BEA White Paper

Adaptive Memory Management for Virtualized Java Environments
# Table of Contents

- The Java Virtual Machine and hypervisor virtualization .................................................... 1
- Adaptive JVM technology: BEA LiquidVM ......................................................... 3
- Heap memory management for Java virtualization ...................................................... 3
- Virtual memory and the hypervisor balloon driver ........................................................ 4
- Adaptive memory management for virtualized Java ....................................................... 5
- Summary ................................................................. 7
- About BEA ............................................................... 7
- Join the BEA community ....................................................... 7
The Java Virtual Machine and hypervisor virtualization

The Java Virtual Machine (JVM) is one of the great success stories in computing over the last two decades. It is the JVM that gives Java technology the "write once, run anywhere" character that has made Java ubiquitous in IT, running everything from large, enterprise applications to mobile devices. Now we are seeing the emergence of another virtual machine technology that is changing the way many organizations manage their IT operations: hypervisor-based virtualization.

With hypervisor virtualization, physical server resources (such as CPUs, memory, disk storage, and network interfaces) can be abstracted by a hypervisor or virtual machine monitor (VMM) that uses software to emulate the complete runtime environment of a physical machine. An obvious question about this approach is: How can one combine these two technologies to achieve a fully virtualized environment for enterprise Java applications, and what should that look like?

Currently, a great deal of interest is focused on the potential for virtualizing hardware based on industry-standard x86 architectures. In this area, VMware’s Virtual Infrastructure is by far the most widely adopted hypervisor platform and this is the combination of technologies we will concentrate on in this paper. As we shall see, BEA is leading the industry in providing software that is uniquely optimized for enterprise Java virtualization through its BEA JRockit® JVM technology—a high-performance JVM developed to provide exceptional levels of reliability, manageability, and performance for Intel-based environments.

Given a hypervisor-based virtualization platform such as VMware’s Virtual Infrastructure 3, it would be natural to start by running a traditional OS-based software stack inside the virtual machines as shown in Figure 1:
Here each virtual machine controlled by the hypervisor runs an operating system (Microsoft Windows or Linux) on which the application server and Java applications run normally within a JVM. This is the “virtualized OS” model. This approach simply mirrors the software stack used to run enterprise Java applications on dedicated servers. However, there is an alternative approach, based on a special, hypervisor-enabled version of BEA JRockit called BEA LiquidVM™, which can run inside the VMware virtual machine and directly on top of the hypervisor without a traditional operating system, as illustrated in Figure 2.

In this model, we see the application server (BEA WebLogic Server) and the JVM (BEA LiquidVM) running directly on the hypervisor, making use of a limited set of OS-level functionality that is built into BEA LiquidVM to interface with the hypervisor’s networking, file-system, and memory-management capabilities. This also enables the virtual appliance model, where the entire runtime support for enterprise Java applications (the JVM and the application server) is prepackaged as a “virtual software appliance” that can be stored on a network file system, ready to deploy anywhere on the virtualized infrastructure. In this case, the software appliance is BEA WebLogic Server—Virtual Edition, which is deployed directly into its own virtual machine and which can be preconfigured to run Java Enterprise Edition applications in exactly the same way as BEA WebLogic Server does for non-virtualized environments. BEA WebLogic Server—Virtual Edition instances also benefit from the adaptive memory-management capabilities of BEA LiquidVM.
Adaptive JVM technology: BEA LiquidVM

One key advantage of this approach is that BEA LiquidVM is in complete control of all the memory that is available to the virtual machine and can therefore optimize its use of memory more aggressively than would be possible with a normal OS-based JVM, which potentially has to coexist with other applications. However, equally important is the fact that BEA LiquidVM can hook directly into the hypervisor’s mechanism for alerting the operating environment in its virtual machines to changes in the overall memory pressure on the system—that is, the balance between the total memory available to a virtual machine and the application’s memory requirements at any given time.

Here is why that’s so important: The Java Virtual Machine is a finely tuned instrument, with overall performance depending on many complex, overlapping tradeoffs of memory allocation, garbage collection, dynamic optimization, thread management, and much more. Tuning the JVM for optimal performance of any particular application is an expert task, requiring a deep understanding of both the application and the Java runtime environment. A high-performance JVM like BEA JRockit is constantly self-optimizing for best results, but key memory-management and garbage-collection strategies must be set on startup. For example, an expert administrator will tune a JVM very differently for a system where, for example, memory pressure is high, compared to a situation in which memory is readily available. What makes BEA LiquidVM special is the fact that it is adaptive; it can respond dynamically and automatically to changes in the underlying runtime environment by altering its operating strategies.

Because BEA LiquidVM can use the hypervisor’s balloon driver mechanism (see below) to sense increasing memory pressure as overall system load increases, it can optimize for the changed environment without needing to be paused, reconfigured, and restarted. As we will see, that is the key to the performance advantages of BEA LiquidVM over the virtualized OS model. In the latter, a multitasking OS like Windows or Linux is largely redundant (BEA LiquidVM is able to implement the small subset of OS service required to run a fully featured JVM with a fraction of the code needed for even a minimal OS installation), but worse, the interaction between the hypervisor and the OS, which itself expects to manage available memory, prevents such adaptive, dynamic optimization.

Heap memory management for Java virtualization

For enterprise Java applications, memory is usually the most important, and the most costly, limiting factor on server utilization, and the key challenge is how to manage the Java heap, which typically accounts for as much as 80 percent of the total memory used by an application. In non-virtualized environments it is normal to start the Java Virtual Machine with a relatively large initial amount of memory to minimize the likelihood that the operating system will have to resort to swapping to meet demand, which could have a serious impact on application performance. However, the memory resources available in any environment are ultimately finite and the key question is: How does the system cope when there is competition for memory?

When a non-virtualized system finds itself under memory pressure (that is to say, the system as a whole does not have enough physical memory to support all the applications it needs to run), then the operating system typically swaps memory to and from the disk to enable competing processes to share available memory. In general, the multipurpose operating systems we use today are well-developed in the way they perform this task. However, Java applications offer a particular challenge, because the Java heap region consists of objects that are active in memory within the application. In this case, the operating system has no easy way to determine which areas of memory can safely be swapped out to disk without the risk of substantial performance impact.
In a virtualization environment such as VMware’s Virtual Infrastructure 3, the hypervisor manages virtual machines similarly to how the operating system manages multiple processes in the non-virtualized world. It also uses page-swapping techniques to allow virtual machines to share available memory. This creates an additional layer of complexity in which the hypervisor needs to support a “guest operating system” inside a virtual machine, since there is a very real risk that the hypervisor and the operating system will perform swapping of the same memory in an uncoordinated way that makes the overall system inefficient.

**Virtual memory and the hypervisor balloon driver**

To address this problem, hypervisor-based virtualization environments like VMware’s Virtual Infrastructure 3 use the concept of a “balloon driver,” a device driver that is inserted into the guest operating system, and that the hypervisor uses to signal to the OS that it needs to reduce memory usage. The operating system uses its own heuristics to swap memory out to disk and only if that fails to free sufficient memory will the hypervisor perform additional swapping of its own. While this works well as a general technique, it runs into the same problem noted above when attempting to manage memory for large enterprise Java applications since the OS has little visibility into the active memory contained within the Java heap, which itself is managed by the JVM.

The JVM, which allocates and keeps track of memory for the Java application that runs within it, needs to have a great deal of information about how the application uses memory. BEA JRockit JVM technology is unsurpassed in the way it can optimize memory usage based on the detailed behavior of the application at run time. Its consistently industry-leading performance in running benchmark applications, as well as other breakthrough innovations such as deterministic garbage collection for real-time applications and advanced memory-leak detection and analysis tools, all demonstrate the power of the JVM in this regard.

So, in a virtualized environment, when the hypervisor signals that it needs a virtual machine to reduce its memory requirements, the JVM is best placed to determine how the application can shrink its memory footprint. This is the simple but important insight behind the remarkable performance of BEA LiquidVM in hypervisor environments. The JVM uses its detailed knowledge of the application’s runtime behavior to shrink the number of active pages used by the application and so reduce the size of the Java heap, freeing up memory that can be “given back” to the hypervisor and reducing the need for relatively costly swapping of memory to and from the disk.

Unlike traditional JVMs which have to run on top of a standard OS, BEA LiquidVM is designed to run fully virtualized, in its own virtual machine, without an operating system. There is no need for a multitasking OS since the entire virtual machine is dedicated to running a single Java application. Therefore, BEA LiquidVM contains a minimum subset of OS functionality, specifically designed and written by the BEA JRockit engineering team to enable the JVM to sit directly on top of the hypervisor. This includes an implementation of the VMware balloon driver, which allows BEA LiquidVM to be notified directly by the hypervisor when memory pressure increases. And because BEA LiquidVM runs in its own virtual machine, it has total freedom to optimize the memory available to it. The result is unparalleled efficiency in running enterprise Java applications.
Adaptive memory management for virtualized Java

What does this mean for large-scale enterprise Java deployments? These will nearly always consist of multiple instances of the application, each running in an application server (such as the industry-leading BEA WebLogic Server) inside its own Java virtual machine. To scale up to support larger application workloads, system administrators will deploy additional instances of the application and that means starting extra JVMs, each with a relatively large amount of memory dedicated for its exclusive use.

Generally, this imposes a limit on the number of instances that can be started on any given hardware platform, since the number of JVMs times the initial memory required for each JVM cannot exceed the total memory available. If additional instances beyond this limit are started, the operating system must use swapping to provide the necessary memory to enable all processes to run with equal priority. This inevitably impacts performance and in fact, one quickly finds that the operating system is obliged to spend more and more time swapping memory to and from disk. At this point, the system is “thrashing” and overall system performance will degrade sharply.

However, in virtualized environments with BEA LiquidVM running directly on top of the hypervisor, it is possible to start significantly more application instances than with a standard operating system, per any given set of physical resources. As we have seen, BEA LiquidVM manages memory “adaptively,” so that it can respond to high-memory-pressure situations by shrinking the number of active pages in the application’s Java heap. That means that more memory is available for the hypervisor to start extra application instances—in some cases, as many as twice the number of instances, as illustrated by Figure 3 (BEA LiquidVM versus virtualized OS).

This illustrates the power of BEA LiquidVM adaptive memory management. Both hardware environments are identical and each is running VMware’s ESX Server. However, the left-hand chart shows the results of running virtual machines with BEA LiquidVM and no additional OS layer, while the right-hand chart shows what happens when each virtual machine contains a guest operating system (in this case, Red Hat Enterprise Linux) and a standard BEA JRockit JVM. The system on the right effectively reaches its maximum potential throughput at a total of four virtual machines; however, the system on the left (with BEA LiquidVM) is able to run as many as eight virtual machines, with a maximum potential throughput twice that of the other system.

![Figure 3: BEA LiquidVM versus virtualized Linux.](image_url)
A similar picture emerges from another study, this time using a standard Java server benchmark application to compare performance between two VMware ESX Server installations on identical Intel Xeon servers. The first system (shown in blue) ran BEA LiquidVM 1.1 inside each virtual machine instance, while the second ran Red Hat Enterprise Linux 4.0 plus BEA JRockit R27.3. In a test environment with no physical memory constraints, BEA LiquidVM was able to deliver up to 40 percent higher throughput than the virtualized Linux environment. However, the real advantage of BEA LiquidVM adaptive memory management can be seen in the second test run, where memory pressure increased as additional instances were started. Under these conditions, BEA LiquidVM was able to scale up to eight application server instances running over 500 transactions per second, while additional virtual machines could not even be instantiated using the Linux-based software stack due to physical memory constraints.

Figure 4

BEA LiquidVM versus virtualized Linux (Java server benchmark application).

Under non-constrained environment, BEA LiquidVM delivers up to 40% higher throughput than a virtualized OS.

JVM -Xmx=1600MB, -Xnopt=300

Under memory-constrained environment, BEA LiquidVM delivers much greater efficiency than virtualized OS.

JVM -Xms=-Xmx=1600MB, -Xnopt=300

Intel Xeon 2.0 GHz quad-core dual socket server, 8 GB RAM, VMware ESX 3.0, BEA LiquidVM 1.1, BEA JRockit R27.3, BEA WebLogic Server 9.2 MP2, RHEL 4.0
Each VM allocated 1 vCPU and 2 GB vMem.
Summary

In virtualized environments based on hypervisor technology like VMware's Virtual Infrastructure 3, adaptive memory management within the Java Virtual Machine plays an important role in optimizing the performance of Java applications. This becomes particularly apparent when multiple instances of an application are run within a memory-constrained environment. BEA LiquidVM technology is unique in its ability to respond to changes in memory pressure on the underlying hypervisor infrastructure by changing its heuristics and behavior to match its runtime environment. In virtualized environments running enterprise Java applications, the adaptive memory management of BEA LiquidVM technology can allow up to two times the number of virtual machine instances to be run without any external reconfiguration, resulting in much higher application throughput than is achievable with a standard, OS-based software stack.

About BEA

BEA Systems, Inc. (NASDAQ: BEAS) is a world leader in enterprise infrastructure software. BEA® Enterprise 360°, the industry's most advanced SOA-based offering, is a comprehensive approach to delivering business results that includes technology, professional services, best practices, and world-class partners. Information about how BEA helps customers build a Liquid Enterprise™ that transforms their business can be found at bea.com.

Join the BEA community

At BEA, we understand that developers need different kinds of resources than IT managers. And that architects face different challenges than executives. That’s why we’ve created four unique communities that give you exclusive access to a formidable group of your peers, to a world of shared thinking, and to the kind of meaningful information that can make you more effective and more competitive. To join one or more of the BEA communities, simply register online at bea.com/register.