Chapter 1

Introduction to Large-Scale Java Platforms

This chapter defines three categories of large-scale Java platforms:

- **Category 1:** Large number of Java Virtual Machines (JVMs) (100s–1000s of JVMs)
- **Category 2:** Smaller number of JVMs with large heap sizes
- **Category 3:** A combination of category 1 consuming data from category 2

In addition, the chapter discusses various trends and outlines technical considerations to help you understand the range of technical issues associated with designing large-scale Java platforms.

Large-Scale Java Platform Categories

Based on field interactions with customers, large-scale Java platforms typically fall into three main categories, as follows:

- **Category 1:** This category is distinguished by its large number of Java Virtual Machines (JVMs). In this category, hundreds to thousands of JVMs are deployed on the Java platform, and these are typically JVMs that function within a system that might be servicing millions of users. I have seen some customers with as many as 15,000 JVMs. Whenever you are dealing with thousands of JVM instances, you must consider the manageability cost and whether opportunities exist to consolidate the JVM instances.

- **Category 2:** This category is distinguished by a smaller number of JVMs (usually 1 to 20) but with large heap size (8GB to 256GB or higher). These JVMs usually have in-memory databases deployed on them. In this category, garbage collection (GC) tuning becomes critical, as discussed in later chapters.
- **Category 3**: The third category is a combination of the first two categories, where perhaps thousands of JVMs run enterprise applications that are consuming data from category 2 types of large JVMs in the back end.

With regard to virtualizing and tuning large-scale Java platforms, four key requirement trends hold true across these three categories:

- Compute-resource consolidation
- JVM consolidation
- Elasticity and flexibility
- Performance

Let's look at each one of these trends in more detail.

Large-Scale Java Platform Trends and Requirements

Compute resource consolidation, JVM instance consolidation, elasticity and flexibility, and performance are some of the major trends that exist within large-scale Java platform migration projects. The following subsections examine each of these in more detail.

Compute-Resource Consolidation

Many VMware customers find that their middleware deployments have proliferated and are becoming an administrative challenge with increasing costs. Customers, therefore, are looking to virtualization as a way of reducing the number of server instances. At the same time, customers are taking the consolidation opportunity to rationalize the number of middleware components needed to service a particular load. Middleware components most commonly run within a JVM with an observed scale of hundreds to thousands of JVM instances and provide many opportunities for JVM instance consolidation. Hence, middleware virtualization provides an opportunity to consolidate twice—once to consolidate server instances and then to consolidate JVM instances. This trend is widespread; after all, every IT shop on the planet is considering the cost savings of consolidation.

One customer in the hospitality sector went through the process of consolidating their server footprint and at the same time consolidated many smaller JVMs with a heap smaller

than 1GB. They consolidated many of these smaller 1GB JVMs into two categories: those that were 4GB and others that were 6GB. They performed the consolidation in such a manner that the net total amount of RAM available to the application was equal to the original amount of RAM, but with fewer JVM instances. They did all of this while improving performance and maintaining good service level agreements (SLAs). They also reduced the cost of administration considerably by reducing the number of JVM instances they had to originally manage; this refined environment helped them easily maintain SLAs.

Another customer, in the insurance industry, achieved the same result, but was also able to overcommit CPU in development and QA environments to save on third-party software license costs.

JVM Instance Consolidation

Sometimes we come across customers that have a legitimate business requirement to maintain one JVM for an application and/or one JVM per a line of business. In these cases, you cannot really consolidate the JVM instances because doing so would cause intermixing of the lifecycle of one application from one line of business with another. However, although such customers do not benefit from eliminating additional JVM instances through JVM consolidation, they do benefit from more fully utilizing the available compute resources on the server hardware, resources that otherwise would have been underutilized in a nonvirtualized environment

Elasticity and Flexibility

It is increasingly common to find applications with seasonal demands. For example, many of our customers run various marketing campaigns that drive seasonal traffic toward their application. With VMware, you can handle this kind of traffic burst by automatically provisioning new virtual machines (VMs) and middleware components when needed; you can then automatically tear down these VMs when the load subsides.

The ability to change updating/patching hardware without causing outage is paramount for middleware that supports the cloud era scale and uptime. VMware VMotion enables you to move VMs around without needing to stop applications or the VM. This flexibility alone makes virtualization of middleware worthwhile when managing large-scale middleware deployments. One customer in the financial space, handling millions of transactions per day, used VMotion quite often, without any downtime, to schedule their hardware upgrades; a process that otherwise would be costly to their business because of the required scheduled downtime.

Performance

Customers often report improved middleware platform performance when virtualizing. Performance improvements are partly due to the updated hardware that customers will typically refresh during a virtualization project. Some performance improvement occurs, too, due to the robust VMware hypervisor. The VMware hypervisor has improved considerably in the past few years, and Chapter 5, "Performance Studies," discusses a few performance studies done to showcase some of the heavy workloads that were tested in a virtualized environment.

Large-Scale Java Platform Technical Considerations

When designing large-scale Java platforms, many technical considerations come into play. For example, a good understanding of Java garbage collection (GC) and of JVM architecture, hardware, and hypervisor architectures is essential to building good large-scale Java platforms. At a high level, GC, Non-Uniform Memory Architecture (NUMA), and theoretical versus practical memory limits are discussed. Later chapters provide a more detailed description, but it is imperative to start at a high-level understanding of the issues surrounding large-scale Java platform designs.

Theoretical and Practical Limits of Java Platforms

Figure 1-1 depicts the theoretical and practical sizing limits of Java workloads, critical limits to remember when sizing JVM workloads.

- It is important to highlight that the JVM theoretical limit is 16 exabytes; however, no practical system can provide this amount of memory. So, we capture this as the first theoretical limit.
- The second limit is the amount of memory a guest operating system can support; in most practical cases, this is several terabytes (TB) and depends on the operating system being used.
- The third limit is the ESXi5 1TB RAM per VM, which is ample for any workload that we have encountered with our customers.
- The fourth limit (really the first practical limit) is the amount of RAM that is cost-effective on typical ESX servers. We find that, on average, vSphere hosts have 128GB to 144GB, and at the top end 196GB to 256GB. Certainly from a feasibility standpoint, the hard limit is probably around 256GB. There are, of course, larger RAM-based vSphere hosts, such as 384GB to 1TB; however, these are probably more suited for category 2 types of in-memory database workloads and more likely

suited for traditional relational database management systems (RDBMS) that would utilize such vast compute resources. The primary reason these systems need such large vSphere hosts is because most (with some minor exceptions, such as Oracle RAC) traditional RDBMS do not scale out and mainly scale up. In the case of category 1 and category 2, a scale-out approach is available and so the potential selection of a more cost-effective vSphere host configuration is afforded. In category 1 types of Java workloads, you should consider vSphere hosts with a more reasonable RAM range of less than 128GB.

- The fifth limit is the total amount of RAM across the server and how this is divided into a number of NUMA nodes, where each processor socket will have one NUMA node worth of NUMA-local memory. The NUMA-local memory can be calculated as the total amount of RAM within the server divided by the number of processor sockets. We know that for optimal performance you should always size a VM within the NUMA node memory boundaries; no doubt, ESX has many NUMA optimizations that come into play, but it is always best to stay NUMA local.

  If the ESX host, for example, has 256GB of RAM across two processor sockets (that is, it has two NUMA nodes with 128GB (256GB/2) of RAM across each NUMA node), this implies that when you are sizing a VM it should not exceed the 128GB limit for it to be NUMA local.

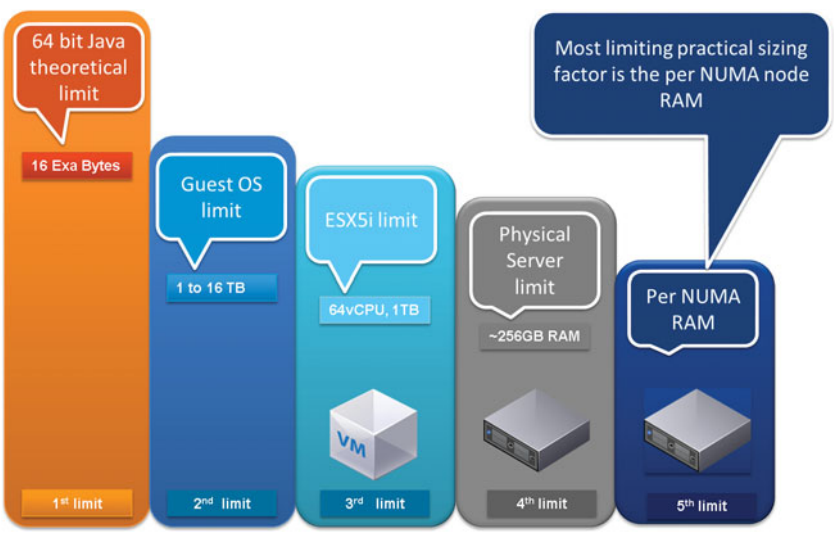

Figure 1-1 Theoretical and Practical Limits of Java Platforms

The limits outlined in Figure 1-1 and the list will help drive your design and sizing decision as to how practical and feasible it is to size large JVMs. However, other considerations come with sizing very large JVMs, such as GC tuning complexity and knowledge needed to maintain large JVMs. In fact, most JVMs within our customer base are in the vicinity of 4GB of RAM for the typical enterprise web application, or what has been referred to in this book as category 1 workloads. However, larger JVMs exist, and we have customers that run large-scale monitoring systems and large distributed data platforms (in-memory databases) on JVMs ranging from 4GB to 128GB. This is also true for in-memory databases such as vFabric GemFire and SQLFire, where individual JVM members within a cluster can be as big as 128GB and total cluster size can be 1 to 3TB. With such large JVMs comes the need to have a better knowledge of GC tuning. At VMware, we have helped many of our customers with their GC tuning activities over the years, even though GC tuning on physical is no different from on virtual. The reason being is that we have uniquely integrated the vFabric Java and vSphere expertise into one spectrum, which has helped our customers optimally run many Java workloads on vSphere. When faced with the decision of whether to vertically scale the size of the JVM and VM, always first consider a horizontal scale-out approach; we have found that our customers get better scalability with a horizontally scaled-out platform. If horizontal scalability is not feasible, consider increasing the size of the JVM memory and hence VM memory. When opting to increase the size of the JVM by increasing the heap space/memory, the next point of consideration is GC tuning and the in-house knowledge you have to handle large JVMs.

NOTE

With regard to the third limit, as of this writing, ESXi 5.1 is the GA released official version; however, by the time this book is published, some of these maximum vSphere limits might change. Double-check official VMware product documentation for the latest maximums. Note, as well, that at these VM limits no cost-effective hardware would need such a large number of vCPUs; however, it is still assuring for those who might need it.

As mentioned earlier in this chapter, in the enterprise today large-scale Java platforms fall into one of three categories. Figure 1-2 shows the various workload types and relative scale. A common trend is that as the size of the JVM increases so, too, does the required JVM GC tuning knowledge.

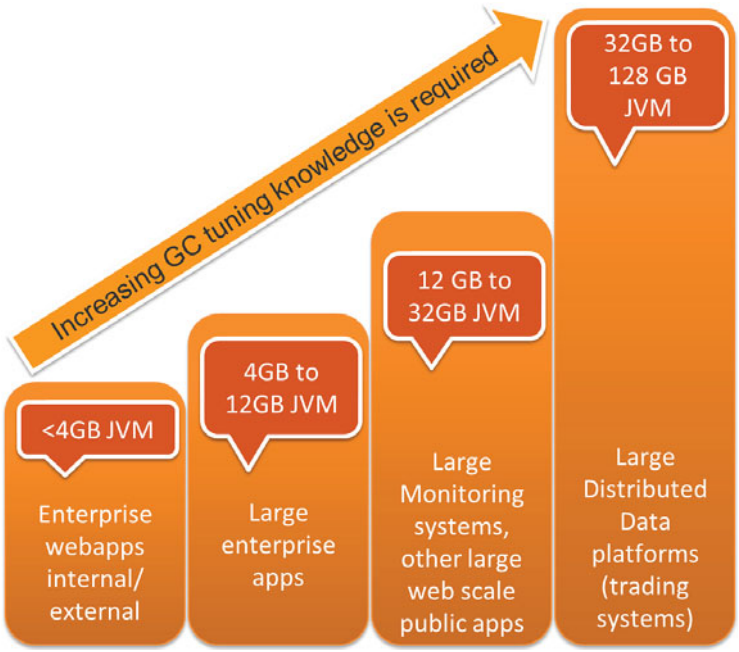

Figure 1-2 GC Tuning Knowledge Requirements Increase with Larger JVMs

It is important to keep the following in mind (from left to right in the figure):

- JVMs with a less than 4GB heap size are the most common among workloads today. The 4GB is a special case because it has the default advantage of using 32-bit address pointers within a 64-bit JVM space (and so has a very efficient memory footprint). These require some tuning, but not a substantial amount. This workload type falls into the realm of category 1 as defined earlier in this chapter. The default GC algorithm on server class machines is adequate. The only time you need to tune these is if the response time measurements do not suffice. In such cases, you want to follow the guidance on GC tuning in Chapter 3, "Tuning Large-Scale Java Platforms," and in Chapter 6, "Best Practices."
- The second workload case is still within category 1, but it is probably a serious user base internal to the organization. In this workload, we typically see heavily used (1,000 to 10,000 users) enterprise Java web applications. In these types of environments, GC tuning and slightly larger than 4GB JVMs are the norm. The DevOps team almost always has decent GC tuning knowledge and has configured the JVM away from the default GC throughput collector. Here we start to see the use of the concurrent mark and sweep (CMS) GC algorithms for these types of workloads to deliver decent response times to the user base. The CMS GC algorithm is offered

by the Oracle JVM (formerly Sun JVM). For further details and information about other GC algorithms within the Oracle JVM or IBM JVM, see Chapter 3 and Chapter 6.

- The third workload type could fall into category 2, but it is a unique case within category 2 because sometimes the larger JVMs are used because the application cannot scale out horizontally. Generic category 2 workloads are usually in-memory databases, as mentioned earlier in the chapter. In this category, a deep knowledge of JVM GC tuning is required. Your DevOps team must be able to articulate all the different GC collectors and select those most suited for improved throughput (throughput collectors) (in contrast to latency-sensitive workloads that need CMS GC to deliver better response times).
- The fourth workload type falls into both category 2 and 3. Here there could be a large distributed system, where the client enterprise Java applications are consuming data from the back-end data fabric where a handful or more of in-memory database JVM nodes are running. Tuning GC at expert level is required here.

Other than having to maintain a very large JVM, you must know the workload choices. After all, customers often scale the JVM vertically because they believe it is an easy deployment and that it is best to just leave the existing JVM process intact. Let's consider some JVM deployment and usage scenarios (perhaps something in your current environment or something you have encountered at some point):

- A customer has one JVM process deployed initially. As demand for more applications to be deployed increases, the customer does not horizontally scale out by creating a second JVM and VM. Instead, the customer takes a vertical scale-up approach. As a consequence, the existing JVM is forced to vertically scale and carry many different types of workloads with varied requirements.
- Some workloads, such as a job scheduler, require high throughput, whereas a public-facing web application requires fast response time. So, stacking these types of applications on top of each other, within one JVM, complicates the GC cycle tuning opportunity. When tuning GC for higher throughput, it is usually at the cost of decreased response time, and vice versa.
- You can achieve both higher throughput and better response time with GC tuning, but it certainly extends the GC tuning activity unnecessarily. When faced with this deployment choice, it is always best to split out the types of Java workloads into their own JVMs. One approach is to run the job scheduler type of workload in its own JVM and VM (and do the same for the web-based Java application).
- In Figure 1-3, JVM-1 is deployed on a VM that has mixed application workload types, which complicates GC tuning and scalability when attempting to scale up

this application mix in JVM-2. A better approach is to split the web application into JVM-3 and the job scheduler application into JVM-4 (that is, horizontally scaled out and with the flexibility to vertically scale if needed). If you compare the vertical scalability of JVM-3 and JVM-4 versus the vertical scalability of JVM-2 you will find JVM-3 and JVM-4 always scale better and are easier to tune.

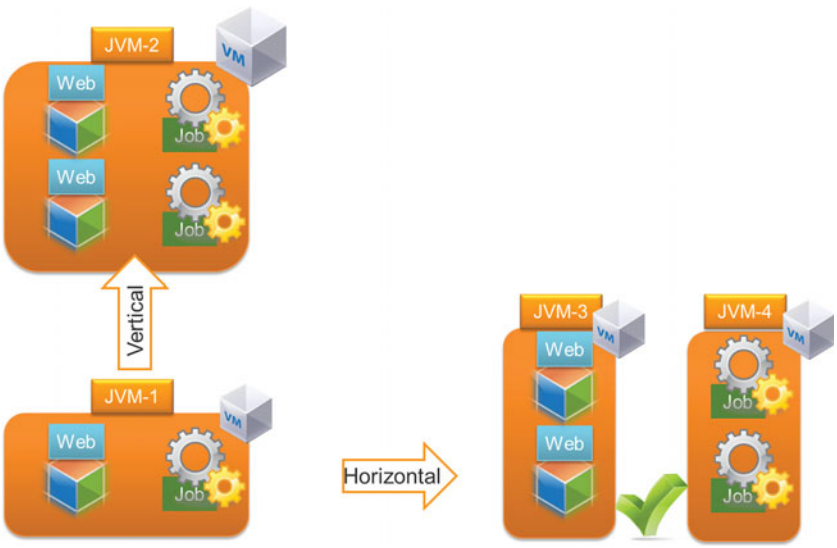

Figure 1-3 Avoiding Mixed Workload Types in the Same JVM

NUMA

Non-Uniform Memory Architecture (NUMA) is a computer memory design used in multiprocessors, where the memory access time depends on the memory location relative to a processor. Under NUMA, a processor can access its own local memory faster than nonlocal memory (that is, memory local to another processor or memory shared between processors).

Understanding NUMA boundaries is critical to sizing VM and JVMs. Ideally, the VM size should be confined to the NUMA boundaries. Figure 1-4 shows a vSphere host made of two sockets, and hence two NUMA nodes. The workload shown is that of two vFabric SQLFire VMs, each VM sized to fit within the NUMA node boundaries for memory and CPU. If a VM is sized to exceed the NUMA boundaries, it might possibly interleave with the other NUMA node to fulfill the request for additional memory that otherwise cannot be fulfilled by the local NUMA node. The figure depicts memory interleaving by the red arrows (dashed curved arrows show the interleaving), highlighting that this type of memory interleaving should be avoided because it may severely impact performance.

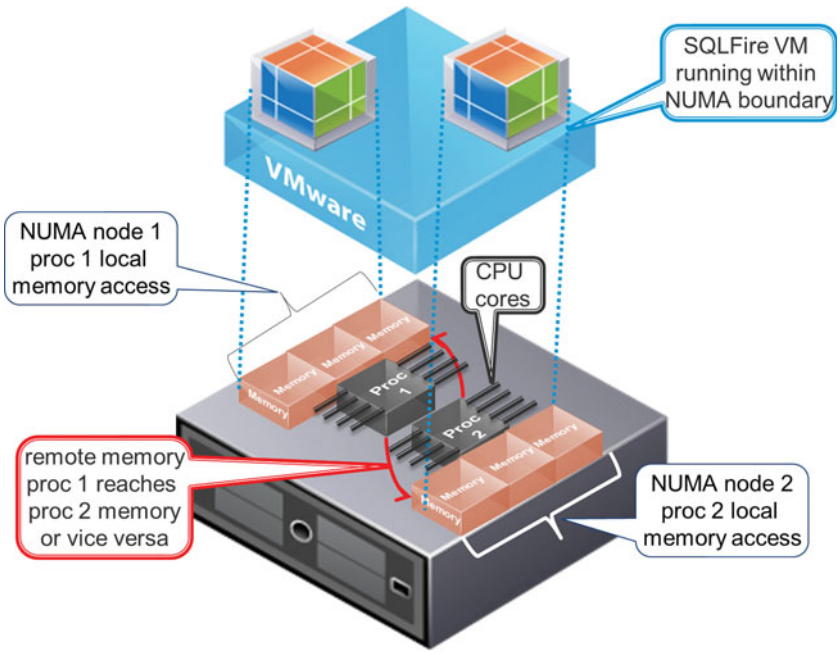

Figure 1-4 Two-Socket Eight-Core vSphere Host with Two NUMA Nodes and One VM on Each NUMA Node

To calculate the amount of RAM available in each NUMA node, apply the equation in Formula 1-1.

NUMA Local Memory = Total RAM on Server /Number of Sockets

Formula 1-1 Per-NUMA Node RAM Size (NUMA Local Memory)

For example, if a server has 128GB of RAM configured on it and has two sockets (as shown in Figure 1-4), this implies that the per-NUMA RAM is 128/2, which equals 64GB. This is not entirely true, however, because ESX overhead needs to be accounted for. So, a more accurate approximation results from the equation shown in Formula 1-2. The formula accounts for the ESXi memory overhead (1GB as a constant, regardless of the size of the server) and a 1% VM memory overhead as 1% of the available memory. The formula is a conservative approximation, and every VM and workload will vary slightly, but the approximation should be pretty close to the worst-case scenario.

```
NUMA Local Memory =
[Total RAM on Host- {(Total RAM on Host* nVMs * 0.01)+1GB}] /Number of Sockets
```

Formula 1-2 Per-NUMA Node RAM (NUMA Local Memory) with ESXi Overhead Adjustment

The following explains the different parts of the formula:

- **NUMA Local Memory:** The local NUMA memory for best memory throughput and locality, with VM and ESXi overhead already accounted for
- **Total RAM on Host:** The amount of physical RAM configured on the physical server
- **nVMs:** The number of VMs you plan to deploy on the vSphere host
- **1GB:** The overhead needed to run ESXi
- **Number of Sockets:** The number of sockets available on the physical server, 2 socket or 4 socket

NOTE

Formula 1-2 assumes the most pessimistic end of the overhead range, especially as you increase the number of VMs—clearly, as you add more VMs you will have more overhead. Despite a lower number of VMs, the approximation of Formula 1-2 is pretty fair and accurate. Also, this assumes a non-overcommitted memory situation. This formula is beneficial for sizing large VMs, which is when NUMA considerations are most pertinent. When sizing large VMs, typically you are trying to maintain fewer than a handful of configured VMs, so this overhead formula accurately applies. In fact, the most optimal configuration for larger VMs that have memory-bound workloads is one VM per NUMA node. If you try to apply this formula to a deployment that has more than six VMs configured, say 10 VMs, the formula can overestimate the amount of overhead needed. More accurately, you can use the 6% rule, which maintains that regardless of the number of VMs, always assume that 6% of memory overhead is ample, whether you have 10 VMs or 20.

If you don't have time to crunch through the formula and want to quickly start configuring, assume about 6% of overhead due to memory. There are many times when not all of this is being used. For example:

Example 1—Using 6% approximation approach: This would imply that if you have a server which has 128GB of physical RAM (two socket hosts, eight cores on each socket) and you choose the 6% overhead approach while configuring two VMs on the host, the total NUMA local memory would be => ((128 * 0.94) – 1) / 2 => 59.7GB per VM available for memory. Because there are two VMs, the total memory offered to the two VMs is approximately 59.7 * 2 => 119.32GB.

You also can apply the approach in Formula 1-2 as shown in Example 2 that follows:

Example 2—Using Formula 1-2 to calculate NUMA local available memory: Again, assuming a 128GB host with two sockets (eight cores on each socket) and two VMs to be configured on it, NUMA local memory = (128 – (128 * 2 * 0.01) - 1) / 2 => 124.44GB. Note that this is for two VMs. If you decide instead to configure 16 VMs of 1vCPU (1vCPU = 1 core), then the NUMA local memory per VM would be NUMA local memory = (128 – (128 * 16 * 0.01) - 1) / 2 => 53.26GB. This probably is overly conservative, and a more accurate representation would be around the 6% overhead calculation approach.

For best guidance, the best approximation of overhead is the 6% of total physical RAM (plus 1GB for ESXi) approach shown in Example 1.

In the preceding example, showing a calculation based on a server having 128GB of RAM, the true local memory would be ((128 * 0.99) – 1GB)/2 => 62.86GB, which is the maximum VM size that can be configured. In this case, you can safely configure two VMs of 62.68GB of RAM and eight vCPUs each, because each of the VMs would be deployed on one NUMA node. Alternatively, you can deploy four VMs if you want to deploy smaller VMs of 62.86GB / 2 => 31.43GB of RAM and four vCPUs each, and the NUMA scheduling algorithm would still localize the VMs to the local NUMA node.

NOTE

On hyperthreaded systems, VMs with a number of vCPUs greater than the number of physical cores in a NUMA node but lower than the number of logical processors (logical processors are usually shown as 2.x of physical cores, but more practically, logical processors are 1.25x of physical cores) in each physical NUMA node might benefit from using logical processors with local memory instead of full cores with remote memory. You can configure this behavior for a specific VM with the numa.vcpu.preferHT flag. For further details, see http://www.vmware.com/pdf/Perf_Best_Practices_vSphere5.1.pdf and the KB article kb.vmware.com/kb/2003582.

It is always advisable to start with vCPUs equal to the number of physical cores and then adjust vCPUs upward when needed, but less than approximately 1.25x of available physical cores.

To further elaborate on the ESXi NUMA scheduling algorithm, Figure 1-5 shows an example of two sockets and six cores on each socket of the server.

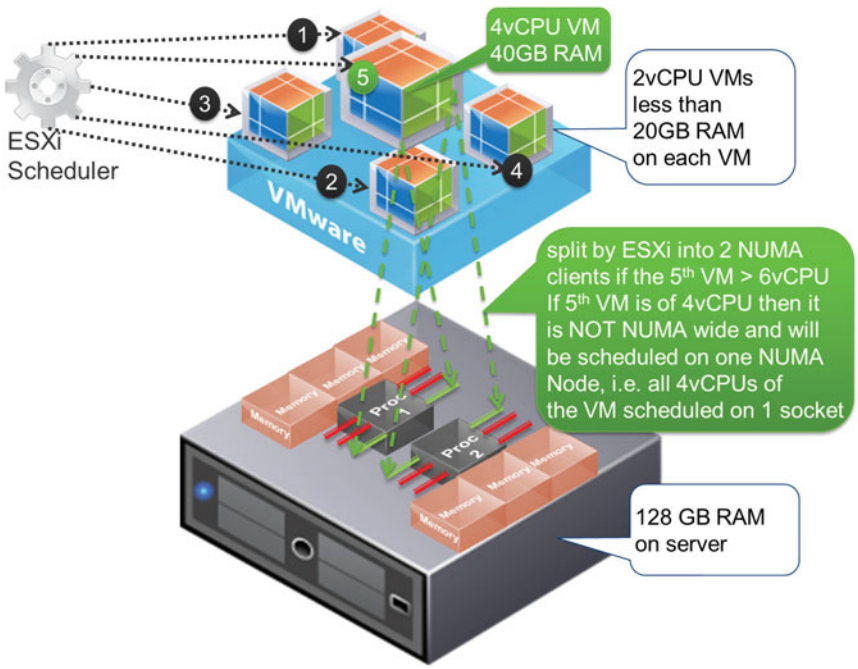

Figure 1-5 ESXi NUMA Scheduling on a Two-Socket Six-Core Server

In this figure, there are initially four VMs of two vCPUs and approximately 20GB RAM on each. The initial ESXi scheduling algorithm will follow a round-robin fashion. First, step 1 occurs (as shown by the black circle with the number 1), and then the next two vCPU VMs are scheduled on the next available empty NUMA node, and then so on (steps 3 and 4) for scheduling the third and fourth VMs. At the point where all four of the 2vCPU 20GB VMs have been scheduled, and as a result of this scheduling, the four VMs will occupy the four cores on each of the sockets, as shown by red pins in this figure (red pins are the pins the four 2 vCPU VMs were initially scheduled ESXi). Moments later, a fifth VM made of four vCPUs and 40GB RAM is deployed, and now ESXi attempts to schedule this VM across one NUMA node. This is because the VM is 4vCPU and is not considered a NUMA-wide VM, so all four of its vCPUs will be scheduled on one NUMA node, even though only two vCPUs are available. What will likely happen in terms of the NUMA balancing awareness algorithm is that the ESXi scheduler will eventually force one of the two vCPU VMs to migrate to the other NUMA node in favor of trying to fit the fifth 4vCPU VM into one NUMA node. The ESXi scheduler behaves like this because it uses a concept of NUMA client and schedules VMs per NUMA client, where the default size of the NUMA client is the size of the physical NUMA node. In this case, the default is 6, so any VM that is 6vCPU and less will be scheduled on one NUMA node because it fits into one NUMA client. If you want to change this behavior, you would have to force

the NUMA client calculation to something more granular. The NUMA client calculation is controlled by *numa.vcpu.maxPerClient*, which can be set as Advanced Host Attributes -> Advanced Virtual NUMA Attributes, and if you were to change this to 2, then effectively every socket in our example will have three NUMA clients, so each 2vCPU VM will be scheduled into one NUMA client, and the fifth 4vCPU VM will be scheduled across two NUMA clients, and potentially across two sockets if need be. You seldom need get to this level of tuning, but this example illustrates the power of the NUMA algorithm within vSphere, which far exceeds any nonvirtualized Java platforms.

In general, when a virtual machine is powered on, ESXi assigns it a home node as part of its initial placement algorithm. A virtual machine runs only on processors within its home node, and its newly allocated memory comes from the home node as well. Unless a virtual machine's home node changes, it uses only local memory, avoiding the performance penalties associated with remote memory accesses to other NUMA nodes. When a virtual machine is powered on, it is assigned an initial home node so that the overall CPU and memory load among NUMA nodes remains balanced. Because internode latencies in a large NUMA system can vary greatly, ESXi determines these internode latencies at boot time and uses the information when initially placing virtual machines that are wider than a single NUMA node. These wide virtual machines are placed on NUMA nodes that are close to each other for lowest memory access latencies. Initial placement-only approaches are usually sufficient for systems that run only a single workload, such as a benchmarking configuration that remains unchanged as long as the system is running. However, this approach cannot guarantee good performance and fairness for a datacenter-class system that supports changing workloads. Therefore, in addition to initial placement, ESXi 5.0 does dynamic migration of virtual CPUs and memory between NUMA nodes for improving CPU balance and increasing memory locality. ESXi combines the traditional initial placement approach with a dynamic rebalancing algorithm. Periodically (every two seconds by default), the system examines the loads of the various nodes and determines whether it should rebalance the load by moving a virtual machine from one node to another.

This calculation takes into account the resource settings for virtual machines and resource pools to improve performance without violating fairness or resource entitlements. The rebalancer selects an appropriate virtual machine and changes its home node to the least loaded node. When it can, the rebalancer moves a virtual machine that already has some memory on the destination node. From that point on, the virtual machine allocates memory on its new home node and runs only on processors in the new home node. Rebalancing is an effective solution to maintain fairness and ensure that all nodes are fully used. The rebalancer might need to move a virtual machine to a node on which it has allocated little or no memory. In this case, the virtual machine incurs a performance penalty associated with a large number of remote memory accesses. ESXi can eliminate this penalty by transparently migrating memory from the virtual machine's original node to its new home node.

NOTE

In vSphere 4.1/ESXi 4.1, the underlying physical NUMA architecture is not exposed by the hypervisor to the operating system, and therefore application workloads running on such VMs cannot take specific advantage of additional NUMA hooks that they may provide. However, in vSphere5, the concept of vNUMA was introduced, where through configuration you can expose the underlying NUMA architecture to the operating system, and so NUMA-aware applications can take advantage of it. In Java, the -XX:+UseNUMA JVM option is available; however, it is compatible only with the throughput GC and not the CMS GC. Paradoxically, in most memory-intensive cases where NUMA is a huge factor, latency sensitivity is a big consideration, and therefore the CMS collector is more suitable. This implies that you cannot use CMS and the -XX:+UseNUMA option together. The good news is that vSphere NUMA algorithms are usually good enough to provide locality, especially if you have followed good NUMA sizing best practices—such as sizing VMs to fit within NUMA boundaries for memory and vCPU perspective.

Most Common JVM Size Found in Production Environments

Having discussed thus far the various JVM sizes that you can deploy (in some cases, very large JVMs), it is important to keep in mind that the most common JVMs found in data centers are of 4GB heap size. This may be a fairly busy JVM with 100 to 250 concurrent threads (actual thread count will vary because it depends on the nature of the workload), 4GB of heap, approximately 4.5GB for the JVM process, 0.5GB for the guest operating system, and so a total recommended memory reservation for the VM of 5GB with two vCPUs and one JVM process, as shown in Figure 1-6.

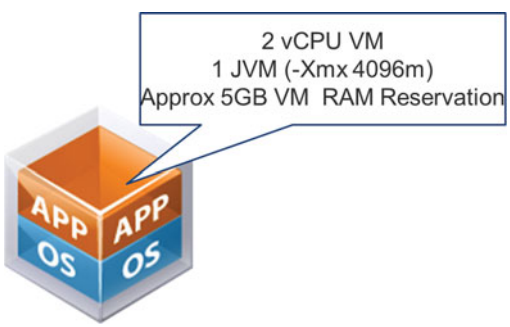

Figure 1-6 Most Common JVM Size Found in Production Environments

Horizontal Scaling Versus Vertical Scaling of JVMs and VMs

When considering horizontal scaling versus vertical scaling, you have three options, as Figure 1-7 shows.

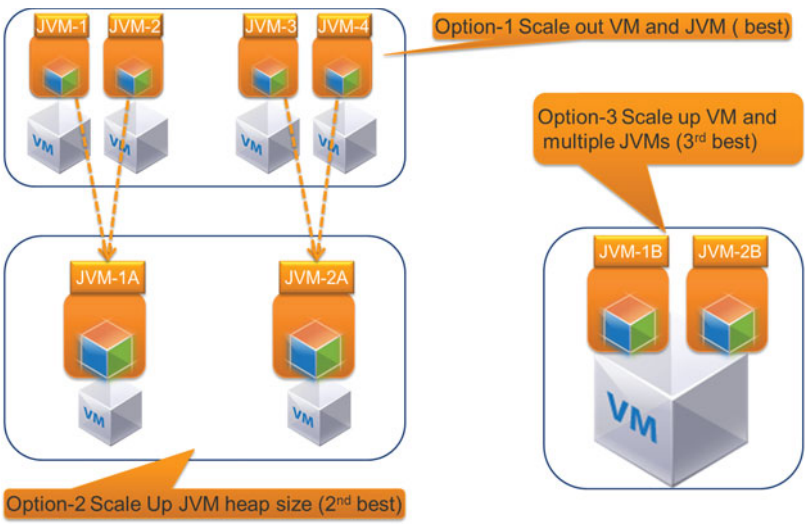

Figure 1-7 Horizontal Versus Vertical JVM Scalability Choices

The sections that follow detail the pros and cons of these three options.

Option 1

With Option 1, JVMs are introduced into the Java platform by creating a new VM and deploying a new JVM on it (hence, a scale-out VM and JVM model).

Option 1 Pros

This option provides the best scalability because the VM and the JVM are scheduled out as one unit by the ESXi scheduler. It is really the VM that is scheduled by the ESXi, but because there is only one JVM on this VM, the net effect is that the VM and the JVM are scheduled as one unit.

This option also offers the best flexibility to shut down any VM and JVM in isolation without impacting the rest of the Java platform. This is no doubt in relative terms, however, because most Java platforms are horizontally scalable, and in most cases there are enough instances to service traffic, even though JVM instances are being shut down. The relative comparison in terms of more instances having better scalability is based on having 100 JVMs and VMs versus having 150 JVMs and VMs for the exact same system, if for a

specific instance you where comparing and contrasting platform design and were trying to choose between 100 JVM systems versus 150 JVMs, with both cases of 100 and 150 JVMs having the same net RAM. Clearly, the system with 150 JVMs will have the better flexibility and scalability. In the 150 JVM scenario, because you have more JVMs, it is likely that the size of the JVM is smaller compared to a system that has 100 JVMs. In this case, if a JVM from the 150 JVM platform encounters a problem, likely the impact is smaller because the JVM holds less data than in the 100 JVM scenario. So, the scale-out robustness of the 150 JVMs will prove to be more prudent.

If the system has been refined, the horizontal scalability advantages assumed previously apply. *Refined* here means that VM and JVM best practices have been applied based on a 64-bit JVM architecture having a reasonable-size JVM with an approximate minimum of 4GB heap space, and not fragmented around a legacy 32-bit JVM limit of 1GB heap space. (Some legacy 32-bit JVMs could withstand greater than 1GB, but for practical use, 32-bit JVMs have a legacy 1GB limit.)

Option 1 Cons

This option is expensive because it leads to having more operating system copies, and licensing becomes expensive quite quickly. Administering such a system is more expensive because there are more VMs and JVMs to keep track of.

No technical reason requires you to place one JVM on one VM. The only exception is in the case of systems that are in-memory databases (like category 2) that require high throughput memory from the local NUMA node. In those cases, the VMs are sized to fit within the NUMA node and will have only one JVM on them. Also note that the JVMs in in-memory databases tend to be quite large, sometimes as big as 128GB, as opposed to category 1 JVM sizes (typically 1 to 4GB heap size). In cases such as option 1, however, which are essentially in category 1 (as defined earlier in this chapter), you have many opportunities to consolidate the JVMs and eliminate wasteful JVMs and VM instances.

This is a common pattern among legacy 32-bit JVMs, where the 1GB limit of the 32-bit JVM would have forced Java platform engineers to install more JVM instances to deal with increases in traffic. The downside here is that you are paying for additional CPU/licenses. If you consolidate JVMs by migrating to 64-bit JVMs and increasing the heap at the same time, you will save by having fewer JVMs servicing the same amount of traffic. Of course, the JVM size will likely increase from, for example, 1GB to 4GB.

Option 2

Option 2 involves scaling up the JVM heap size by consolidating fragmented smaller JVMs and also as a result consolidate VMs.

Option 2 Pros

The pros for using Option 2 are as follows:

- Reduced administration cost due to the lower number of JVMs and VMs
- Reduced licensing cost due to fewer operating system copies
- Improved response times as more transactions are (most likely) now executed within the same heap spaces, as opposed to requiring marshaling across the network to other JVMs
- Reduced hardware cost

NOTE

If you look at option 2 in Figure 1-7, it shows that two JVMs (JVM-1A and JVM-2A) were consolidated from the four JVMs (JVM-1, 2, 3, and 4) of option 1. In this process, the four VMs were also consolidated into two VMs, as shown in the figure. For example, if JVM-1, 2, 3, and 4 were all of 2GB heap size, each running on VMs of 2vCPU, this implies the total RAM serviced to the heap, and in turn to the application, is 8GB across all the JVMs. The total vCPUs across all the VMs is eight vCPUs. Now when consolidating down to two VMs and two JVMs, the JVMs in option 2 (JVM-1A and JVM-2A) are each of 4GB heap, for a total of 8GB, and the VMs are two vCPUs each. This implies a total of four vCPUs across both VMs, a savings of four vCPUs, because originally in option 1 there were four VMs of two vCPUs each.

It is possible to scale down vCPUs while still maintaining an equal amount of RAM (Java heap space) because with larger JVM heap spaces, GC can scale vertically fairly well without having to excessively consume CPU. This is largely workload behavior dependent, and some workloads may indeed exhibit increased CPU usage when JVMs are scaled up. However, most category 1 workloads have exhibited a behavior of releasing the unneeded vCPU when consolidated into a larger JVM heap. 64-bit JVMs are highly capable runtime containers, and although there is an initial cost of launching one, they do enable you to crunch through a massive number of transactions that are within much larger heap spaces. When you are thinking about creating a new JVM, you want to ask the same questions as if you were about to create a new VM. If someone needs a new VM, a vSphere administrator always asks why it is needed. Because the JVM is a highly capable machine (just as the VM is a highly capable compute resource), vSphere administrators and DevOps engineers should always scrutinize whether the creation of a new JVM is necessary (as opposed to leveraging existing JVM instances and perhaps increasing the heap space, within reason, to facilitate more traffic).

Option 2 Cons

The cons for using Option 2 are as follows:

- Because of the larger-size JVMs, you risk losing more data (when compared with the case of smaller JVMs in option 1) if a JVM crashes without proper redundancy or persistence of transactions in place.
- Due to consolidation, you might have fewer high-availability (HA) JVM instances.
- Consolidation is limited to line of business. You do not want to mix applications from different lines of business into the same JVM; a crash of the JVM would impact both lines of business if you were to mix two into one JVM.
- Larger JVMs may require some more GC tuning.

Option 3

If option 1 and option 2 are not possible, consider option 3. In this case, you are placing multiple JVMs on a larger VM. Now JVM-1B and JVM-2B could be JVMs that are consolidated copies, like the ones in option 2, or nonconsolidated copies like in option 1. In either case, you can stack these JVMs on a larger VM, or multiple large VMs for that matter.

Option 3 Pros

The pros for using Option 3 are as follows:

- If the current platform is similar to that in option 1, it might be an advantage, because of logistical reasons, to keep the current number of JVMs intact in the deployment, but then consider building larger VMs with multiple JVMs stacked on them.
- Reduced number of operating system licenses.
- Reduced number of VM instances.
- Reduced administration cost due to having fewer VMs.
- You can have dedicated JVMs for each line of business but can also deploy JVMs from multiple lines of business on the same VM. You should do this only if the cost of VM consolidation outweighs the danger of having multiple lines of business impacted during a VM crash.
- Large VMs makes it possible to have more vCPUs for JVMs. If a VM has two large JVMs on it from different lines of business, for example, and they peak at different times, it is likely that all the vCPUs are available to the busy JVM, and then similarly for the next JVM when its peak arrives.

Option 3 Cons

Larger VMs will most likely be required. Scheduling larger VMs may require more tuning than smaller VMs.

> **NOTE**
>
> Various performance studies have shown that the sweet-spot VM size is two vCPUs to four vCPUs for category 1 workloads. Category 2 workloads require more than four vCPUs; at a minimum, four vCPUs may be needed. Remember, though, that scheduling opportunity from an HA perspective may be diminished. However, category 2 workloads, as in-memory databases, are mostly fault tolerant, redundant, and disk persistent, and therefore might not rely as much on VMware HA or automatic Distributed Resource Scheduler (DRS).

Because this option is about trying to consolidate VMs, it is highly likely that JVMs from different lines of business may be deployed on the same VM. You must manage this correctly because inadvertent restart of a VM may potentially impact multiple lines of business.

You can attempt to consolidate JVMs in this case and also stack them up on the same VM; however, this forces the JVMs to be much larger to fully utilize the underlying memory. If you configure fewer larger VMs, it literally means that you have VMs with a lot more RAM from the underlying hardware, and to fully consume this you might need larger JVM heap spaces. Because of the larger-size JVMs, you risk losing more data if a JVM crashes without proper redundancy or persistence of transactions in place.

This option might require large vSphere hosts, and larger servers cost more.

Summary

This chapter introduced the concept of large-scale Java platforms and described how they generally fall into one of three categories:

- **Category 1**: Large number of JVMs
- **Category 2**: Smaller number of JVMs with large heap sizes
- **Category 3**: A combination of category 1 and 2

The chapter also examined the various theoretical and practical limits that exists within the JVM and outlined various workload types and commonly encountered JVM sizes. The chapter also discussed the NUMA and the various pros and cons of horizontal scalability, vertical scalability, JVM consolidation, and VM consolidation.