The Java HotSpot Performance Engine Architecture

Table of Contents

Introduction and Overview
Java HotSpot VM -- Built on a Solid Foundation
The Java HotSpot VM Architecture
Overview
Memory Model
Garbage Collection
Ultra-Fast Thread Synchronization
64-bit Architecture
The Java HotSpot Compilers
Overview
Hot Spot Detection
Method Inlining
Dynamic Deoptimization
Java HotSpot Client Compiler
Java HotSpot Server Compiler
Compiler Optimizations
Advanced Features of the Java HotSpot VM
Scalability
Performance
Reliability, Availability and Serviceability
Impact on Software Reusability
Overview
Summary
Availability
Resources
Back to Top

Chapter 1. Introduction and Overview

The Java HotSpot™ virtual machine implementation (Java HotSpot™ VM) is Sun Microsystems, Inc.'s high-performance VM for the Java platform. Java HotSpot technology provides the foundation for the Java SE platform, the premier solution for rapidly developing and deploying business-critical desktop and enterprise applications. Java SE technology is available for the Solaris Operating Environment (OE), Linux, and Microsoft Windows, as well as other platforms through Java technology licensees.

The Java platform has become a mainstream vehicle for software development and deployment. With millions of developers and users, the Java platform is growing explosively in many dimensions: from credit cards to wireless devices, desktops to mainframes. It is the underlying foundation for deploying Web page applets, Web services, large commercial applications, and more.

The Java HotSpot VM builds upon Java technology's cross-platform support and robust security model with new features and capabilities for scalability, quality, and performance. In addition to new features, this version is upward compatible with previous releases.

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 the Sun Java Studio Tools, the NetBeans open source project, IBM's Eclipse, IntelliJ IDEA, Oracle JDeveloper, and others.

Application server vendors, such as the Sun Java System Application Server, BEA Systems' WebLogic software, IBM's WebSphere software, Apple Computer, Inc.'s WebObjects software, and others.

Sun Microsystems, Inc. is also driving performance improvements through the use of various industry standard and internally developed benchmarks. These improvements are applicable to both the client- and server-side Java VM environments.

The Java Standard Edition Platform contains two implementations of the Java VM:

Java HotSpot Client VM, which is tuned for best performance when running applications in a client environment by reducing application start-up time and memory footprint.

Java HotSpot Server VM, which is designed for maximum program execution speed for applications running in a server environment.

Java HotSpot VM -- Built on a Solid Foundation

The Java HotSpot VM builds on a strong foundation of features and capabilities. An architecture supporting dynamic, object-oriented optimizations enables world-class performance. Multithreading support throughout the VM yields high scalability on even the largest computing systems available today. Advanced Reliability, Availability and Serviceability (RAS) features provide enterprise-grade reliability while enabling rapid development, introspection, and management.

Chapter 2. The Java HotSpot VM Architecture

Overview

The Java HotSpot Virtual Machine is Sun's VM for the Java platform. It delivers the optimal performance for Java applications using many advanced techniques, incorporating a state-of-the-art memory model, garbage collector, and adaptive optimizer. It is written in a high-level, object-oriented style, and features:

Uniform object model

Interpreted, compiled, and native frames all use the same stack

Preemptive multithreading based on native threads

Accurate generational and compacting garbage collection

Ultra-fast thread synchronization

Dynamic deoptimization and aggressive compiler optimizations

System-specific runtime routines generated at VM startup time

Compiler interface supporting parallel compilations

Run-time profiling focuses compilation effort only on "hot" methods

The JDK includes 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 that are suited to the distinctly unique performance characteristics of clients and servers. These differences include the compilation inlining policy and heap defaults.

The JDK contains both of the these systems in the distribution, so developers can choose which system they want by specifying `-client` or `-server`.

Although the Server and the Client VMs are similar, the Server VM has been specially tuned to maximize peak operating speed. It is intended for executing long-running server applications, which need the fastest possible operating speed more than a fast start-up 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 JDK. The Client VM offers improved run time performance for applications and applets. The Java HotSpot Client VM has been specially tuned to reduce application start-up 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 does not try to execute many of the more complex optimizations performed by the compiler in the Server VM, but in exchange, it requires less time to analyze and compile a piece of code. This means the Client VM can start 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 cannot 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.

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. When an object is relocated during memory reclamation, the garbage collector is responsible for finding and updating all references to the object in place.

Two-Word Object Headers

The Java HotSpot VM uses a two machine-word object header, as opposed to three words in the Classic VM. Since the average Java object size is small, this has a significant impact on space consumption -- saving approximately eight percent in heap size for typical applications. The first header word contains information such as the identity hash code and GC 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 are 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 take advantage of native OS multiprocessing support transparently. Because the Java HotSpot VM is designed to be insensitive 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.

Back to Top

Garbage Collection

The generational nature of the Java HotSpot VM's memory system provides the flexibility to use specific garbage collection algorithms suited to the needs of a diverse set of applications. The Java HotSpot VM supports several different garbage collection algorithms designed to serve different pause time and throughput requirements.

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 (GC). In traditional languages, dynamic memory is allocated using an explicit allocate/free model. In practice, this turns out to be not only a major source of memory leaks, program bugs, and crashes in programs written in traditional languages, but also a performance bottleneck and a major impediment to modular, reusable code. (Determining free points across module boundaries is 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 reclaiming an object only when it can prove that the object is no longer accessible to the running program. Automation of this process completely eliminates not only the memory leaks caused by freeing too little, but also 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, with modern garbage collection technology, performance has improved so much that overall performance is actually substantially better than that provided by explicit freeing of objects.

The Java HotSpot Garbage Collector

In addition to including the state-of-the-art features described below, the memory system is designed as a clean, object-oriented framework that can easily be instrumented, experimented with, or extended to use new garbage collection algorithms.

The major features of the Java HotSpot garbage collector are presented below. Overall, these capabilities are well-suited 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.

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 is very easy to add 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 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 GC 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 references to the objects. This cannot be done if the collector does not know for sure whether 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 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 GC 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, which provides two major benefits:

Increased allocation speed and overall garbage collection efficiency for most programs, compared to nongenerational collectors

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 percent) 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 merely involves 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.

Parallel Young Generation Collector

The single-threaded copying collector described above, while suitable for many deployments, could become a bottleneck to scaling in an application that is otherwise parallelized to take advantage of multiple processors. To take full advantage of all available CPUs on a multiprocessor machine, the Java HotSpot VM offers an optional multithreaded collector for the young generation, in which the tracing and copying of live objects is accomplished by multiple threads working in parallel. The implementation has been carefully tuned to balance the collection work between all available processors, allowing the collector to scale up to large numbers of processors. This reduces the pause times for collecting young space and maximizes garbage collection throughput. The parallel collector has been tested with systems containing more than 100 CPU's and 0.5 terabytes of heap. The parallel young generation collector is the default garbage collection algorithm used with the Server VM.

When moving objects, the parallel collector tries to keep related objects together, resulting in improved memory locality and cache utilization, and leading to improved mutator performance. This is accomplished by copying objects in *depth first order*.

The parallel collector also uses available memory more optimally. It does not need to keep a portion of the old object space in reserve to guarantee space for copying all live objects. Instead, it uses a novel technique to speculatively attempt to copy objects. If old object space is scarce this technique allows the collector to switch smoothly to compacting the heap without the need for holding any space in reserve. This results in better utilization of the available heap space.

Finally, the parallel collector is able to dynamically adjust its tunable parameters in response to the application's heap allocation behavior, leading to improved garbage collection performance over a wide range of applications and environments. This means less hand-tuning work for customers. This capability was first introduced with the parallel collector and is now available for many of the other garbage collection algorithms.

In comparison with the default single-threaded collector, the break-even point for the parallel collector appears to be somewhere between two and four CPUs, depending on the platform and the application. This is expected to further improve in future releases.

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

by default uses a standard mark-compact collection algorithm, which traverses the entire graph of live objects from its *roots*, 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.

Mostly Concurrent Mark-Sweep Collector

For applications that require large heaps, collection pauses induced by the default old generation mark-compact collector can often cause disruptions, as application threads are paused for a period that is proportional to the size of the heap. The Java HotSpot VM has implemented an optional concurrent collector for the old object space that can take advantage of spare processor cycles (or spare processors) to collect large heaps while pausing the application threads for very short periods. This is accomplished by doing the bulk of the tracing and sweeping work while the application threads are executing. In some cases, there may be a small decline in peak application throughput as some processor cycles are devoted to concurrent collection activity; however, both average-and worst-case garbage collection pause times are often reduced by one or two orders of magnitude, allowing much smoother application response without the burstiness sometimes seen when the default synchronous mark-compact algorithm operates on large heaps.

Parallel Old Generation Collector

The current version of the Java HotSpot VM introduces a parallel mark-compact collector for the old generation designed to improve scalability for applications with large heaps. Where the concurrent mark-sweep collector focuses on decreasing pause times, the parallel old collector focuses on increasing throughput by using many threads simultaneously to collect the old generation during stop-the-world pauses. The parallel old collector uses many novel techniques and data structures internally to achieve high scalability while retaining the benefits of exact garbage collection and with a minimum of bookkeeping overhead during collection cycles.

For More Information

For more information on the garbage collection algorithms supported in the Java HotSpot VM, please refer to the memory management white paper.

Back to Top

Ultra-Fast Thread Synchronization

The Java programming language allows for 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 such as that in the Classic VM were highly inefficient relative to other micro-operations in the Java programming language, making use of fine-grain synchronization a major performance bottleneck.

The Java HotSpot VM incorporates leading-edge techniques for both uncontended and contended synchronization operations which boost synchronization performance by a large factor. Uncontended synchronization operations, which dynamically comprise the majority of synchronizations, are implemented with ultra-fast, constant-time techniques. With the latest optimizations, in the best case these operations are essentially free of cost even on multiprocessor computers. Contended synchronization operations use advanced adaptive spinning techniques to improve throughput even for applications with significant amounts of lock contention. As a result, synchronization performance becomes so fast that it is not a significant performance issue for the vast majority of real-world programs.

64-bit Architecture

Early releases of the Java HotSpot VM were limited to addressing four gigabytes of memory -- even on 64-bit operating systems such as the Solaris OE. While four gigabytes is a lot for a desktop system, modern servers can contain far more memory. For example, the Sun Fire E25K server supports up to 1.15 terabytes of memory per domain. With a 64-bit JVM, Java technology-based applications can now utilize the full memory of such a system.

There are several classes of applications where using 64-bit addressing can be useful. For example, those that store very large data sets in memory. Applications can now avoid the overhead of paging data from disk or extracting it from a RDBMS. This can lead to dramatic performance improvements in applications of this type.

The Java HotSpot VM is now 64-bit safe, and the Server VM includes support for both 32-bit and 64-bit operations. Users can select either 32-bit or 64-bit operation by using commandline flags `-d32` or `-d64` respectively. Users of the Java Native Interface will need to recompile their code to run it on the 64-bit VM.

Object Packing

Object packing functionality has been added to minimize the wasted space between data types of different sizes. This is primarily a benefit in 64-bit environments, but offers a small advantage even in 32-bit VMs.

For example:

```
public class Button {
  char shape;
  String label;
  int xposition;
  int yposition;
  char color;
  int joe;
  object mike;
  char armed;
}
```

This would waste space between: `color` and `joe` (three bytes to pad to an `int` boundary) `joe` and `mike` (four bytes on a 64-bit VM to pad to a pointer boundary) Now, the fields are reordered to look like this:

```
...
  object mike;
  int joe;
  char color;
  char armed;
...
```

In this example, no memory space is wasted.

Back to Top

Chapter 3. The Java HotSpot Compilers

Overview

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 running on the end user's machine actually executes the bytecodes and compiles each method the first time it is executed.

However, there are several issues 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 slow down compilation performance significantly.

Secondly, even if a JIT 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:

The Java language is dynamically *safe*, meaning that it ensures 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 language allocates all objects on the *heap*, in contrast to C++, where many objects are stack allocated. This means that object allocation rates are much higher for the Java language than for C++. In addition, because the Java 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 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 such as inlining) are much harder to perform for method invocations. Many traditional optimizations are most effective between calls, and the decreased distance between calls in the Java 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 a powerful ability to perform dynamic loading of classes. This makes it far more difficult to perform many types of global optimizations. The compiler must not only be able to detect when these optimizations become invalid due to dynamic loading, but also be able to undo 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 language performance must provide nontraditional answers to these performance issues, rather than blindly applying traditional compiler techniques.

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

Back to Top

Hot Spot Detection

Adaptive optimization solves the problems of JIT compilation by taking advantage of an interesting program property. Virtually all programs spend the vast majority of their time executing a minority of their code. Rather than compiling method by method, just in time, the Java HotSpot VM immediately runs the program using an interpreter, and analyzes the code as it runs to detect the critical hot spots in the program. Then it 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 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 user's needs.

A subtle but important benefit of this approach is that by delaying compilation until after the code has already been executed for a while (measured in machine time, not user time), information can be gathered on the way the code is used, and then utilized to perform more intelligent optimization. As well, 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

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 the hot spot 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. This creates a situation that significantly increases the effectiveness of traditional compiler optimizations, 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, described in the last section, is an important optimization, it has traditionally been very difficult to perform for dynamic object-oriented languages like the Java 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 language can not 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 class may contain new methods that need to be inlined in the appropriate places. Therefore the Java HotSpot VM must be able to dynamically deoptimize (and then reoptimize, if necessary) previously optimized hot spots, even while executing code for the hot spot. Without this capability, general inlining cannot be safely performed on Java technology-based programs.

Both the Java HotSpot Client and Server compilers fully support dynamic deoptimization. This enables both aggressive and optimistic optimization as well as other techniques such as full-speed debugging, which is described later.

Java HotSpot Client Compiler

The Java HotSpot Client Compiler is a simple, fast three-phase compiler. In the first phase, a platform-independent front end constructs a high-level intermediate representation (HIR) from the bytecodes. The HIR uses static single assignment (SSA) form to represent values in order to more easily enable certain optimizations, which are performed during and after IR construction. In the second phase, the platform-specific back end generates a low-level intermediate representation (LIR) from the HIR. The final phase performs register allocation on the LIR using a customized version of the linear scan algorithm, does peephole optimization on the LIR and generates machine code from it.

Emphasis is placed on extracting and preserving as much information as possible from the bytecodes. The client compiler focuses on local code quality and does very few global optimizations, since those are often the most expensive in terms of compile time.

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, constant propagation, global value numbering, and global code motion. It also features optimizations more specific to Java technology, such as null-check and range-check elimination and optimization of exception throwing paths. The register allocator is a global graph coloring allocator and makes full use of large register sets that are 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 for compiled code.

Compiler Optimizations

The Java HotSpot compilers support a suite of advanced optimizations to enable high performance of both traditional straight-line programs as well as object-oriented programming styles. Some of these optimizations include:

Deep inlining and *inlining of potentially virtual calls*: as described above, method inlining combined with global analysis and dynamic deoptimization are used by both the client and server compilers to enable deep inlining and therefore elimination of a substantial amount of method call overhead.
Fast instanceof/checkcast: the Java HotSpot VM and compilers support a novel technique for accelerating the dynamic type tests frequently required by the Java programming language for type safety. This further decreases the run-time cost of programming in object-oriented style.
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 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.
Feedback-directed optimizations: the Server VM performs extensive profiling of the program in the interpreter before compiling the Java bytecode to optimized machine code. This profiling data provides even more information to the compiler about data types in use, hot paths through the code, and other properties. The compiler uses this information to more aggressively and optimistically optimize the code in certain situations. If one of the assumed properties of the code is violated at run time, the code is deoptimized and later recompiled and reoptimized.

Back to Top

Chapter 4. Advanced Features of the Java HotSpot VM

The Java HotSpot VM supports many advanced features to enable high scalability, high performance, and enterprise-class reliability, availability and serviceability.

Scalability

The Java HotSpot VM has recently added automatic self-sizing and adaptation mechanisms called ergonomics. Currently, ergonomics manifests in two principal areas. First, depending on the physical configuration of the machine (taking into account number of processors and available physical memory, for example), either the Client VM or Server VM will be automatically selected. In particular, the Server VM will be selected for machines with larger numbers of processors and larger amounts of RAM, and the size of the garbage-collected heap will be automatically selected for reasonable server-side applications on such hardware. Second, the garbage collection algorithms in the Java HotSpot VM are now self-tuning, so that it is no longer necessary to explicitly specify the relative sizes of the young and old generations. The garbage collectors will self-tune to improve the throughput of the application and reduce pause times. Ergonomic techniques automatically improve scalability of server-side applications, and more work in this area is planned for future releases.

Performance

In addition to the core object-oriented optimizations enabled by the Java HotSpot VM's architecture, the VM and the Sun Java Runtime Environment support some other key performance optimizations:

Fast reflection: the Java libraries now generate bytecode stubs for frequently used reflective objects such as Methods and Constructors. This technique exposes the reflective invocations to the Java HotSpot compilers, yielding much higher performance and, in some cases with the Server VM, complete elimination of the overhead associated with the reflective invocation. This provides significant speedups in reflection-intensive code, such as that used in serialization, RMI, and CORBA code environments.
New I/O optimizations: the Java HotSpot compilers treat operations on New I/O Buffer objects specially, producing high-quality machine code for `get` and `put` method invocations. Combined with the large performance and scalability improvements in network and file I/O offered by the New I/O APIs, Java programming language applications can now achieve similar throughput as applications coded in C and C++. The New I/O Buffer optimizations are also applicable to other problem domains, such as 3D graphics, transferring large amounts of data between the Java platform and the outside world.

Reliability, Availability and Serviceability

The current release of the Java HotSpot VM is the most reliable to date. Recent releases of the VM have set new records for reliability and availability for enterprise applications, based on execution of various large applications by Sun Microsystems, Inc.

The Java HotSpot VM contains the reference implementation of the Java Virtual Machine Tools Interface (JVM TI). This interface allows tools such as profilers, debuggers, and monitors to observe and control the JVM. Included features are:

Full-speed debugging: the Java HotSpot VM utilizes dynamic deoptimization technology to support debugging of applications at full speed. In earlier Java virtual machine implementations, when debugging support was enabled, the application was run only in the interpreter. Enabling the Java HotSpot compilers in debugging scenarios improves performance tremendously, and in many situations it is possible to run with debugging support always enabled for better serviceability of applications. Additionally, the launch of a debugger can be triggered upon the throwing of an exception.
HotSwap support: the object-oriented architecture of the Java HotSpot VM enables advanced features such as on-the-fly class redefinition, or "HotSwap". This feature provides the ability to substitute modified code in a running application through the debugger APIs. HotSwap adds functionality to the Java Platform Debugger Architecture, enabling a class to be updated during execution while under the control of a debugger. It also allows profiling operations to be performed by hotswapping in versions of methods in which profiling code has been inserted.

Several additional features in the Java HotSpot VM improve development and serviceability of Java programming language applications.

JNI error checking: a command-line option, `-Xcheck: jni`, for performing additional JNI checks is available. This enables checks for argument validity to be run during development, where they can be detected before they are deployed and slow down production runs. Specifically, the Java HotSpot VM validates the parameters passed to the JNI function as well as the runtime environment data before processing the JNI request. Any invalid data encountered indicates a problem in the native code, and the VM will terminate with a fatal error in such cases.
Error reporting: if the JVM detects a crash in native code, such as JNI code written by the developer, or if the JVM itself crashes, it will print and log debug information about the crash. This error message normally will include information such as the function name, library name, source-file name, and line number where the error occurred. The result is that developers can more easily and efficiently debug their applications. If an error message indicates a problem in the JVM code itself, it allows a developer to submit a more accurate and helpful bug report.

Signal-chaining facility: Signal-chaining enables the Java platform to better interoperate with native code that installs its own signal handlers. The facility works on both Solaris OE and Linux platforms. The signal-chaining facility was introduced to remedy a problem with signal handling in previous versions of the Java HotSpot VM. Prior to version 1.4, the Java HotSpot VM would not allow application-installed signal handlers for certain signals including, for example, SIGBUS, SIGSEGV, or SIGILL, since those signal handlers could conflict with the signal handlers used internally by the Java HotSpot VM.

Back to Top

Chapter 5. Impact on Software Reusability

Overview

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 allowing companies to take full advantage of object-oriented reusability mechanisms for the first time, 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 fundamentally change the way 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. This trend points 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.

Back to Top

Chapter 6. Summary

The Java HotSpot VM delivers optimal performance for Java applications, delivering advanced optimization, garbage collection, and thread synchronization capabilities. In addition, the VM offers debugging capabilities designed to improve overall reliability, availability and serviceability of Java technology-based applications. The Java HotSpot VM provides separate compilers for client and server environments so that applications can be optimized according to their target deployment environments. Scalability has been significantly enhanced with the availability of a 64-bit Java HotSpot Server Compiler.

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.

Back to Top

Chapter 7. Availability

The Java HotSpot VM is included in the Java SE platform environment. It is available at *Java Sun* for the following environments:

Solaris Operating Environment for SPARC platforms.
Solaris Operating Environment for Intel platforms.
Linux Operating Systems: several versions of Red Hat Enterprise Linux, SuSE Linux and the Sun Java Desktop System are officially supported on Intel platforms. Recent versions of the Java SE platform have undergone limited testing on other Linux platforms. Please see the supported system configurations for Java SE for more information.
Microsoft Windows on Intel platforms.
The Java HotSpot VM is shipping as part of Apple Computer, Inc.'s Macintosh OS X operating system. The Java HotSpot Server VM is also included in the release of Java SE technology from Hewlett-Packard for their PA-RISC hardware platforms.

Back to Top

Chapter 8. Resources

These web sites provide additional information:

Java HotSpot Home Page
Memory Management in the Java HotSpot VM
Ergonomics Documentation for the Java HotSpot VM
Tuning Garbage Collection with the Java HotSpot VM

Back to Top

Technical Articles
Demos and Videos
Forums
Java Magazine
Java.net
Developer Training
Tutorials
Java.com

E-mail this page Printer View