The Java HotSpot™ Virtual Machine

Technical White Paper

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
1 (800) 786.7638
1.512.434.1511

Please
Recycle

Table of Contents

CHAPTER 1

Introduction and Overview

The Java™ platform has become a mainstream vehicle for software development and deployment. Adoption of the Java platform is growing explosively in many dimensions — from credit cards to mainframes, and from Web page applets to large commercial applications. As a result, the quality, maturity, and performance of Java technology is of critical importance to developers and users everywhere. Sun Microsystems is investing heavily in technologies to "raise the bar" across many processor and operating system platforms, so that software developers can count on their Java technology-based programs running efficiently and reliably, regardless of processor or operating system.

One major reason for the interest in the Java platform is that unlike programs written in traditional languages, Java technology-based programs are distributed in a portable, secure form. In the past, use of a portable distribution format generally incurred a performance penalty during execution. By applying modern dynamic compilation technology, it is possible to alleviate this performance penalty while still maintaining portability across platforms.

The Java HotSpot™ virtual machine (VM) is a key component in maximizing deployment of enterprise applications. It is a core component of Java 2 Platform, Standard Edition (J2SE™) software, supported by leading application vendors and technologies. The Java HotSpot VM supports virtually all aspects of development, deployment, and management of corporate applications, and is used by:

- Integrated development environments (IDEs), including Forte™ for Java™, Community Edition, Borland JBuilder, WebGain VisualCafé, Oracle JDeveloper, Metrowerks CodeWarrior, and the NetBeans™ Open Source Project
- Application server vendors, such as BEA Systems (WebLogic Server) and iPlanet (iPlanet™ Application Server)

With millions of downloads and over two and one-half million developers, the Java 2 technology environment is a proven development and deployment platform in hundreds of thousands of installations.

The Java HotSpot VM is enhanced over previous versions, with better performance and reliability. Overall performance is significantly improved by using a wide variety of techniques that include enhanced garbage collection and thread handling. In addition, the on-the-fly adaptive optimization technology that detects and accelerates performance-critical code has been further refined and tuned.

Ultra-fast thread synchronization offers maximum performance of thread-safe Java technology-based programs. The Java HotSpot VM provides a garbage collector that is not only fast, but also fully "accurate." It more reliably collects unused or free objects. Use of state-of-the-art algorithms dramatically reduces or eliminates user-perceivable garbage collection pauses. Finally, at a source code level the Java HotSpot VM is written in a clean, high-level object-oriented style for maintainability and extensibility — a unified source base enables consistent behavior of applications across all supported platforms.

Two different but compatible Java VMs — the Java HotSpot Client VM and the Java HotSpot Server VM — can be deployed to maximize application performance with either "client" or "server" characteristics.

New in this Release

The Java HotSpot VM contains many new performance and serviceability enhancements in the Java 2 SDK, Standard Edition v1.3.1 release. The main areas of improvement include:

Runtime

- Better fatal error reporting — When a fatal error occurs in the virtual machine, there is improved output, including a mechanism to detect if the crash was in the VM itself, or if it was caused by user-supplied native code external to it.
- Improved support for debugging and profiling — Many features of the Java Virtual Machine Debugger Interface (JVMDI) and Java Virtual Machine Profiler Interface (JVMPI) are now fully supported.
- Unified source base — This release is built from a unified source base, providing consistent VM behavior across all supported platforms.

Garbage Collection

- Larger heaps — Garbage collectors have been changed so that they can use the complete address space available in a 32-bit systems. This provides access up to 4-Gbyte heaps. Note that not all operating environments support heaps this large. The Solaris™ Operating Environment supports 4-Gbyte heaps.
- Better soft reference policy — The policy for soft reference pointers has been modified to model the behavior of the Classic VM.
- Enhanced operation — The garbage collectors have been tuned to support large applications and the UltraSPARC™ III platform.

Java HotSpot Client VM

- Unified source base — While no changes have been made to the client compiler, the source base has been unified across all platforms. This assures consistent VM behavior across all supported platforms.

Java HotSpot Server VM Optimizations

- Range check elimination — The Java programming language specification requires array bounds checking to be performed with each array access. An index bounds check can be eliminated when the compiler can prove that an index used for an array access is within bounds.
- Loop unrolling — The Server VM now features loop unrolling, a standard compiler optimization that enables faster loop execution. Loop unrolling increases the loop body size while simultaneously decreasing the number of iterations. Loop unrolling also increases the effectiveness of other optimizations.
- Instruction scheduling — Machine instructions generated by the compiler's code generator do not necessarily appear in optimal order for a particular hardware platform. Instruction scheduling is a compiler optimization that rearranges the generated machine instructions such that the execution speed is improved. This is an UltraSPARC III optimization.
- Object-oriented optimizations for the Java reflection API — The compiler is aware of important library functions, resulting in better performance when generating code for them.

CHAPTER 2

The Java HotSpot VM Architecture

The Java HotSpot virtual machine is Sun Microsystems' VM for the Java platform as well as the technology base for all future Java virtual machines. By using many advanced techniques, it delivers optimal performance for Java technology-based applications.

The Java HotSpot virtual machine architecture is the culmination of many years of basic research in the Smalltalk and Self programming languages. Much of this research was conducted at Sun Microsystems Labs. It incorporates a state-of-the-art memory model, garbage collector, and adaptive optimizer, and is written in an extremely high-level, object-oriented style. The core VM is a complete Java virtual machine implementation using an interpreter for the execution of programs. It features:

- Uniform object model
- Interpreted, compiled, and native frames all using the same stack
- Preemptive multithreading based on native threads
- Accurate generational and compacting garbage collection
- Ultra-fast thread synchronization
- Dynamic deoptimization and aggressive optimizations
- System-specific runtime routines generated at VM startup time
- Compiler interface supporting parallel compilations
- Runtime profiling that focuses compilation effort only on "hot" methods

In the Java 2 SDK, Standard Edition v1.3.1 release there are two flavors of the VM — a client-side offering, and a VM tuned for server applications. These two solutions share the Java HotSpot runtime environment code base, but use different compilers suited to the distinctly different performance characteristics of clients and servers. These differences include the compilation inlining policy and heap defaults.

The Java 2 SDK, Standard Edition v1.3.1 contains both of the aforementioned systems in the distribution, and developers choose which system they want by specifying `-client or -server`.

The client and the server VMs are similar, except that the server VM has been specially tuned to maximize peak operating speed. It is intended for executing long-running server applications, for which having the fastest possible operating speed is generally more important than having a fast startup time or smaller runtime memory footprint.

The client VM compiler serves as an upgrade for both the Classic VM and the just-in-time (JIT) compilers used by previous versions of the Java SDK. The client VM offers improved runtime performance for applications and applets. The Java HotSpot Client VM has been specially tuned to reduce application startup time and memory footprint, making it particularly well-suited for client environments. In general the client system is better for GUIs.

The client VM compiler doesn't try to execute many of the more complex optimizations performed by the compiler in the server VM, but in exchange requires less time to analyze and compile a piece of code. This means the client VM starts up faster and requires a smaller memory footprint.

The server VM contains an advanced adaptive compiler that supports many of the same types of optimizations performed by optimizing C++ compilers, as well as some optimizations that can't be done by traditional compilers, such as aggressive inlining across virtual method invocations. This is a competitive and performance advantage over static compilers. Adaptive optimization technology is very flexible in its approach, and typically outperforms even advanced static analysis and compilation techniques.

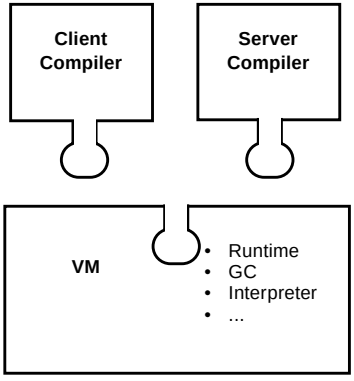

Figure 1: The Java HotSpot Client VM on the left and the Java HotSpot Server VM on the right use a different compiler, but otherwise interface to the same virtual machine, using the same garbage collection (GC) routine, interpreter, thread and lock subsystems, and so on.

Both solutions deliver extremely reliable, secure, and maintainable environments to meet the demands of today's enterprise customers.

Memory Model

Handleless Objects

In previous versions of the Java virtual machine, such as the Classic VM, indirect handles are used to represent object references. While this makes relocating objects easier during garbage collection, it represents a significant performance bottleneck because accesses to the instance variables of Java programming language objects require two levels of indirection.

In the Java HotSpot VM, no handles are used by Java code. Object references are implemented as direct pointers. This provides C-speed access to instance variables. The garbage collector is then responsible for finding and updating all references to an object in place when the object is relocated during memory reclamation.

Two-Word Object Headers

The Java HotSpot VM uses a two machine-word object header as opposed to three words, as in the Classic VM. Since the average Java object size is small, this has a significant impact on space consumption — approximately an eight percent savings in heap size for typical applications. The first header word contains information such as the identity hash code and garbage collector status information. The second is a reference to the object's class. Only arrays have a third header field, for the array size.

Reflective Data Represented as Objects

Classes, methods, and other internal reflective data are represented directly as objects on the heap (although those objects may not be directly accessible to Java technology-based programs). This not only simplifies the VM internal object model, but also allows classes to be collected by the same garbage collector used for other Java programming language objects.

Native Thread Support, Including Preemption and Multiprocessing

Per-thread method activation stacks are represented using the host operating system's stack and thread model. Both Java programming language methods and native methods share the same stack, allowing fast calls between the C and Java programming languages. Fully preemptive Java programming language threads are supported using the host operating system's thread scheduling mechanism.

A major advantage of using native OS threads and scheduling is the ability to transparently take advantage of native OS multiprocessing support. Because the Java HotSpot VM is designed to resistant to race conditions caused by preemption and/or multiprocessing while executing Java programming language code, the Java programming language threads will automatically take advantage of whatever scheduling and processor allocation policies the native OS provides.

Garbage Collection

Background

A major attraction of the Java programming language for programmers is that it is the first mainstream programming language to provide built-in automatic memory management, or garbage collection. In traditional languages, dynamic memory allocation is done using an explicit allocate/free model. In practice, this turns out to be not only one of the largest sources of memory leaks, program bugs, and crashes in programs written in traditional languages, but also a performance bottleneck and major impediment to modular, reusable code. (Determining free points across module boundaries is often nearly impossible without explicit and hard-to-understand cooperation between modules.) In the Java programming language, garbage collection is also an important part of the “safe” execution semantics required to support the security model.

A garbage collector automatically handles “freeing” of unused object memory behind the scenes by only reclaiming an object when it can “prove” that the object is no longer accessible to the running program. Automation of this process completely eliminates the memory leaks caused by freeing too little, as well as the program crashes and hard-to-find reference bugs caused by freeing too much.

Traditionally, garbage collection has been considered an inefficient process that impeded performance, relative to an explicit-free model. In fact, modern garbage collection technology has improved to the extent that its overall performance is actually substantially better than that provided by explicit freeing of objects.

The Java HotSpot Garbage Collector

The Java HotSpot VM contains an advanced garbage collector. In addition to including the state-of-the-art features described below, it takes maximum advantage of the clean, object-oriented design to provide a high-level garbage collection framework that can easily be instrumented, experimented with, or extended to use new collection algorithms.

Overall, the combination of techniques used is better, both for applications where the highest possible performance is needed and for long-running applications, where memory leaks and memory inaccessibility due to fragmentation are highly undesirable. The Java HotSpot VM provides not only state-of-the-art garbage collector performance, but also full memory reclamation, while eliminating memory fragmentation.

Accuracy

The Java HotSpot garbage collector is a fully accurate collector. In contrast, many other garbage collectors are conservative or partially accurate. While conservative garbage collection can be attractive because it can be easily added to a system without garbage collection support, it has certain drawbacks. In general, conservative garbage collectors are prone to memory leaks, disallow object migration, and can cause heap fragmentation.

A conservative garbage collector does not know for sure where all object references are located. As a result, it must be conservative by assuming that memory words that appear to refer to an object are in fact object references. This means that it can make certain kinds of mistakes, such as confusing an integer for an object pointer. Memory cells that look like a pointer are regarded as a pointer — and garbage collection becomes “inaccurate.” This has several negative impacts.

First, when such mistakes are made — which in practice is not very often — memory leaks can occur unpredictably in ways that are virtually impossible for application programmers to reproduce or debug. Second, since it might have made a mistake, a conservative collector must either use handles to refer indirectly to objects — decreasing performance — or avoid relocating objects, because relocating handleless objects requires updating all the references to the object. This cannot be done if the collector does not know for sure that an apparent reference is a real reference. The inability to relocate objects causes object memory fragmentation and, more importantly, prevents use of the advanced generational copying collection algorithms described below.

Because the Java HotSpot garbage collector is fully accurate, it can make several strong design guarantees that a conservative collector cannot make:

- All inaccessible object memory can be reclaimed reliably.
- All objects can be relocated, allowing object memory compaction, which eliminates object memory fragmentation and increases memory locality.

An accurate garbage collection mechanism avoids accidental memory leaks, enables object migration, and provides for full heap compaction. The garbage collection mechanism in the Java Hotspot VM scales well to very large heaps.

Generational Copying Collection

The Java HotSpot VM employs a state-of-the-art generational copying collector that provides two major benefits:

- Major increases in both allocation speed and overall garbage collection efficiency for most programs, compared to non-generational collectors
- A corresponding decrease in the frequency and duration of user-perceivable garbage collection “pauses”

A generational collector takes advantage of the fact that in most programs the vast majority of objects (often greater than 95%) are very short-lived (for example, they are used as temporary data structures). By segregating newly-created objects into an object “nursery,” a generational collector can accomplish several things. First, because new objects are allocated contiguously in stack-like fashion in the object nursery, allocation becomes extremely fast (since it involves merely updating a single pointer and performing a single check for nursery overflow). Secondly, by the time the nursery overflows, most of the objects in the nursery are already “dead,” allowing the garbage collector to simply move the few surviving objects elsewhere and avoid doing any reclamation work for dead objects in the nursery.

Mark-Compact “Old Object” Collector

Although the generational copying collector collects most dead objects efficiently, longer-lived objects still accumulate in the “old object” memory area. Occasionally, based on low-memory conditions or programmatic requests, an old object garbage collection must be performed. The Java HotSpot VM can use a standard mark-compact collection algorithm, which traverses the entire graph of live objects from its “roots,” and then sweeps through memory, compacting away the gaps left by dead objects. By compacting gaps in the heap rather than collecting them into a freelist, memory fragmentation is eliminated, and old object allocation is streamlined by eliminating freelist searching.

Incremental “Pauseless” Garbage Collector

The mark-compact collector does not eliminate all user-perceivable pauses. User-perceived garbage collector pauses occur when “old” objects (objects that have “lived” for a while in machine terms) need to be garbage collected, and these pauses are proportional to the amount of live object data that exists. This means that the pauses can become arbitrarily large as more data is manipulated, which is a very undesirable property for server applications, animation, or other soft real-time applications.

The Java HotSpot VM provides an alternative old-space garbage collector to solve this problem. This collector is fully incremental, eliminating user-detectable garbage collection pauses. This incremental collector scales smoothly, providing relatively constant pause times even when extremely large object data sets are being manipulated. This provides excellent behavior for:

- Server applications, especially high-availability applications
- Applications that manipulate very large “live” object data sets
- Applications where all user-noticeable pauses are undesirable, such as games, animation, or other highly interactive applications

The pauseless collector works by using an incremental old space collection scheme referred to academically as the “train” algorithm. This algorithm breaks up old space collection pauses into many tiny pauses (typically less than ten milliseconds) that can be spread out over time so that the program virtually never appears to a user to pause. Since the train algorithm is not a hard real-time algorithm, it cannot guarantee an upper limit on pause times; however, in practice, much larger pauses are extremely rare, and are not caused directly by large data sets.

The pauseless collector also has the highly desirable side-benefit of producing improved memory locality. This happens because the algorithm works by attempting to relocate groups of tightly-coupled objects into regions of adjacent memory, providing excellent paging and cache locality properties for those objects. This can also benefit highly multithreaded applications that operate on distinct sets of object data.

Ultra-Fast Thread Synchronization

The Java programming language allows the use of multiple, concurrent paths of program execution — threads. The Java programming language provides language-level thread synchronization, which makes it easy to express multithreaded programs with fine-grained locking. Previous synchronization implementations were highly inefficient relative to other microoperations in the Java programming language, making use of fine-grain synchronization a major performance bottleneck.

The Java HotSpot VM incorporates a breakthrough in thread synchronization implementation that boosts synchronization performance by a large factor. As a result, synchronization performance becomes so fast that it is not a significant performance issue for the vast majority of real-world programs.

In addition to the space benefits mentioned in the Memory Model section of this paper, the synchronization mechanism provides its performance benefits by providing ultra-fast, constant-time performance for all uncontended synchronizations, which dynamically comprise the great majority of synchronizations.

The Java HotSpot VM provides a leaner, speedier thread-handling capability that is designed to scale readily for use in large, shared-memory multiprocessor servers.

CHAPTER 3

The Java HotSpot Compilers

Most attempts to accelerate Java programming language performance have focused on applying compilation techniques developed for traditional languages. Just-in-time (JIT) compilers are essentially fast traditional compilers that translate the Java technology bytecodes into native machine code on-the-fly. A JIT complier runs on the end-user's machine and actually executes the bytecodes, compiling each method the first time it is executed.

There are several problems with JIT compilation. First, because the compiler runs on the execution machine in “user time,” it is severely constrained in terms of compile speed: if it is not very fast, then the user will perceive a significant delay in the startup of a program or part of a program. This entails a trade-off that makes it far more difficult to perform advanced optimizations, which usually slows down compilation performance significantly.

Secondly, even if a JIT compiler had time to perform full optimization, such optimizations are less effective for the Java programming language than for traditional languages like C and C++. There are a number of reasons for this effect:

- The Java programming language is dynamically “safe,” meaning that it is ensured that programs do not violate the language semantics or directly access unstructured memory. This means dynamic type-tests must frequently be performed (when casting and when storing into object arrays).
- The Java programming language allocates all objects on the “heap,” whereas in C++ many objects are stack allocated. This means that object allocation rates are much higher for the Java programming language than for C++. In addition, because the Java programming language is garbage-collected, it has very different types of memory allocation overhead (including potentially scavenging and write-barrier overhead) than C++.

- In the Java programming language, most method invocations are “virtual” (potentially polymorphic), and are more frequently used than in C++. This means not only that method invocation performance is more dominant, but also that static compiler optimizations (especially global optimizations like inlining) are much harder to perform for method invocations. Most traditional optimizations are most effective between calls, and the decreased distance between calls in the Java programming language can significantly reduce the effectiveness of such optimizations, since they have smaller sections of code to work with.
- Java technology-based programs can change on-the-fly due to the powerful ability to perform dynamic loading of classes. This makes it far more difficult to perform many types of global optimization. The compiler must not only be able to detect when these optimizations become invalid due to dynamic loading, but also be able to undo and/or redo those optimizations during program execution, even if they involve active methods on the stack. This must be done without compromising or impacting Java technology-based program execution semantics in any way.

As a result, any attempt to achieve fundamental advances in Java programming language performance must provide non-traditional answers to these performance issues, rather than blindly applying traditional compiler techniques.

The Java HotSpot VM architecture addresses the Java programming language performance issues described above by using adaptive optimization technology.

Hot Spot Detection

Adaptive optimization solves the problems of JIT compilation by taking advantage of an interesting property of most programs. Virtually all programs spend the vast majority of their time executing a small minority of their code. Rather than compiling method-by-method, just in time, the Java HotSpot VM runs the program immediately using an interpreter and analyzes the code as it runs to detect the critical “hot spots” in the program. It then focuses the attention of a global native-code optimizer on the hot spots. By avoiding compilation of infrequently executed code (most of the program), the Java HotSpot compiler can devote much more attention to the performance-critical parts of the program, without necessarily increasing the overall compilation time. This hot-spot monitoring is continued dynamically as the program runs, so that it literally adapts its performance on-the-fly to the needs of the user.

A subtle but important benefit of this approach is that by delaying compilation until after the code has already been executed for a while (“a while” in machine time, not user time), information can be gathered on the way the code is used, and then used to perform more intelligent optimization. Also, the memory footprint is decreased. In addition to collecting information on hot spots in the program, other types of information are gathered, such as data on caller-callee relationships for “virtual” method invocations.

Method Inlining

As mentioned in the Background section of this paper, the frequency of virtual method invocations in the Java programming language is an important optimization bottleneck. Once the Java HotSpot adaptive optimizer has gathered information during execution about program hot spots, it not only compiles them into native code, but also performs extensive method inlining on that code.

Inlining has important benefits. It dramatically reduces the dynamic frequency of method invocations, which saves the time needed to perform those method invocations. But even more importantly, inlining produces much larger blocks of code for the optimizer to work on, significantly increasing the effectiveness of traditional compiler optimizations, and thus overcoming a major obstacle to increased Java programming language performance.

Inlining is synergistic with other code optimizations because it makes them more effective. As the Java HotSpot compiler matures, the ability to operate on large, inlined blocks of code will open the door to a host of even more advanced optimizations in the future.

Dynamic Deoptimization

Although inlining is an important optimization, it has traditionally been very difficult to perform for dynamic object-oriented languages like the Java programming language. Furthermore, while detecting hot spots and inlining the methods they invoke is difficult enough, it is still not sufficient to provide full Java programming language semantics. This is because programs written in the Java programming language cannot only change the patterns of method invocation on-the-fly, but can also dynamically load new Java code into a running program.

Inlining is based on a form of global analysis. Dynamic loading significantly complicates inlining because it changes the global relationships in a program. A new Java class may contain new methods that need to be inlined in the appropriate places. So the Java HotSpot VM must be able to dynamically deoptimize (and then reoptimize if necessary) previously optimized hot spots, even during the execution of the code for the hot spot. Without this capability, general inlining cannot be safely performed on Java technology-based programs.

Java HotSpot Client Compiler

The client compiler is tuned for the performance profile of typical client applications. The Java HotSpot Client Compiler is a simple and fast two-phased compiler. In the first phase, a platform-independent front end constructs an intermediate representation (IR) from the bytecodes. In the second phase, the platform-specific background generates machine code from the IR. Emphasis is placed on extracting and preserving as much information as possible from the bytecode level (for example, locality information, initial control flow graph), which directly translates into reduced compilation time. Note that the client VM does only minimal inlining and no deoptimization.

Java HotSpot Server Compiler

The server compiler is tuned for the performance profile of typical server applications. The Java HotSpot Server Compiler is a high-end fully-optimizing compiler. It uses an advanced static single assignment (SSA)-based IR for optimizations. The optimizer performs all the classic optimizations, including dead code elimination, loop invariant hoisting, common subexpression elimination, and constant propagation. It also features optimizations more specific to Java technology, such as null-check and range-check elimination. The register allocator is a global graph coloring allocator and makes full use of large register sets commonly found in RISC microprocessors. The compiler is highly portable, relying on a machine description file to describe all aspects of the target hardware. While the compiler is slow by JIT standards, it is still much faster than conventional optimizing compilers. And the improved code quality “pays back” the compile time by reducing execution times of compiled code. The server compiler performs full inlining and full deoptimization.

CHAPTER 4

Impact on Software Reusability

A primary benefit of object-oriented programming is that it can increase development productivity by providing powerful language mechanisms for software reuse. In practice, however, such reusability is rarely attained. Extensive use of these mechanisms can significantly reduce performance, which leads programmers to use them sparingly. A surprising side-effect of the Java HotSpot technology is that it significantly reduces this performance cost. Sun believes this will have a major impact on how object-oriented software is developed, by enabling companies for the first time to take full advantage of object-oriented reusability mechanisms, without compromising the performance of their software.

Examples of this effect are easy to come by. A survey of programmers using the Java programming language will quickly reveal that many avoid using fully virtual methods (and also write bigger methods) because they believe that every virtual method invocation entails a significant performance penalty. Ubiquitous, fine-grain use of virtual methods, such as methods that are not “static” or “final” in the Java programming language, is extremely important to the construction of highly reusable classes because each such method acts as a “hook” that allows new subclasses to modify the behavior of the superclass.

Because the Java HotSpot VM can automatically inline the vast majority of virtual method invocations, this performance penalty is dramatically reduced, and in many cases eliminated altogether.

It is hard to overstate the importance of this effect. It has the potential to change fundamentally the way that object-oriented code is written, since it significantly changes the performance trade-offs of using important reusability mechanisms. In addition, it has become clear that as object-oriented programming matures, there is a trend towards both finer-grain objects and finer-grain methods. Both these trends point strongly to increases in the frequency of virtual method invocations in the coding styles of the future. As these higher-level coding styles become prevalent, the advantages of Java HotSpot technology will become even more pronounced.

CHAPTER 5

Summary

The Java HotSpot VM delivers optimal performance for Java applications, delivering advanced optimization, garbage collection, and thread synchronization capabilities. The Java HotSpot VM provides separate compilers for client and server environments so that applications can be optimized according to their target deployment environments.

With the Java HotSpot VM, client applications start up faster and require a smaller memory footprint, while server applications can achieve better sustained performance over the long run. Both solutions deliver an extremely reliable, secure, and maintainable environment to meet the demands of today's enterprise customers.

CHAPTER 6

Availability

The Java HotSpot VM is included in the Java 2 Platform, Standard Edition v1.3.1. It is available at java.sun.com/products/hotspot for the following environments:

- Solaris Operating Environment (SPARC™ Platform Edition)
- Solaris Operating Environment (Intel Architecture Edition)
- Linux operating system for the Intel Architecture platform
- Microsoft Windows 95, 98, 2000, and NT 4.0 for the Intel Architecture platform

In addition, Java HotSpot technology is also available from:

- Hewlett-Packard, which licenses the Java 2 Platform, Standard Edition and includes Java HotSpot technology as part of their HP-UX OS for HP 9000 PA-RISC systems
- Apple, which ships the Java HotSpot Client VM v1.3.1 as part of their Mac OS X

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303

1 (800) 786.7638
1.512.434.1511

http://java.sun.com/products/hotspot

May 2001