1 JVM (simplification)

- The JVM memory regions are made of the JVM Heap which is governed/bounded by –Xmx (Max Heap) and –Xms (Initial Heap)
- Additionally, there is the **Perm Gen**, this is where Class level information is kept, typical this is a small area, but it can be 256m-512m at times.

*NOTE: For Category 1 workloads with web/large number of temporary object creations, setting **-Xmn** for 30 % or more is favorable. This is not something that is unique to JVM on vSphere, but it has to do with how Java applications behave.*

Category 2 JVM Sizing Example

Let's review an example Category 2 JVM sizing depicted by Figure 7.

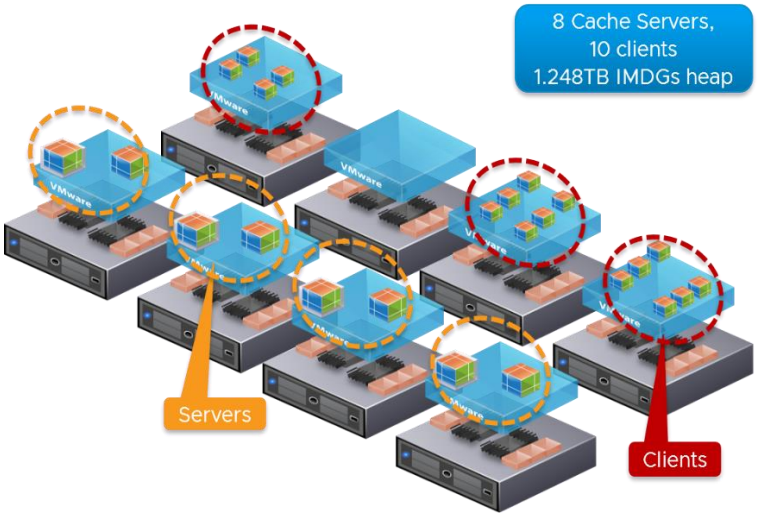

Figure 7. Category 2 JVM Sizing Example: Separate Clients from Cache Servers 10 clients, 8 cache servers

From Figure 7, review the following information:

- ESXi hosts' configuration: Gen9 Intel 2-socket servers of 12 cores per socket, 512GB of RAM
- Each Cache Server VM has 192 GB of RAM of memory reservation:
  - (512GB x 0.85)/2 = 217.6 GB which means 192GB VM can easily be supported. 217GB could have been chosen, however 192GB is an ample amount of memory for this configuration. To calculate the cache node's JVM heap size, use the following: 192GB x 0.81 = 156GB (as the heap size used).
- Each VM runs a GemFire JVM with 156 GB of heap space
- Each VM to have 12 vCPUs configured with all cores allocated on 1 socket

Here are the recommendations for the cluster depicted on Figure 7:

- Enable High Latency Sensitive on VMs
- 2 Client VMs of 4vCPU each, total 10 clients

Here is how to apply the JVM config:

-J-Xms156g -J–Xmx156g -J-Xmn46g -J-XX:+UseConcMarkSweepGC J-Xss2048k -J-XX:+UseParNewGC -J–XX:CMSInitiatingOccupancyFraction=75

-J–XX:+UseCMSInitiatingOccupancyOnly -J-XX:+ScavengeBeforeFullGC

-J-XX:TargetSurvivorRatio=80 -J-XX:SurvivorRatio=8 -J-XX:+DisableExplicitGC

-J-XX:MaxTenuringThreshold=15 -J-XX:ParallelGCThreads=6 -J-XX:+AlwaysPreTouch

-J-XX:+OptimizeStringConcat -J-XX:+UseStringCache

In general, the formula to calculate the VM memory for in-memory cache is the following:

VM Memory for in-memory cache = Guest OS memory + JVM Memory for cache node

- Set –Xms156g, also set –Xmx156g.
- The other segment of NumberOfConcurrentThreads*(-Xss) depends largely on NumberOfConcurrentThreads the JVM will process, and the –Xss value that's been chosen.
- -Xss is OS and JVM dependent; if the stack is not sized correctly, it results in a StackOverflow error. In Catgeory-2 we find performance benefits by using a large stack size of 1MB to 4MB.

NUMA Local Memory

It is important to realize that the overhead memory change with various-sized VMs (vSphere 6.7). Table 1 presents overhead memory values for various VM sizes.

Table 1. Overhead memory values in megabytes for various VM sizes.

Memory (MB)	1 VCPU	2 VCPUs	4 VCPUs	8 VCPUs
256	20.29	24.28	32.23	48.16
1024	25.90	29.91	37.86	53.82
4096	48.64	52.72	60.67	76.78
16384	139.62	143.98	151.93	168.60

Understanding NUMA boundaries is critical to sizing VM and JVMs. Remember, NUMA sizing is essential in sizing Category 2 JVMs and the "golden" rule is 1 JVM per VM per NUMA node.

Let's examine the sizing of JVMs for Category 2 JVMs such as in-memory DBs. It advisable to use the following PowerCLI script to retrieve the number of NUMA nodes on ESXi host:

PS C:\> $view = Get-VMHost -Name *<esxi host name>* | Get-View

PS C:\> $view.hardware.numainfo.NumNodes

The above script will return a number that indicates the number of NUMA nodes on the ESXi host.

Consider the following formula for NUMA Local Memory with Overhead Adjustment vSphere 5.5-6.7:

NUMA Local Memory Optimized VM Size =

*[Total RAM on Host – {(Total RAM on Host * nVMs * 0.01) + 4GB}] /Number of NUMA nodes*

where:

Total RAM on Host – Physical Ram on vSphere host

nVMs – Number of VMs on a vSphere host

Overhead 0.01 - 1%

vSphere RAM overhead - 4GB

Number of NUMA nodes per host – Number of NUMA nodes on vSphere host.

To examine NUMA calculation further, Figure 8 presents an example of NUMA sizing.

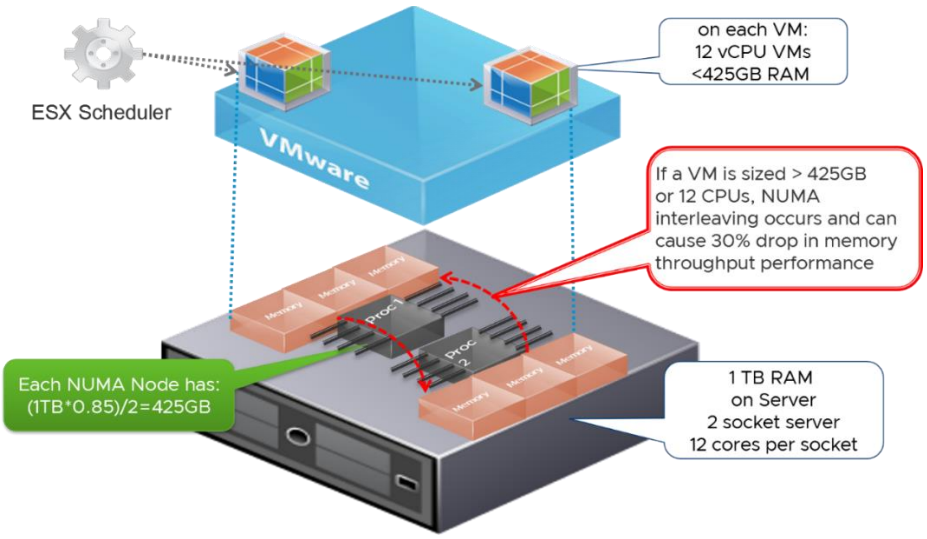

Figure 8. NUMA Local Memory Sizing

From Figure 8 example, total RAM on the host is 1000 GB with an Intel 2-socket architecture and 12 cores per socket. By using the formula for *NUMA Local Memory Optimized VM Size*, the following can be calculated:

*Intel NUMA Local Memory Optimized VM Size = (0.85 * Total RAM on Host) /Number of Sockets*

Using the values from Figure 8, calculation of the NUMA is for the Figure 8 example is as follows:

*NUMA size = (1000GB * 0.85)/2 = 425 GB*

*NOTE: If VM is sized greater than 425 GB or 12 CPUs, then NUMA interleaving occurs and can cause **30% drop in memory throughput performance.***

Virtual Machine vCPU Best Practices

Best Practice 1 (BP1): VM Sizing and VM-to JVM ratio through a performance load test

Establish a workload profile and conduct a load test to measure how many JVMs can be stacked on a particularly-sized VM. In this test establish a best-case scenario of how many concurrent transactions can be pushed through a configuration before it can be safely deemed as a good candidate for scaling horizontally in an application cluster. When in doubt, either due to time pressure or lack of information about the application, assume 1-JVM to 2-vCPU ratio.

BP2: VM vCPU CPU over-commit

For performance-critical enterprise Java applications VMs in production, try to make sure that the total number of vCPUs assigned to all of the virtual machines does not cause greater than 80% CPU utilization on the ESXI Host. If hyperthreading is enabled, assume Total vCPUs= pCPUs+25%. Approach this suggestion with caution – not all applications benefit from hyperthreading, but hyperthreading should always be enabled and it should not be figured into the overcommitment calculations.

BP3: VM vCPU Do not oversubscribe to CPU cycles that aren't needed

The optimal sizing of VMs in terms of vCPU is workload dependent. A good starting point is to size the VMs as large as the underlying cores within a single NUMA node/socket. If a performance load test determines that 2 vCPUs are adequate up to 70% CPU utilization, but instead, 4 vCPU are allocated to a VM, there is a possibility of having 2 vCPUs being idle, which is not an optimal configuration. If the exact workload is unknown, size the virtual machine with a smaller number of vCPUs initially and then increase the number later, if necessary.

BP4: Understanding How to Build a Scalability Building Block

Whether using Windows or Linux as a guest OS, refer to the technical specifications of the various vendors for memory requirements. It is common to see the guest OS allocated about 1GB in addition to the JVM memory size. However, each installation may have additional processes running; for example, monitoring agents - accommodate their memory requirements as well. The *–Xmx* value is the value found during load testing for an application on physical servers. This value does not need to change when moving to a virtualized environment. Load testing an application once deployed on vSphere, will help confirm the best -Xmx value. If memory is over-committed, ensure that no swapping is taking place.

Do not disable the Balloon driver.

BP5: Set memory reservation for VM memory needs

JVMs running on VMs have an active heap space requirement that must always be present in physical memory. Using the VMware vSphere Client set the reservation equal to the needed VM memory.

Reservation Memory = VM Memory = guest OS Memory + JVM Memory

Set this reservation to the active memory being used by the VM for a more efficient use of the amount of memory available. Or, a simpler approach is to set the reservation equal to the total configured memory of the VM.

BP6: it is advisable to use large memory pages (mostly Category 2)

Some of testing showed gains with large pages; however, this is workload dependent and mostly large pages aren't used to achieve the desired performance. If all of the best practices are exhausted and the usage large pages is still desired, establish a good benchmark and test thoroughly. If a performance benefit is found, then large page memory may be helpful. Large memory pages help performance by optimizing the use of the Translation Look-aside Buffer (TLB), where virtual to physical address translations are performed. The operating system and the JVM must be informed that large memory pages are in use just like in physical systems. Refer to the vendor's JVM large page configuration setting since large pages are turned at the JVM level.

Set Memory Reservations

Set a memory reservation value in the vSphere Client to the size of memory for the virtual machine. This VM will always get the reserved memory on any ESXi host on which it runs. To set the memory reservation select the VM, right-click and select **Edit Settings** > **Resources** tab.

Vertical Scalability Best Practices

If an enterprise Java application deployed on vSphere is experiencing heavy CPU utilization and it has been determined that an increase in the vCPU count may remediate the problem, use vSphere Hot Add to add an additional vCPU.

BP7: Hot Add CPU/Memory

VMs with a guest OS that support Hot Add CPU and Hot Add memory can take advantage of the ability to change the VM configuration at runtime without any interruptions to VM's operations. This is particularly useful when trying to increase the ability of a VM to handle more traffic. Plan ahead to enable this feature- some VMs must be turned off to have the Hot Add feature enabled. However, once enabled, Hot Add CPU and Hot Add memory at runtime works without VM shutdown. This feature must be supported by the guest OS.

NOTE: CPU Hot Add will disable the vNUMA presentation to a guest OS. In other words, if vNUMA needs to be exposed to a VM, disable the CPU Hot Add feature.

When Java heap space needs to grow, it typically causes an increase in vCPU count to get the best GC cycle performance. Remember, there are many other aspects to GC tuning and one should refer to a vendors' JVM documentation for GC tuning purposes.

Horizontal Scalability, Clusters, and Pools Best Practices

Enterprise Java applications deployed on VMware vSphere can benefit from using vSphere features for horizontal scalability such as ESXi host clusters, resource pools, host affinity and DRS.

BP8: Use ESXi host cluster

To enhance availability and scalability use ESXI host clusters. When creating cluster enable VMware HA and VMware DRS.

- VMware HA – Detects failures and provides rapid recovery for the VM running in a cluster. Core functionality includes a host, VM, and application monitoring to minimize downtime.
- VMware DRS – Enables vCenter Server to manage hosts as an aggregate pool of resources. Cluster resources can be divided into smaller pools for users, groups, and VMs. DRS enables vCenter to manage the assignment of VMs-to-hosts automatically, suggesting placement when VMs are powered on, and migrating running VMs to balance load and enforce allocation policies.

BP9: Use resource pools

Multiple resource pools can be used within a cluster to manage compute resource consumption by either reserving the needed memory for the VMs within a resource pool or by limiting/restricting it to a certain level. This feature also helps to meet quality of service requirements. For example, create a Tier-2 resource pool for the less critical applications and Tier-1 resource pool for business-critical applications.

BP10: Affinity Rules

In addition to existing VM affinity rules, the VM Host affinity rule was introduced in vSphere 4.1. The VM-Host affinity rule provides the ability to place VMs on a subset of hosts in a cluster. This is very useful in satisfying ISV-licensing requirements.

Rules can be for VMs to run on ESXi hosts in different blades for higher availability (VM anti-affinity). Conversely, limit the ESXi host to one blade in case network traffic between the VMs needs to be optimized by keeping them in one chassis location.

BP11: Use vSphere aware load balancers

vSphere makes it easy to add resources, such as hosts and VMs at runtime. It is possible to provision these ahead of time. However, it is simpler to use a load balancer that is able to integrate with vSphere APIs to detect the newly added VMs and add them to its application-load balancing pools without the need for network changes or any downtime.

Inter-tier Configuration Best Practices

As discussed previously, there are five critical technology tiers that sit on top of vSphere. These tiers are the Load Balancer tier, the Web Server tier, the Java Application Server tier, and the DB Server tier. The configurations for compute resources at each tier must translate to an equitable configuration at the next tier. For example, if the Web Server tier is configured to handle 100 HTTP requests per second, then of those requests you must determine how many Java application server threads are needed; and, how many DB connections are needed in the JDBC Pool configuration.

BP12: Establish appropriate thread ratios that prevents bottlenecks (HTTP threads:Java threads:DB connections)

This is the ratio of HTTP threads to Java threads to DB connections. Establish initial setup by assuming that each layer requires a 1:1:1 ratio of **HTTP threads:Java Threads:DB-connections**, and then base it on the response time and throughput numbers, adjusting each of these properties accordingly until SLA objectives are met. For example, if there are 100 HTTP requests submitted to the Web Server initially, assume that all of these will have an interaction with Java threads and DB connections. Benchmarking may show results that not all HTTP threads are submitted to the Java application server; and therefore not all Java application server threads each require a DB connection. It is possible that of ratio 100 requests translates to **100 HTTP threads:25 Java threads:10 DB connections** which depends on the nature of the enterprise Java application behavior. Benchmarking helps establishing this ratio.

BP13: Load Balancer (LB) algorithm choice and VM symmetry

Consider the available algorithms of the available load balancer. While using the Scale-Out approach, ensure that all VMs are receiving an equal share of traffic. Some LB algorithms include: *Round Robin, Weighted Round Robin, Least Connections,* and *Least Response Time*. Try using Least Connections and then adjust as suitable with load test iterations.

Keep VMs symmetrical in terms of the size of the compute resource. For example, for an application, use 2 vCPU VMs as a repeatable and horizontally scalable building block that helps with a load balancing algorithm versus a pool of non-symmetrical VMs. Mixing 2 vCPU VMs with 4 vCPU VMs in one load balancer-facing pool is non-symmetrical and this load balancer has no notion of weighing these server activities, unless it is configured at the load balancer level (which make require a lot of time).

High Level vSphere Best Practices

It is important to follow the best practices in vSphere configurations. See *Performance Best Practices for VMware vSphere® 6.7.* This section of the document summarizes some of the key networking, storage and hardware-related best practices.

BP14: vSphere Networking

In addition to *Performance Best Practices for VMware vSphere® 6.7,* refer to the *VMware Virtual Networking Best Practices*.

BP15: vSphere Storage

VMware recommends a minimum of 4 paths from an ESXI host to a storage array, which implies that the host requires at least two HBAs. For detailed description of VMware Storage Best Practices refer to the *VMware vSphere Storage Best Practices.*

BP16: ESXi Host Hardware

Refer to the *vSphere Resource Management Guide 6.7*.

NUMA Considerations – IBM (X-Architecture), AMD (Opteron-based), and Intel (Xeon-based) non-uniform memory access (NUMA) systems are supported by ESXi. On AMD Opteron-based systems, BIOS settings for node interleaving determine whether the system behaves like a NUMA system or like a uniform memory accessing (UMA) system.

By default, ESXi NUMA scheduling and related optimizations are enabled only on systems with a total of at least 4 CPU cores and with at least 2 CPU cores per NUMA node. Virtual machines with a number of vCPUs equal to or less than the number of cores in each NUMA node are managed by the NUMA scheduler and have the best performance.

Hardware BIOS – verify the BIOS is set to enable all populated sockets, and enable all cores in each socket. Enable Turbo Mode if a host's processor supports it. Make sure hyper-threading is enabled in the BIOS. Disable any power-saving mode in the BIOS.

Enterprise Java Applications on vSphere FAQs

With UNIX-based hardware I have very large machines running all of my Java applications. What should be my migration sizing strategy and what are the VMware vSphere maximums that I need to be aware of?

NOTE: One of the most important steps is to conduct a load test to help determine the ideal individual VM size and how many JVMs you can stack up (vertical scalability). Based on this repeatable building block VM, scale-out to determine what is best for the application traffic profile. Know the VMware vSphere maximums. See VMware Configuration Maximums.

The following Table 2 provides a summary of maximums for vSphere 6.7 per VM, host, and vCenter.

Table 2. vSphere 6.7 Configuration Maximums

vSphere Configuration	Maximum (vSphere 6.7)
Per VM	256 vCPUs 6TB vRAM 62TB of storage minus 512 bytes per VMDK
Per Host	4096 vCPUs 1024 VMs 16TB RAM 16 NUMA nodes
Per vCenter	2000 Hosts 25000 Powered on VMs 35000 registered VMs 15 Linked vCenter Servers 5000 Hosts in linked vCenter servers 50000 Powered on VMs in linked vCenter Servers 70000 Registered VMs in linked vCenter Servers 180 Concurrent Clients Sessions (Web/HTML)

What are decisions must be made due to virtualization?

Define the fault domains for the applications considering the following facts - multiple VMs can reside on a single vSphere ESXi host. Consider using DRSs affinity/anti-affinity rules.

Determine the size of the repeatable building block VM. This is established by benchmarking, along with total scale-out factor. Determine how many concurrent users each single vCPU configuration of the application can handle and extrapolate that to the production traffic to determine overall compute resource requirement. Have a symmetrical building block - for example, every VM having the same number of vCPUs, helps keep load distribution from the load balancer. Essentially, the benchmarking test help to determine how large a single VM is (vertical scalability) and how many of these VMs are needed (horizontal scalability).

Pay special attention to scale-out factor and see up to what point it is linear within the application running on top of VMware. Enterprise Java applications are multi-tiered and bottlenecks can appear at any point along the scaling out and can quickly cause non-linear results. The assumption of linear scalability may not always be true; it is essential to load test a pre-production replica an environment to accurately size traffic.

I have conducted extensive GC sizing and tuning for our current enterprise Java application running on physical. Do I have to adjust anything related to sizing when moving this Java application to a virtualized environment?

No. All tuning that is performed for a Java application on physical is transferrable to your virtual environment. However, because virtualization projects are typically about driving a high consolidation ratio, it is advisable to conduct adequate load testing in order to establish the ideal compute resource configuration for individual VMs, number of JVMs within a VM and overall number of VMs on the ESXI host. Additionally, because this type of migration involves an OS/platform change as well as a JVM vendor change.

What is the correct number of JVMs per virtual machine?

There is not one definite answer. This is largely dependent on the nature of the application. The benchmarking conducted can determine the limit of the number of JVMs you can stack up on a single VM.

More JVMs that are put on a single VM, more JVM overhead/cost of initializing a JVM is incurred. Instead of stacking up multiple JVMs within a VM, increase the JVM size vertically by adding more threads and larger heap size. This can be achieved if the JVM is within an application server such as Tomcat. Additionally, instead of increasing the number of JVMs, increase the number of concurrent threads available and resources that a single Tomcat JVM services for the N-number of applications deployed and their concurrent requests per second. The limitation of how many applications can be stacked up within a single application server instance/JVM is dependent on how large JVM heap size can be set. The trade-off of a very large JVM heap size beyond 4GB needs to be tested for performance and a GC cycle impact. This concern is not specific to virtualization as it equally applies to physical server setup.

References

Benjamin E. (2014). Virtualizing and Tuning Large-Scale Java Platforms. Pearson plc, Published as VMware Press.

Cloud Foundry (2016). Hope Versus Reality: Containers In 2016. Global Perception Study conducted by ClearPath. Retrieved on December 20, 2019 from *https://www.cloudfoundry.org/wp-content/uploads/2017/01/Cloud-Foundry-2016-Container-Report.pdf*

Javin P. (2017). 10 points about Java Heap Space or Java Heap Memory, May 2017. Retrieved from *https://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html*

Joines S., Willenborg R., Hygh K., (2002). Performance Analysis for Java Web Sites. Addison-Wesley. Copyright © 2003 Pearson Education, Inc.

Markova T. (2016). Software Stacks Market Share: First Quarter of 2016, April 14, 2016. Retrieved on September 3, 2019 from *https://jelastic.com/blog/software-stacks-market-share-first-quarter-of-2016/*

ORACLE (n.d.). Tuning Java Virtual Machines (JVMs) Retrieved on November 12, 2019 from *https://docs.oracle.com/cd/E21764_01/web.1111/e13814/jvm_tuning.htm#PERFM150*

PKS on the VMware SDDC (2019). Best Practices for VMware Enterprise PKS on the VMware SDDC. Retrieved on October 15, 2019 from *https://blogs.vmware.com/apps/files/2019/11/Best-Practices-for-VMware-Enterprise-PKS-on-the-VMware-SDDC-Final01.pdf*

Acknowledgements

Emad Benjamin has spent the past 25 years in various software engineering positions involving software development of application platforms and distributed systems for various industries such as finance, health, IT, and heavy industry – in various international locations. Emad is currently the Sr. Director and Chief Technologist of Application Platforms with Office of the CTO at VMware, focusing on building hybrid cloud distributed runtimes that are application aware.

Timur Mirzoev, Ph.D., is a Senior Technical VMware instructor and has several research publications. Timur has served as an SME for several projects worldwide. Additionally, Timur has trained thousands of people and has been recognized as an international speaker on many occasions.

This document would not have been possible without the help from **Dan Linsley**, Application Platform Architect, VMware Office of the CTO and **Brett Guarino**, VCDX, a Senior Technical Instructor at VMware.

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com.