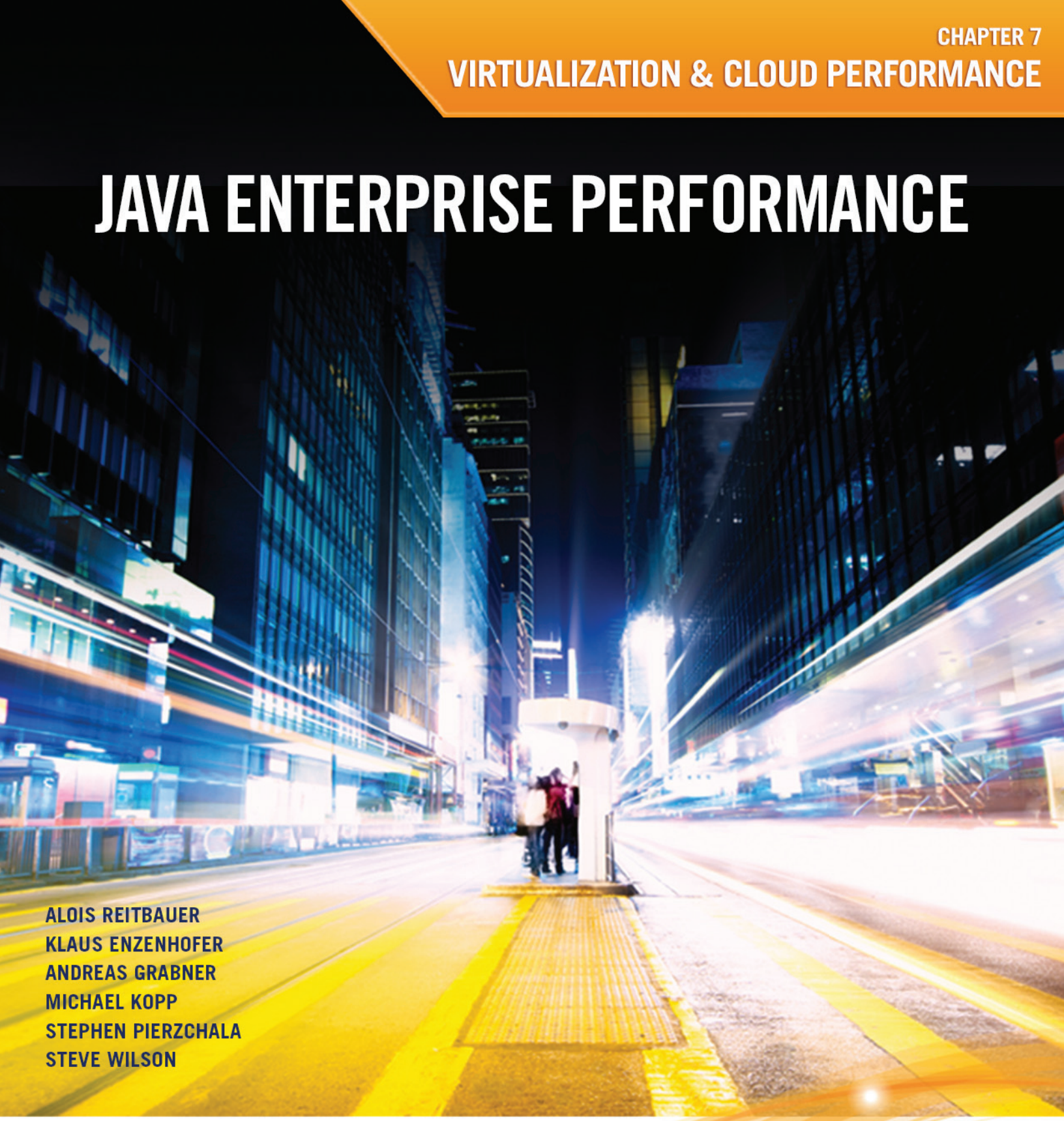
CHAPTER 7

VIRTUALIZATION & CLOUD PERFORMANCE

JAVA ENTERPRISE PERFORMANCE



ALOIS REITBAUER
KLAUS ENZENHOFER
ANDREAS GRABNER
MICHAEL KOPP
STEPHEN PIERZCHALA
STEVE WILSON

Compuware
THE TECHNOLOGY PERFORMANCE COMPANY

Java Enterprise Performance

Chapter 7

Introduction to Performance Monitoring in Virtualized and Cloud Environments

By

Michael Kopp

Chapter 7

Introduction to Performance Monitoring in Virtualized and Cloud Environments

What's in this chapter?

1. IaaS, PaaS, and SaaS—All Cloud, All Different
2. Virtualization's Impact on Performance Management
3. Why Running Isolated VMs Hides the Root Cause of Performance Problems
4. Monitoring Applications in Virtualized Environments
5. Monitoring and Understanding Application Performance in the Cloud
6. Performance Analysis and Resolution of Cloud Applications

Cloud computing seems to be everywhere these days. Some hail it as the future of IT, while others see it as an overhyped technology that brings nothing new to the table. Both arguments have some truth to them; let's understand why this is.

It's unlikely that cloud computing could have been realized without the underlying technology of virtualization, which has been an integral aspect of computing since the '50s. This is why many view the cloud as evolutionary and not revolutionary.

Historically, virtualization has fulfilled two main purposes:

- Hardware emulation, which we will ignore, as it is not important in terms of mainstream IT
- Running multiple isolated systems within a single physical hardware environment

Mainframes have been able to run multiple isolated systems on single big machines forever, but it is in the last 20 years that this form of virtualization has entered the mainstream.

We can summarize the advantages of virtualization in two words: manageability and utilization.

Using virtualization, IT departments can manage a large number of systems on comparatively few physical machines. This is accomplished by adding a slim software layer between the guest operating system (the VM) and the physical hardware. This layer of middleware, known as the hypervisor, is like a traffic director. Requests from the guest system are either sent to the hardware or queued because a request from another VM had priority and was sent to the hardware first. In this way much of the administration can be done remotely, even moving VMs to different hardware or allocating more memory without restart.

At the same time, virtualization allows us to achieve greater hardware utilization. From the perspective of a performance expert, this might not always be desirable—higher utilization increases the potential for negative performance impacts on the application. But from an operations perspective, higher utilization means potentially less hardware. Ultimately, this is why virtualization is so desirable and successful: less hardware and, more importantly, lower operational cost. This key point has given birth to the cloud!

Virtualization has provided IT with the ability to provision new systems quickly without the need to buy extra hardware. In turn, this flexibility has made it possible to start, stop, and move deployments ever more quickly. A new configuration and management layer has been added to help set policies and control virtual-machine assignments and hardware allocations. While this is all achievable with less effort than it would without virtualization, the flexibility and agility demands can become a burden for large IT organizations. In response, IT organizations began automating parts of the process. Some of the biggest, like Amazon, went ahead and automated it all—and thus, the first private clouds were born.

But let’s back up to the basics for a moment. Computing clouds provide on-demand provisioning of VMs and other resources via the network—without manual intervention or physical access to those resources. Therefore, every cloud is controlled by a set of underlying policies to assure the best possible utilization of available resources while providing each and every VM the resources needed to run efficiently and effectively.

Amazon took this cloud-computing idea from the ‘60s and did something completely logical, but at the same time totally unprecedented. Having created a colossal IT infrastructure, Amazon decided to rent out some of its excess capacity during nonpeak hours to others online, turning what had been a private cloud into the first public cloud. In one blow, Amazon became the technology leader in cloud computing and created a market where none had previously existed! Not surprisingly, others, along with many of the large hosting companies, quickly followed suit.

Both the cloud and virtualization came from the relentless quest of IT operations to reduce cost and improve operational efficiency. Interestingly, by making provisioning

automatic, the traditional infrastructural building blocks of IT are being demoted, which means the business importance of applications is increasing. This side effect has been embraced by the industry and has given rise to the newest trends in cloud computing: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

IaaS, PaaS, and SaaS—All Cloud, All Different

As mentioned in the preceding section, the importance of applications in a virtualized environment has brought about three major trends in cloud computing:

Infrastructure as a Service (IaaS): Used to manage *low-level* resources like VMs and disks. The end user is responsible for what is running within the VM, starting with the OS. IaaS is most closely related to a *regular* automated virtualized system. Amazon Web Services is maybe the best-known provider for an IaaS-style cloud service, but there are numerous others in the market.

Platform as a Service (PaaS): Provides faster development and deployment platforms by abstracting the user from the OS while adding well-defined APIs to many essential services (such as the Web, databases, mail, queues, and storage) that the developer must use. Both sides benefit; development should be faster and the end product should be more reliable more quickly. Typically PaaS also provides monitoring and administrative consoles tightly designed around the platform, making operations easier. At the same time, by maintaining tight control over what is executed when, PaaS can make better decisions about machine load, at least in theory. Another side effect is a vendor lock-in, desirable to the vendor but maybe not to the developer. Windows Azure is a good example of a PaaS-style cloud.

Software as a Service (SaaS): Instead of writing and maintaining every application, one uses online services. Examples include Google Mail and Salesforce. It is important to note that the SaaS software provided by a vendor might not be running in a cloud, but SaaS offerings often make sense when developing an application for the cloud. For instance, if I develop an e-commerce solution in the cloud, I might not want to reinvent my own load balancers, databases, or payment-processing system. I might not even want to run or operate them; perhaps all I want is to use these services. Using SaaS vendors allows me to concentrate on my own application.

Virtualization, as well as the three cloud platforms discussed, have a major impact on the discipline of application performance management (APM).

Cloud computing–performance management is not only a big technical challenge, but in my mind, a real game changer, as well. By making infrastructure a throwaway commodity, focus is shifted to the application by default. At the same time, you are

losing a little bit of control and, in the case of a public cloud, visibility, making your application and operation more dependent on third parties.

PaaS and SaaS go one step further and outsource parts of the application itself, putting ever more focus on our core business and core application while depending even more on third-party black-box services. Therefore, APM comes into its own in the cloud. In the next section we'll look at the implications of virtualization on performance management.

Virtualization's Impact on Performance Management

As a performance expert I have long been suspicious of virtualization—not because virtualization doesn't work, but because it hides so much of what it is doing. For me and many other performance experts, that lack of transparency makes virtualization problematic in terms of performance management.

Think about it. Each virtualized operating system, or guest, is hosted inside a virtual machine, and multiple guests share the same hardware, though none has exclusive access. This sleight of hand is managed by the hypervisor, which provides the virtual operating platform for each of the guests.

The hypervisor adds an extra layer on top of the hardware operating system so that it can schedule and route system requests from simultaneously executing applications running on the multiple guests. It's a complex task to ensure that everything gets to the right place at the right time without interference. Doing so efficiently is even harder, which is why there are two important side effects of virtualization:

- The hypervisor adds a measure of overhead to every hardware request. This is typically minimal, but one can only be sure by monitoring the overhead.
- Hardware is shared and finite, and no guest is assured instant access to these resources. Delayed availability of the CPU can cause increased latency, known as steal time, which can seriously degrade performance and scalability in ways that standard utilization metrics are unable to measure. This is true for all resources (I/O, memory, network, and so on), not just CPU.

For the purpose of application performance monitoring, the difficulties associated with virtualization have less to do with the impact of added overhead per se than with measuring this impact on application performance. It's this difficulty that makes hardcore performance people like me suspicious.

Sharing Resources in a Virtual Environment: Setting Priorities and Exaggerating the Truth

Virtual environments, while created in software, can exist only within the physical hardware provided. CPU, memory, disk storage, and network access are shared among

the several virtual machines, and each VM is assigned specific resources as needed. Processing appears to execute as it would on any non-virtualized machine, but there are two major differences:

- The number of available resources (CPU, memory, etc.) as seen by the VM are not necessarily physically available.
- The Guest OS does not get exclusive access to assigned resources, which it doesn't know.

Why We Stretch the Truth, and the Impact on Performance

The virtualization software itself requires resources, limiting what's available for the VM. Also, resources are not shared fairly, but are allocated according to preconfigured settings and policies. Most importantly, virtualization is used to achieve better utilization, and overcommitting resources is the most efficient way make this happen.

Let's say we are running five VMs on a machine that has eight CPUs and we assign two CPUs to each VM. As long as the virtual machines are not simultaneously using 100% of their resources, things should run fine. And because we can do a given amount of virtualized work on less physical hardware than we would need for non-virtualized work, our environment is being used more fully and efficiently.

When a system's utilization approaches 100%, the same logical rules that operating systems use for multiple processes are employed: one or more of the VMs must wait. This means that at any specific point, the guest system may not be able to achieve 100% CPU utilization. That is no different than the fact that a process might not get 100% of the available CPU on a regular system.

But there is more. Instead of just assigning the number of CPUs, many virtualization systems can assign a range of GHz to a specific VM (e.g., 1000–2000 GHz). If a VM has used up its share in a given time slice, it has to wait.

And here's the kicker: since the VM is ignorant of this throttling, internal time measurements and calculations of CPU utilization go completely out of whack! CPU utilization at the guest level has a different meaning from one second to the next, or rather, it really has no meaning. I will explain this phenomenon in detail shortly.

Why Memory Balloons Can Blow Up Your Performance

Sharing CPU resources is relatively straightforward compared to sharing memory. The utilization strategy is the same: more virtual memory is assigned than is physically available. The virtualization system tracks individual allocations in each guest, as well as total memory utilization for all guests. When physical memory is used up, the next VM allocation can only be made by reclaiming some of the memory in use. This is achieved in two ways:

- The hypervisor transparently swaps a portion of guest memory out of physical memory onto disk. To access the portion swapped out, the hypervisor simply swaps it back in, just like normal OS-level swapping. Problems arise because of indiscriminate swapping at the hypervisor level, which seriously degrades performance; for example, when portions of an active JVM are swapped out. Often, virtualization systems will install a balloon driver to coordinate swapping and alleviate this problem, which leads us to the second method for utilizing memory.
- Ballooning makes the guest operating system aware of the low-memory status of the host. To reclaim memory, the hypervisor gives the balloon driver inside the guest a target balloon size. The driver will then inflate the balloon by allocating guest memory and pinning the underlying memory pages. Pinned pages are not swapped to disk. The hypervisor can then reclaim those pinned pages and reassign them to another VM (*Figure 7.1*). It can do that because the original guest is not using those physical memory pages. This works quite well, but it also means that a guest with a balloon might be forced to swap out other areas itself. This is preferable because the guest can make an informed decision about what to swap, whereas the hypervisor cannot.

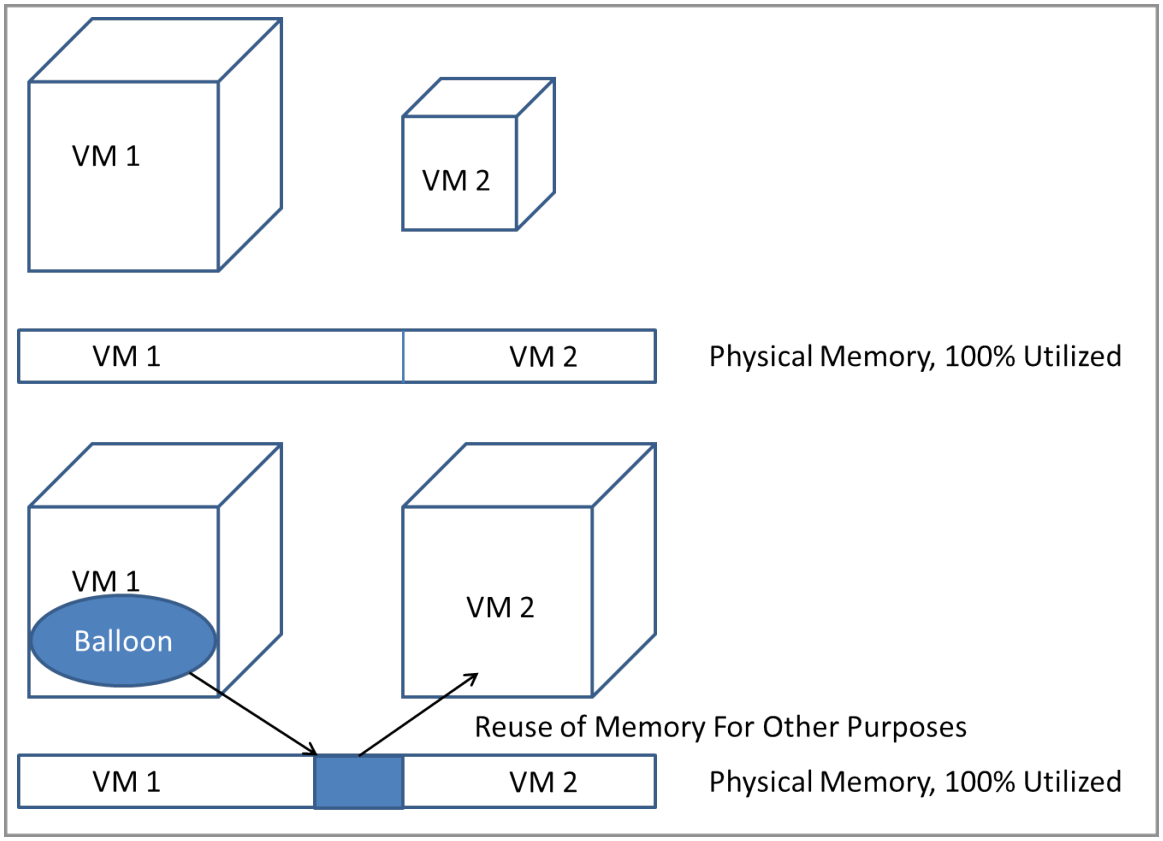


Figure 7.1: How memory ballooning is used to reassign memory from one VM to another

Overcommitting memory leads to better resource utilization because systems seldom need all of their assigned memory. However, too much overcommitting leads to swapping and can degrade performance. This is especially true for Java applications.

Since the JVM accesses all its assigned memory in a random fashion, large portions of assigned memory are accessed throughout the JVM's lifetime. This means that no part of a JVM can be swapped out safely without negatively impacting performance. For this reason, it is essential to understand swapping behavior and the impact this has on applications running within the VM.

Why Running Isolated VMs Hides the Root Cause of Performance Problems

Because resources are shared in virtual environments, VMs, though isolated from each other in terms of memory and access, can nonetheless affect each other in terms of performance (*Figure 7.2*). This can be problematic because although we can see the impact in terms of performance, the underlying root cause stays hidden.

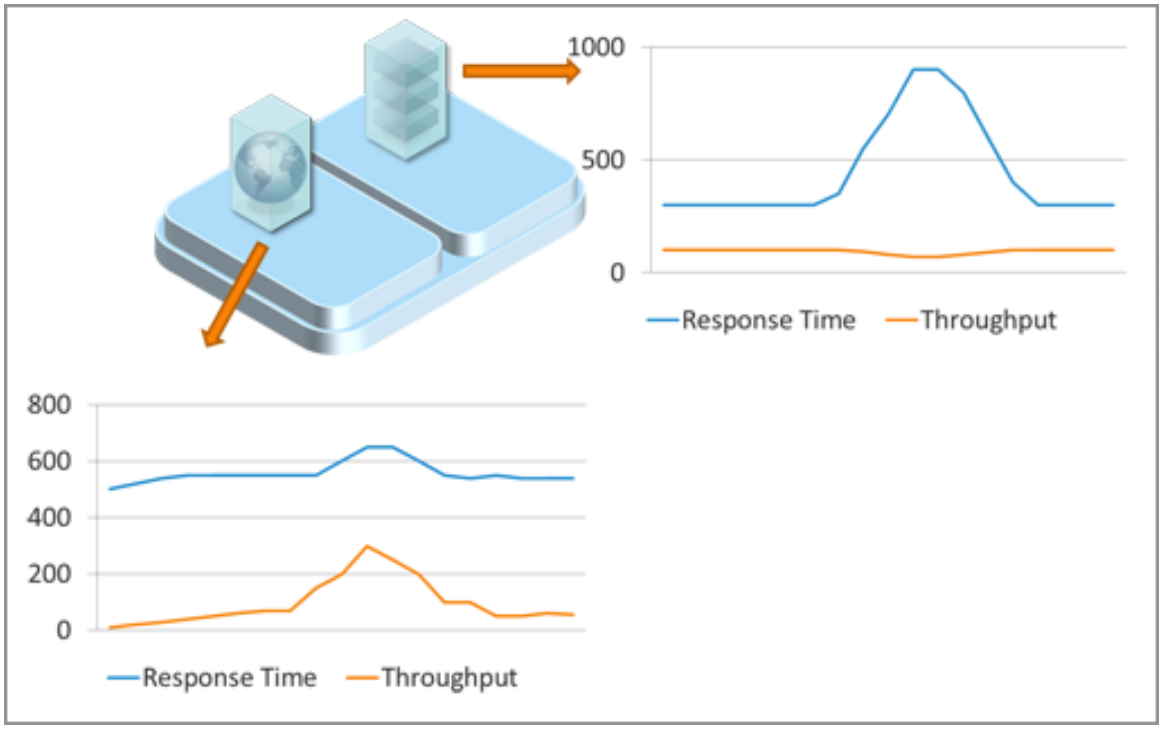


Figure 7.2: A spike on one application can impact the performance of another even though they run in separate VMs.

Let's say the load increases on one application to the point where the maximum hardware allocation is met. The VM is either stalled (has to wait), or it begins to steal resources from other VMs running on the same host. This creates a hardware shortage

that can cause other VMs to slow down, affecting the responsiveness of other applications running.

Sounds like an obvious enough problem, but it can be very difficult to find the root cause in such a situation. A problem caused by an application running in one VM shows up only in the second VM. Analyzing the application in the second VM fails to identify the hot spot because it's the wrong place to look.

Even worse, in many organizations the team responsible for an application has no knowledge or insight into what else is running on the underlying hardware. To quote a customer, "We only know that something else might be running if the performance numbers look funny."

However, by looking at CPU, memory, and I/O utilization at the virtualization level, we see that the first VM's CPU usage is increasing while the second VM's is decreasing. At the same time, the hypervisor reports CPU steal time for the second VM, which we see is also waiting for CPU resources. In order to understand what is really going on, we need to correlate the performance measurements from both applications using monitoring data at the hypervisor layer.

What Tricky Time Measurement in a VM Means for APM

Einstein told us that time is relative. He might just as well have been talking about virtualization. But in a VM, time is relative and has holes.

The timekeeping problem is well known in the VMware community (http://apmblog.compuware.com/2009/09/23/the-problem-with-sla-monitoring-in-virtualized-environments/). It results from de-scheduling VMs without acknowledgement and is compounded by the archaic PC timer architecture, unchanged since the early '80s.

A VMware whitepaper (http://www.vmware.com/files/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf) explains this in quite some detail. However, it doesn't discuss other virtualization solutions, such as Xen, KVM, and Hyper-V, that attempt to solve this problem in different ways. But since various guest operating systems behave very differently from one another, time measurement inside a guest is not to be trusted.

When the hypervisor suspends a guest system, it is just like an OS suspending one process to execute another. But whereas the OS is responsible for timekeeping, the guest system won't usually know when it is suspended and can't account for this when tracking time.

Windows and Linux systems track time by counting the periodically triggered Interrupts. A suspended guest can't count Interrupts as none are triggered, and thus it quickly falls behind real time (*Figure 7.3*). To fix this, the hypervisor records Interrupts and replays them when the VM is resumed. This ensures that the time in the guest keeps up with real time, but it leads to another problem. After resuming, a lot of queued interrupts are

processed in quick succession. Because the Interrupts represent time slices, time effectively speeds up!

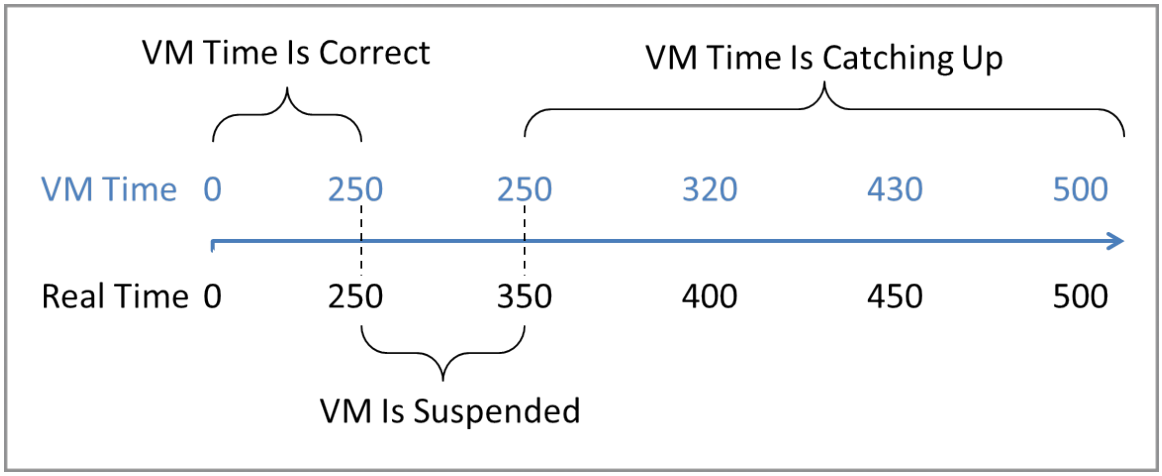


Figure 7.3: The time as seen by the VM and real time can deviate due to VM suspension and the way systems track time.

As a smoothly running system would never suspend a guest for more than tenths or even hundredths of a millisecond, this represents a fine solution for most day-to-day applications. But it is a big problem for monitoring and detailed performance analysis, for two reasons:

- Measured response times in a VM cannot be trusted on a millisecond granularity.
- Method-level performance analysis is a problem because a method might appear to be more expensive (suspension or catchup) than it actually is.

Modern Linux systems and para-virtualizing solve the speedup problem by using tickless timers. A tickless timer does not need periodic interrupts to function; thus there is no catchup problem. For other systems, hypervisors often provide separate virtualization-aware times that our monitoring tool can use. Such systems allow near-accurate measurement of response time. However this still leaves us with the suspension and its impact on method-level analysis.

The suspension problem can be solved by correlating the suspension times with the performance measurement, thus accounting for it. This is very similar to correlating garbage-collector suspensions with the method-execution times. The easiest way to do this is to look at CPU ready time (VMware) or CPU steal time (Xen, Amazon EC2) as reported by the hypervisor or para-virtualized guest.

There is one additional timekeeping issue, which renders our inability measure time accurately essentially unimportant—the implications for system monitoring.

Why System Metrics in the Guest Are Not Trustworthy

APM and monitoring solutions can use virtualization-aware timers and take steal time into account, at least for their own measurements. The guest OS is not doing this! Its time tracking is flawed and that has serious side effects on all system metrics that depend on a time, meaning all utilization (percentage) and rate measures (e.g., requests/second).

For example, CPU utilization is calculated based on CPU cycles available and used within a given timeframe. As the guest system doesn't get the time right, the calculated CPU utilization is simply wrong. The guest cannot deal with the fact that it was not running during the suspension. It even thinks that it and some of its processes consumed CPU resources during suspension (remember, it doesn't know that it wasn't running). VMware and others deal with this by telling the guest system that a special dummy process was running during that time frame. Or they might tell it to account for it under a different metric—e.g., steal time. By doing this they ensure that any CPU usage of process in a guest is mostly accurate, but the CPU utilization (on the guest and process levels) is not!

Here we have the solution to the conundrum. Instead of trusting the CPU utilization we must turn to CPU usage and time measurements at the application level. Not only do the virtualization systems try to ensure that CPU usage (not utilization!) is accurate on the process level, but our APM solutions can measure steal time and CPU usage at the hypervisor level and correlate this correctly to our application measurements. That brings us to the main subject—how to monitor applications in virtualized environments, discussed in the next section.

Monitoring Applications in Virtualized Environments

Virtualization vendors provide built-in tools for monitoring virtual machines and their underlying hosts. One can generally obtain metrics for utilization and throughput, and sometimes for latency of the virtual infrastructure. This allows us to maximize utilization while keeping latency measures low, but we lack the context to guarantee that our application runs smoothly. Without understanding the impact of virtualization itself, we can't understand how hardware latency or utilization actually affect an application's performance.

In a virtual environment, the concept of real time is difficult. We'd like to continue using transactional performance as our measure of optimization, but how do we go about measuring the response time of a single transaction in a virtual world (*Figure 7.4*)? The guest system has no awareness of real time, so we must either use some form of virtualization-aware timer, like a tickless timer, or use something like a network appliance to measure outside of the virtual machine.

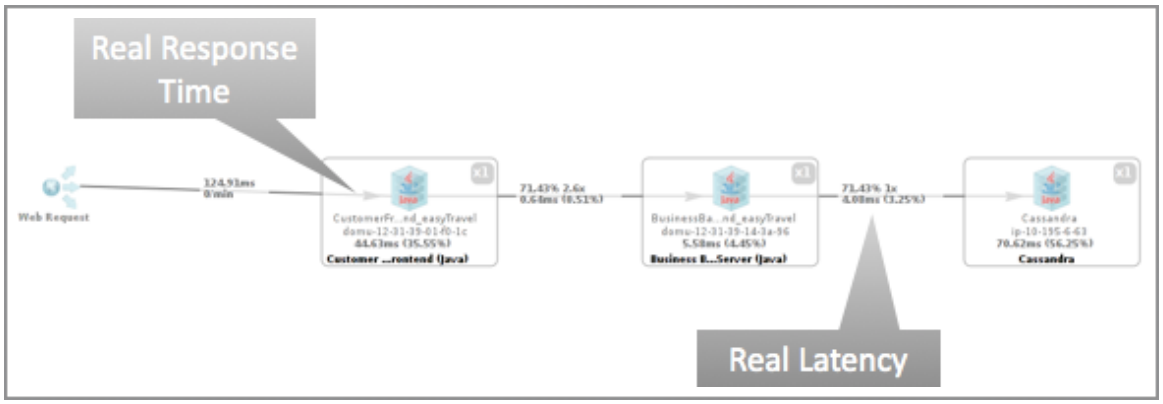


Figure 7.4: We need real response time and real latency time to identify the fault domain.

To identify latency issues in a distributed system (as depicted in *figure 7.4*), we need real response time metrics for every tier from both the client and server side of a call (see red boxes *figure 7.5*). To understand the cause, we need to measure the transaction load on the application and how inter-tier latency is affected by that.

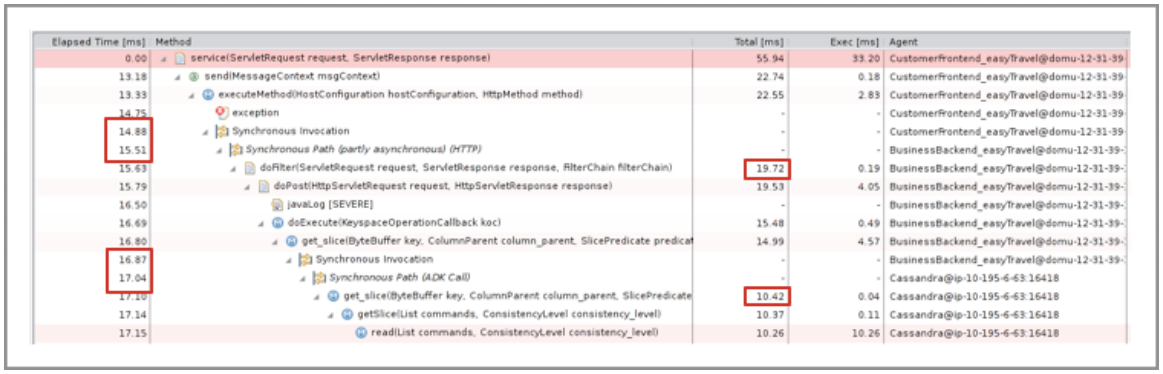


Figure 7.5: Real time on exit and entry points helps identify latency issues and the fault domain.

If inter-tier response time is rising with otherwise-stable tier response times, then we most likely face an overloaded virtual network. We can either reduce the load in terms of application communication or transactions, or talk to the network and VM administrators.

If the response time of a tier is rising while transaction load and CPU consumption remain stable, we likely have de-scheduling issues. We can check this by correlating the CPU ready/steal time to our tier response times. If we see an equivalent rise in steal time, we know that the VM doesn't get enough CPU time allocated, which again results in a walk to our VM guy's office.

Defining Key Virtualization Metrics

The list of virtualization metrics is long and can be very daunting at the start (see *figure 7.6*). For this reason I have identified a list of key metrics that focus on measuring how virtualization and resource shortage impact the application.

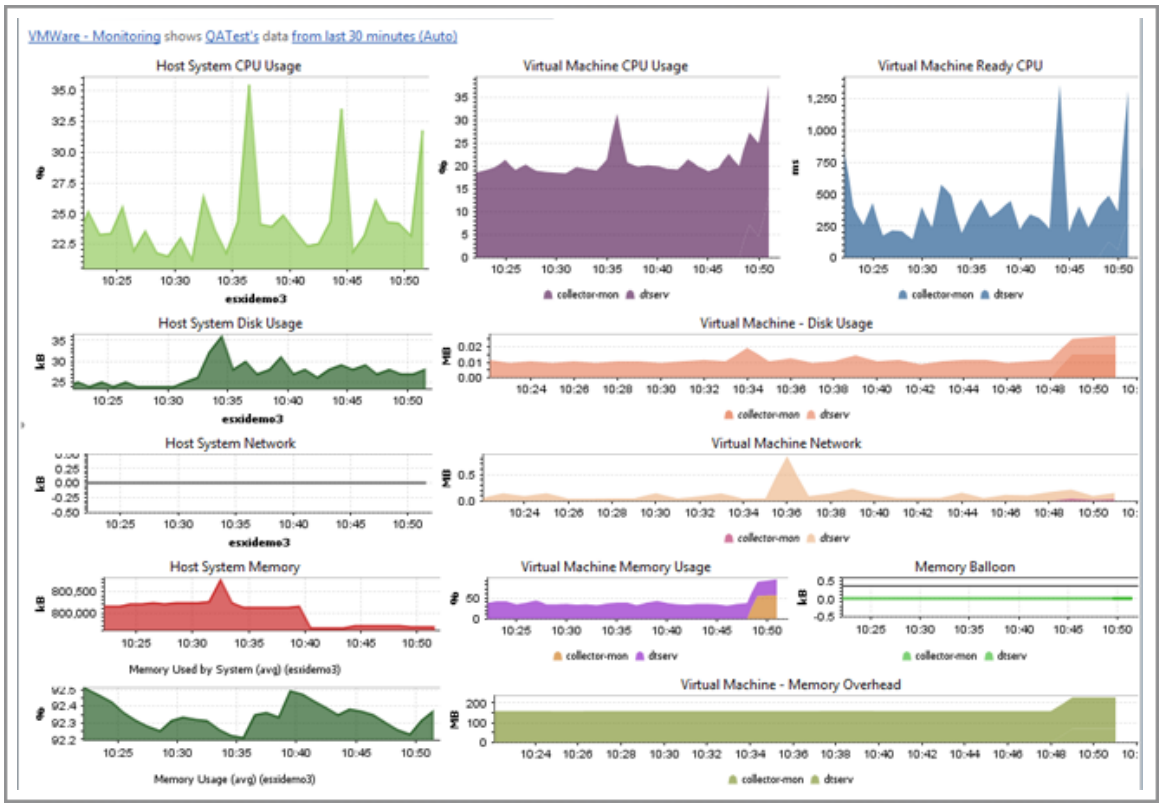


Figure 7.6: A typical monitoring dashboard for a VMware instance

- **CPU Ready Time/CPU Steal Time**—A measure of overhead caused by the hypervisor. More specifically, time during which the VM was suspended and therefore unable to execute CPU instructions. There is always a small, measurable overhead, but it should never grow beyond the 5% range. If it does, the underlying hardware is overloaded. Much as garbage-collector suspension time correlates to application performance, there is a direct correlation between steal time and response time. One can even express steal time as a percentage of response-time-delay. With this, one can actively monitor the impact of virtualization on application performance.

- **CPU Swap Wait**—Time the VM must wait while the hypervisor swaps parts of the VM memory back in from disk. This is caused when the hypervisor itself needs to swap, either because the guest has no balloon driver installed or because there is a critical memory shortage. This measure can also be directly related to

application response times. As a solution, install the balloon driver and/or reduce the number of VMs hosted on the physical machines.

- **CPU System/Wait VKernel**—As a measure of CPU resource overhead, most virtualization systems report CPU or wait time for the virtualization kernel itself, consumed per VM. Although this cannot be correlated directly with response time, if it grows significantly it will likely impact ready time and swap time. The system is either misconfigured or overloaded, and you need to reduce the number of VMs on the machine.
- **Memory Balloon**—When under memory pressure, the host system will steal memory from its VMs via ballooning. Memory ballooning represents a shortage of memory with serious performance implications at the guest level, including swapping, smaller file-system buffers, and smaller system caches.
- **Memory Swap**—The amount of swapping for each VM performed by the virtualization infrastructure. High swap rates increase the CPU swap time and always have a negative impact on application performance. Therefore, the swap rate is the better metric than swap usage because it represents real impact. Note that swapped-out memory has no impact if unused, and if it is swapped in, then the swap rate will reflect the impact.
- **Memory Usage: Host, VM, and Granted**—Memory usage can be measured at the host level, the VM level, and as granted memory. Memory is allocated to the VM only when a guest uses it. Once allocated, the used memory segment is considered granted. Most systems cannot (or will not) reclaim memory, and there's little point in trying, because most guests use any extra for buffers and caches and never relinquish what has been granted. Therefore, we can know a VM's physical-memory usage by looking at granted memory. The difference between used and granted memory represents the rate of overcommitment, and this can be indirectly reassigned via ballooning.
- **Disk/Network Latency**—Some virtualization vendors provide built-in latency measures for disks and network interfaces used by a VM. Since there's a direct correlation between latency and response times, any increased latency at the hypervisor level will result in latency on the application, as well. Too much latency points to a system overload, indicating a need to reconfigure.

These metrics enable us to monitor and detect any negative impact that a misconfigured or overloaded virtualized system might have on our application. As a next step we need to understand how the cloud is different and why it represents an even greater challenge.

Monitoring and Understanding Application Performance in the Cloud

While proponents believe that the cloud solves all problems, skeptics see nothing beyond large-scale virtualization with all its inherent compromises. Which is correct and what are we to think about the desirability of cloud implementations?

It's true that clouds are typically, though not necessarily, implemented as virtualized systems, this isn't an argument for or against a cloud-based solution. Similarly, clouds make little sense for very small environments, but this is no reason small applications can't take advantage of an existing cloud. What matters most is that one can provision new VMs in the cloud without making explicit hardware assignments. It's all done with templates and handled by cloud management.

From an operations perspective this is a huge cost- and time-saver, and it makes it feasible to schedule load tests during off-peak hours while ensuring priority for production VMs. At the rather large scale typical for cloud computing, this can significantly reduce hardware and operations.

The Dark Side of Clouds: Changing Layers of Complexity

However, the key cloud advantages of self-service, automatic provisioning, and rapid elasticity must be bought at the price of increased complexity at the application level. Each newly provisioned instance can have a hidden impact on the performance of already-running applications—an impact that may be visible only when we look at the underlying shared infrastructure.

Cloud features such as automatic provisioning and workload management allow us to ignore the relationship between VMs and assigned underlying hardware, but our applications must still run on the actual hardware. So while we may be blissfully ignorant of these assignments, when things go wrong we need to be able to track down the problem.

For example, Application A might impact Application B today, yet tomorrow it might *steal* CPU time from Application C! How are we to tell? The inherent dynamism of clouds presents significant challenges when it comes to monitoring and application performance management.

Things aren't much better for application owners, who often have no access to the hypervisor layer, usually managed by a separate group, making it nearly impossible to do proper troubleshooting and application tuning.

What's needed is visibility; a means to monitor applications, the VM, and the hypervisor layer at the same time. It's the only way we can trace cause and effect when physical hardware is shared by two running applications. With such an integrated application-monitoring solution, we are able to gather application-level metrics such as response

time, load patterns, and resource usage—the information required to ensure application performance and achieve optimal utilization.

Monitoring cloud applications, though not fundamentally different, is more complex because of the added layer of indirection. But APM also gives us the means to run performant applications cost-effectively, which takes us to the next section of our chapter.

Making Cloud Applications Performant and Cost-Effective

With clouds, we're able to automate the process of dynamic resource allocation based on changing application demands. At the same time, a cloud, whether public or private, cannot make our applications run any faster. In fact, no transaction will execute more quickly in a cloud. Therefore, the wonderful flexibility we gain from cloud computing must be carefully balanced against the inherent need to be ever more aware of transactional efficiency.

The process of performance tuning remains much the same, with the addition of the metrics just discussed. But there is yet another layer of complexity added by the use of third-party services over which one has no control. For example, most cloud vendors provide database services. Like everything else, these services impact transactional performance and must be monitored.

Similarly, as we add VMs to an application, which is simpler and more flexible than changing resource allocations, we do not want to over-provision. For example, adding a huge VM instance when we need 5% more power is overkill (huge in the sense that it requires a lot of CPU and memory resources). Instead, increasing available hardware in small increments helps to maintain the advantages of flexibility.

The use of smaller, less power full VM instances has two logical consequences:

- The number of instances per tier tends to increase, resulting in a very large number of tiers overall.
- Elastic scaling can become necessary for each tier.

At the same time, clouds are designed to accommodate massive horizontal scaling, which means that we're likely to have a greater number of tiers than with an equivalent data center deployment, and with a greater number of nodes. As you might guess, all of this adds to the complexity of the scaling logic, making it more difficult to find performance issues and optimize applications. This is a task that most monitoring tools and scaling mechanisms were not designed to deal with!

As difficult as this sounds, realizing the cost advantages of the cloud means that we must leverage this elastic scalability. To achieve this and maintain the desired single-

transaction performance, we need to rethink the way we design and monitor our applications.

Cloud-deployed applications must be inherently scalable, which means:

- Avoiding all synchronization and state transfers between transactions. This limits sharing and requires a tradeoff in terms of memory and caching. For distributed data, we must use inherently-scalable technologies, which might disallow a particular SQL solution that we are used to.
- Optimizing the critical path so that each tier remains as independent as possible. Essentially, everything that is response time–critical should avoid extra tiers. The best-case scenario is a single tier in such cases.
- Using queues between tiers to aid scaling. Queues make it possible to measure the load on a tier, the queue depth, which makes scaling the consuming tier very easy.

But to achieve the full cost-effectiveness promised by cloud computing, a well-designed application must leverage elasticity as well as scalability. This requires a level of application monitoring to collect data on response time and resource usage, and to measure algorithmic scaling load.

That brings us to the major difference between public and private clouds: cost-effectiveness and elastic scaling.

Public and Private Clouds: Siblings with Different Performance Goals

Before we go on to discuss application monitoring in the cloud, we need to distinguish between public and private clouds. This has little to do with technology and much to do with visibility, goals, and ownership.

In traditional enterprise computing environments, there are two conflicting goals:

- Maximizing utilization, while minimizing hardware
- Optimizing application performance

Virtualized and cloud environments are no different in this respect.

Private clouds have the advantage of being run by the same organization responsible for application performance and IT operations. We're able to enforce application performance management across all layers, which then enables us to optimize for both goals.

In the public cloud the underlying virtualization is opaque. We can neither manage nor optimize it according to our needs or wishes. We must adopt a black-box approach and optimize with a single goal of fulfilling response-time requirements while maintaining

scalability. As if by magic, we're no longer faced with conflicting goals, because it's up to the cloud vendor to optimize resource utilization; it's not our concern!

At the same time, we're no longer faced by the limitations of finite resources. In a private cloud, once our application performance goals are met, we continue to optimize in order to reduce resource usage. In a public cloud, available hardware is of no concern.

Obviously, we're still driven to reduce costs, but this is calculated on the basis of instance time (not CPU usage), disk accesses, network traffic, and database size. Furthermore, your ultimate cost-saving strategy will depend upon your selected vendor's cost structure. As such, it's more important to choose a vendor based on the particular performance characteristics of your cloud application, than it is to worry about a vendor's hardware.

This calls for an example. We have a transaction that makes 10 disk accesses for each search operation, and each access incurs some very low charge. By cutting the number of accesses in half, the search transaction may not run any faster, but think about the potential operational savings for a transaction that might be executed a million times!

Furthermore, we're faced with a situation in which the vendor pricing structure can change at any time. In order to optimize expenses in response, we must monitor performance preemptively by capturing data from the right measures.

Before we start on performance analysis, I'll summarize how and why we monitor cloud applications.

Effective Monitoring: Public Cloud–Based Applications

Cloud environments are designed to be dynamic and large. So to maintain consistency and reliability we use an automated monitoring system, and every new application must be registered with this system upon startup. This can be implemented from within the application framework or through a JVM agent–based monitoring approach. Either way, we must also ensure that any newly deployed instances are monitored automatically.

Cost-effective cloud applications will leverage elastic scaling, which for any given tier requires knowledge of the load and response time on that tier. If increased load negatively impacts response time, we know to scale up the specific tier. Combining this technique with a baselining approach, we can automatically scale up predictively based on historical load patterns (*Figure 7.7*).

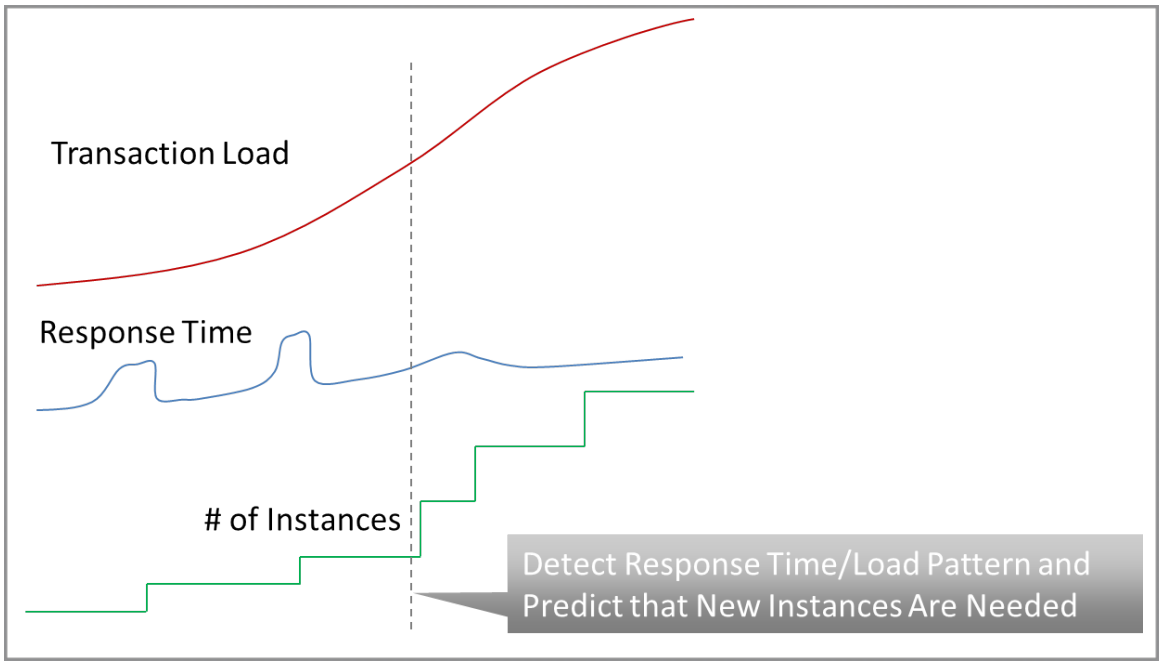


Figure 7.7: Once we have a baseline and load pattern, we can start new instances proactively instead of reactively.

We still need to understand the impact of resource utilization and hardware latencies on our application. In a private cloud we need to maintain the correlation from application to hardware at any given point in order to resolve issues. In a public cloud we do not need this, but still need to know the impact of the cloud infrastructure in our application to make informed deployment and scaling decisions.

By the nature of the cloud, instances are temporary. So our automatic monitoring system must know when to retire instances, while saving any information we might need later for performance analysis. Since the application will have a longer life cycle than the instances it uses, we can aggregate measures such as response time and underlying virtualization metrics from an application point of view for long-term trending.

For example, it's more important when looking over historical data to pinpoint a month-old problem as CPU ready time due to over-utilization than it is to know exactly which instance may have caused the issue. Most likely, the instance in question has long since been retired.

We must also distinguish between data needed for monitoring (longer-term) and that needed for analysis (shorter-term), so that we can store it differently. For instance, a trending analysis might use 10-second or 1-minute interval aggregates; but for detailed diagnostics we'll need the same information, but for every single transaction.

Of course, the real value of a good monitoring setup becomes abundantly clear when resolving any issues, especially during performance analysis.

Performance Analysis and Resolution of Cloud Applications

Detailed performance analysis is a complex process, but the goal is very simple: to identify the root cause of a problem. Toward this end, we proceed in an orderly, step-by-step manner to home in on the fault domain containing the root cause. Since this chapter is specifically about virtualization and cloud computing, we'll discuss how those two domains affect this process. But first let's examine the high-level flow of a root-cause analysis.

We begin with a typical problem, slow application response time, then in step-by-step fashion, we'll narrow down the possibilities:

1. Decide whether the problem affects the entire application, or just a particular transaction or transaction type. If the latter, then our first step is to isolate the transaction type that slows down. This will define the context for further investigation.
2. Identify the problematic tier or tiers. The problem might lie between two tiers (i.e., due to network latency).
3. To isolate the problem further, check whether the environment itself has a negative impact, such as CPU exhaustion, memory constraints. This would include anything that is external to the application itself; for instance garbage-collection suspensions.
4. Finally, isolate the problem to a specific component, method, or service call within the application. From this point, we can determine if the root cause is algorithmic, CPU-centric, a result of excessive VM suspensions, or caused by external bottlenecks, such as I/O, network, or locks (synchronization). This last step is the first step towards a potential solution.

While this flow might be slightly different in your environment, it can be applied to any system, cloud-based included, though with an important caveat: virtual systems suffer the time-keeping problem triggered by VM suspensions, which can cause us to identify the wrong tier as problematic. Thus, whenever we look at response or communication times, we must measure real time (as opposed to apparent time). We can achieve this either via virtualization-aware timers in the VM or by measuring response time outside the VM in a system that is not itself affected (e.g., an appliance).

If inter-tier time is rising, it can be due to an increased number of transactions or an over-utilization of the underlying network infrastructure. (Better pay a friendly visit to the administrator if the latter is the case.)

If instead we've identified a single problematic tier, our first step is to check for excessive VM suspensions. We can do this by looking at the reported steal time and correlating it

with our transactions on a time basis. If the granularity of our measure is sufficiently fine, we can know that a specific transaction, or even method, was suspended. With insufficient granularity we know only that a transaction might have been suspended. For example, if transactions complete in one second but our granularity is measuring 20-second intervals (the default interval of VMware's vCenter), we can't be certain of our results. Finer granularity leads to greater certainty.

Note that in a given virtual system we will always see *some* steal time, but it should not be high. It should be similar in amount to a garbage collection: a few milliseconds per 20-second cycle.

Next we analyze the tier itself for method-level hot spots. Here we face some familiar problems:

- Virtualization-aware timers are more costly than the usual OS timers and impose overhead on detailed measurements that can be counterproductive.
- VM suspensions can create the illusion of slowness even for what should be a fast method, potentially leading our analysis down the wrong path. Therefore, we must differentiate between real time (which includes suspensions) and actual execution time (which does not).
- VM suspensions are nearly never reported at the level of granularity necessary to deal with the two previous points!

This seeming conundrum has two possible solutions:

- We can rely on guest timers alone, which you might think would skew our data since this makes it impossible to use single-transaction diagnostics. However, if we do not experience major VM suspensions (which we can check by looking at the steal-time charts, shown *Figure 7.8)*, the law of big numbers prevents skewing when we look at aggregated data from thousands of transactions.
- We can correlate VM suspensions to a detailed transaction breakdown, much as we do with GC suspensions. By excluding more heavily impacted transactions (more than a couple of milliseconds), our analysis avoids the timing problem.

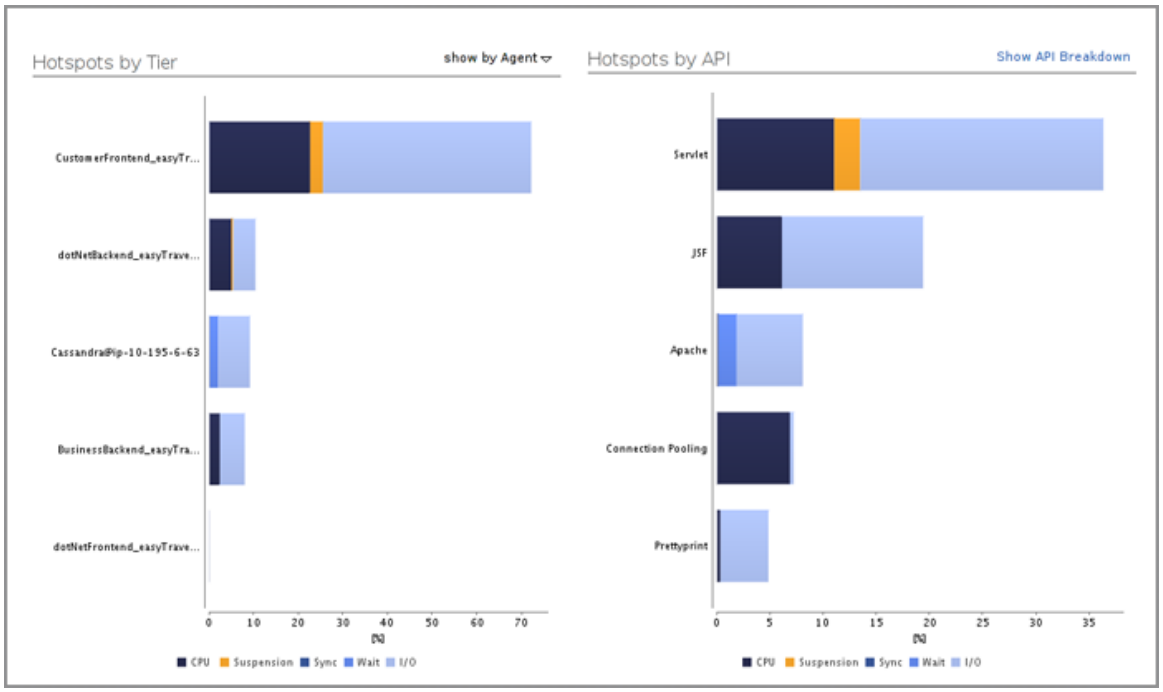


Figure 7.8: This hot spot graphic shows how much the VM suspension contributed to the transaction response time. We can use this to ignore the suspended area in our analysis.

We're finally at the point where we've isolated a particular code area, service call, or method as the likely root cause. Now we figure out why it is slow. The most common cause is too much CPU consumption. Since we're talking about virtual environments, we must ask again if the reported CPU time can be trusted. Surprisingly, the answer is yes, mostly.

Most hypervisors handle steal time in a way that minimizes CPU impact in any of the user processes. This means that average CPU consumption for a transaction measured in CPU time will be fairly accurate. (Remember that this is not true for CPU utilization.) If we see a consistent increase in this measure, we can trust it to mean that the application code is really consuming more CPU time.

The reverse is also true. If the time of our transaction increases in a specific part of the application (as opposed to all over the place) and cannot be attributed to CPU, sync, steal, wait, or garbage-collection time, then it must be attribute to increased latency in the virtualization layer. But we needn't just guess.

We know which methods in our application access the network or the disk. By using virtualization-aware timers for these access I/O points, we know that when these measured times increase, there is either a genuine latency issue or a suspension. Either way, the next step is to look at the virtualization layer for answers and speak with the administrator about solutions.

That's nearly all there is to it. As already mentioned, there is a major difference in goals when doing performance management in private vs. public clouds. There are also technical differences in the performance analysis. That brings us to the next topic.

Analyzing Performance for Cloud-based Applications

In addition to the performance issues we face with any virtualized environment, clouds present a couple of unique problems.

First, a VM might move or be retired at any point. To analyze performance issues after the fact, which is usually the case, we need to reconstruct the flow of a transaction across its VMs, including information on the original physical hosts. We need the physical host not because we need to analyze its utilization per se, but because we need to know about other VMs running on the same host. How many and which other VMs are running on the same host as ours can change at any time, and thus the applications that can impact ours can change any time. Without this knowledge, we might know that we're not getting enough resources, but not why! Root-cause analysis is all about understanding the why.

Here we have the fundamental difference between public and private clouds. In a private cloud I can have the visibility required to analyze the real underlying root cause; in a public cloud I do not!

Public Clouds = Less Visibility

In a public cloud we do not have visibility into the host layer; therefore we cannot make the correlation between host and guest problems. Although other applications (including other companies') can have an impact on ours, we cannot identify these as the root cause. This is also where performance analysis, or rather fixing performance problems, is different from a normal virtual environment or private cloud.

When the application is not the root of a performance issue, and we've confirmed increased latency at the virtualization layer, then we know that a direct fix won't work. Chances are the problem stems from over-utilization or hardware degradation, but we can't know for sure. We have two options:

1. Add a new instance and scale up, under the assumption that the problem is temporary.
2. Recycle the instances by starting a new instance and terminating the old ones afterwards. This more aggressive approach makes the assumption that the new instance will be started on different hardware that is not yet over-utilized or does not degrade. If the assumption is true, then the new instance will execute without problems.

Both actions can be set to run automatically in a cloud environment. Deciding which of the two approaches to take depends on what works best for your application.

With a public cloud, we have not only less visibility into the environment, but also less control. Third-party services, such as load balancers and databases, are part of the package. So it's important to have proper fault-domain identification on a business-transaction level. We can measure calls to cloud and third-party services from within the application. By monitoring our application at this level, we maintain the control we might otherwise have lost to the cloud. Calls to cloud and third party services need to be measured with virtualization-aware timers to allow accurate fault domain isolation (*Figure 7.9*).

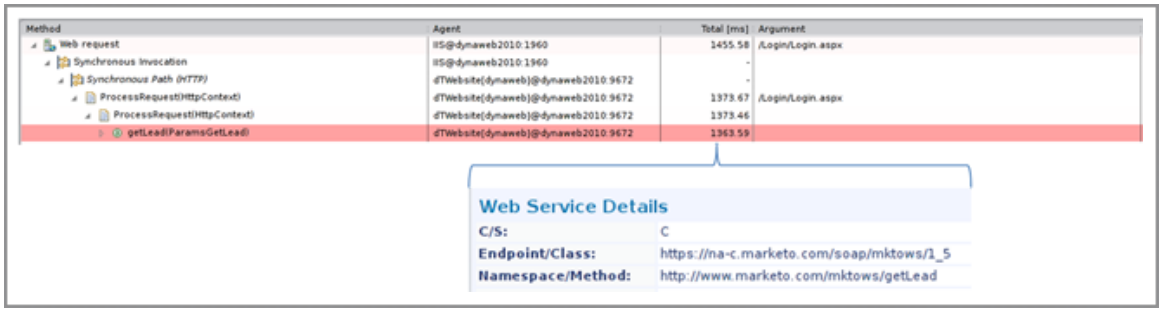


Figure 7.9: Monitoring the response time of service calls is essential in a cloud environment.

Our course of action depends on the service and our contract, and this dependence on the third-party provider is often the major tradeoff when using a public cloud.

Even more important than the technical differences in performance analysis for public and private clouds is the question of why we analyze performance and then what to optimize for!

Optimization Goals and Paradigm Shifts

In a private cloud (or physical datacenter) we optimize first to achieve our application goals (service-level agreements, or SLAs), and next to save resources so that we can delay buying new hardware as long as possible.

In a public cloud this is less of an issue, as we pay for hardware only when we use it. We can even give back what we're not using during off-peak hours to avoid additional charges. Consequently, we needn't optimize resource usage directly, but rather in relation to cost efficiency. Cost in a public cloud depends not only on the number of instances, but also on the number of I/O requests, amount of traffic, number of service calls, and other services. Reducing the number of I/O requests might considerably lower the cost per transaction while not necessarily lowering the number of instances we have in use.

This has important implications for application performance management. Once the application goals (response time, throughput, SLA) are met, we bypass resource efficiency and go directly after cost efficiency!