Java Performance Tuning

Strategies, Approaches, and Methodologies

- Like any other activity, Java Performance Tuning also needs a plan, strategy and approach.
- A set of information or background is required in a given domain to be successful.
- To be successful in a Java performance tuning effort, you need to be beyond the stage of
  - "*I don't know what I don't know*" and into the "*I know what I don't know*" stage or already be in the "*I already know what I need to know*" stage.

Stage –”I don’t know what I don’t know”

- A Task involves understanding a new problem domain.
- The first challenge in understanding a new problem domain is to learn as much about the problem as you can because you may know little if anything about the problem domain.
- In this new problem domain there are many artifacts about the problem domain you do not know, or do not know what is important to know.
- In other words, you do not know what you need to know about the problem domain. Hence, the phrase,
  - *“I don’t know what I don’t know.”*

I know what I don’t know

- Normally when you enter a new problem domain,one that you know little about, you eventually reach a point where you have discovered many different things about the problem domain that are important to know.
- But you do not know the specific details about those things that are important to know.
- When you have reached this stage it is called the “*I know what I don’t know*” stage.

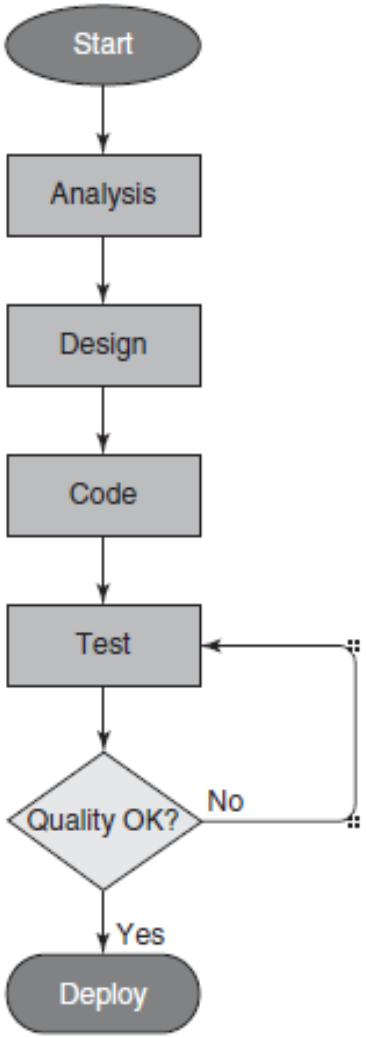
I already know what I need to know.

- At other times you are given a task in a problem domain in which you are familiar or you have developed the necessary skills and knowledge in the area to the point where you are considered a subject matter expert.
- Or as you learn a new problem domain you reach a point where you feel comfortable working within it, i.e., you have learned the information necessary to be effective in the problem domain.
- When you have reached this point, you are at the stage of "*I already know what I need to know.*"

Forces at Play

Traditional Approach for Software Development

- The traditional software development process consists of four major phases:
  - analysis, design, coding, and testing.
- Many applications developed through these traditional software development phases tend to give little attention to **performance** or **scalability** until the application is released, or at the earliest in the testing phase.



Approaches to Performance tuning

- Two approaches
  - Top Down
    - focuses at the top level of the application and drills down the software stack looking for problem areas and optimization opportunities
    - The top down approach is most commonly used by application developers.
  - Bottom Up
    - The bottom up approach is commonly used by performance specialists in situations where the performance task involves identifying performance differences in an application on differing platform architectures, operating systems, or in the case of Java differing implementations of Java Virtual Machines

Java Performance Tuning

Operating System Performance Monitoring

Introduction

- Isolate your application and its operation environment related issues.
- Critical factors in monitoring are
  - Know what to monitor?
  - where in the software stack to monitor?
  - What tools to use?
- The first step in isolating a performance issue is to monitor the application’s behavior.
- Monitoring offers clues as to the type or general category of performance issue.

Definitions

- Three distinct activities are involved when engaging in performance improvement activities:
  - Performance monitoring.
  - Performance profiling, and
  - Performance tuning.

Performance monitoring

- Performance monitoring is an act of nonintrusively collecting or observing performance data from an operating or running application.
- Monitoring is usually a preventative or proactive type of action and is usually performed in a production environnent, qualification environnent, or developent environnent.
- Monitoring is also usually the first step in a reactive situation where an application stakeholder has reported a performance issue but has not provided sufficient information or clues as to a potential root cause.
- In this situation,performance profiling likely follows performance monitoring.

Performance profiling

- Performance profiling in contrast to performance monitoring is an act of collecting performance data from an operating or running application that may be intrusive on application responsiveness or throughput.
- Performance profiling tends to be a reactive type of activity, or an activity in response to a stakeholder reporting a performance issue, and usually has a more narrow focus than performance monitoring.
- Profiling is rarely done in production environments.
- It is typically done in qualification, testing, or development environments and is often an act that follows a monitoring activity that indicates some kind of performance issue.

Performance tuning

- Performance tuning, in contrast to performance monitoring and performance profiling, is an act of changing tune-ables, source code, or configuration attribute(s) for the purposes of improving application responsiveness or throughput.
- Performance tuning often follows performance monitoring or performance profiling activities.

CPU Utilization

CPU Utilization

- For an application to reach its highest performance or scalability it needs to not only take full advantage of the CPU cycles available to it but also to utilize them in a manner that is not wasteful.
- Being able to make efficient use of CPU cycles can be challenging for multithreaded applications running on multiprocessor and multicore systems.
- Additionally, it is important to note that an application that can saturate CPU resources does not necessarily imply it has reached its maximum performance or scalability.
- To identify how an application is utilizing CPU cycles, you monitor CPU utilization at the operating system level.

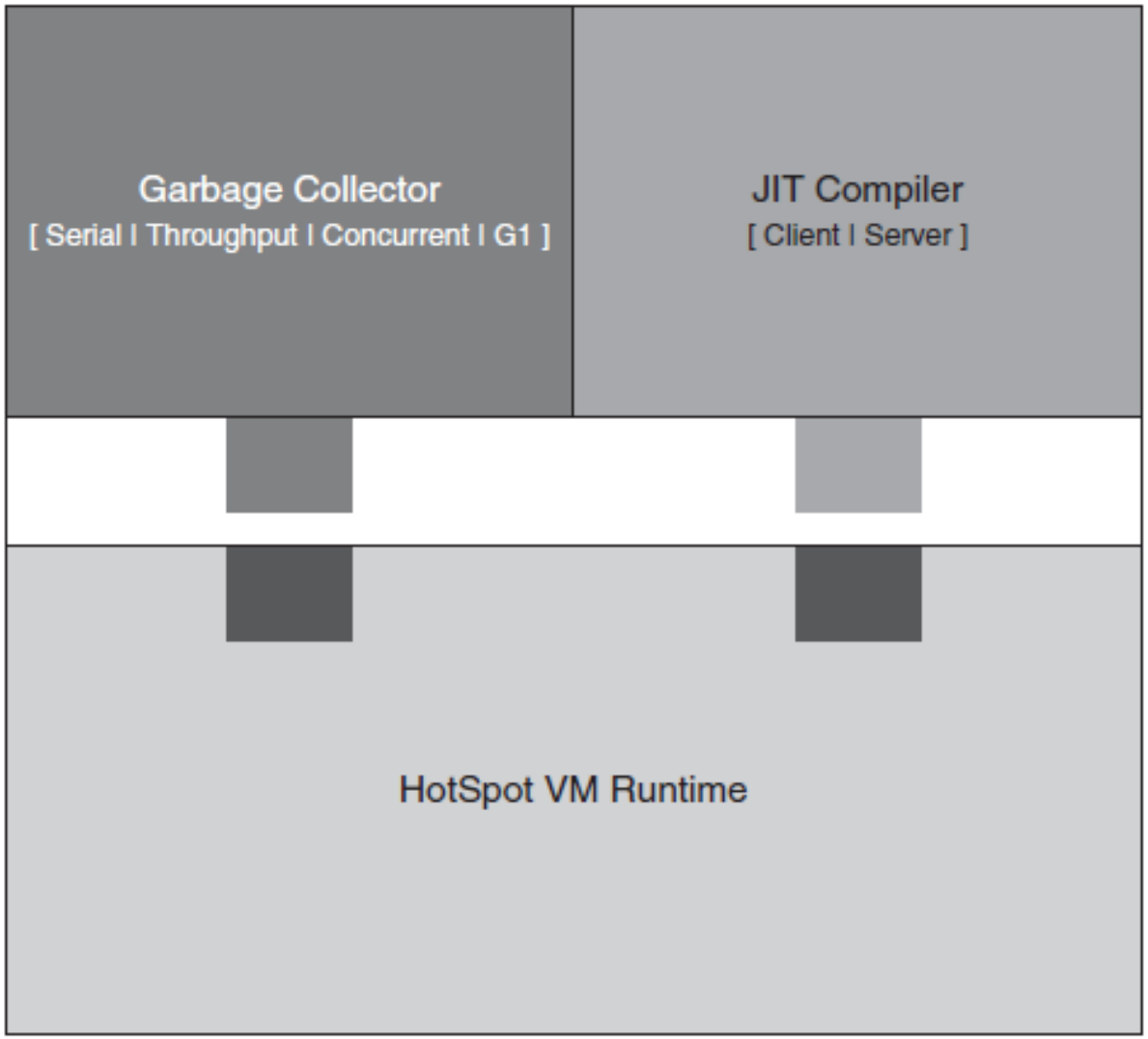
Java Performance Tuning

JVM Overview

Java Virtual Machine

- A **Java virtual machine** (**JVM**) is an abstract computing machine that enables a computer to run a Java program.
- There are three notions of the JVM
  - specification, implementation, and instance.
- The specification is a document that formally describes what is required of a JVM implementation.
- Having a single specification ensures all implementations are interoperable.
- A JVM implementation is a computer program that meets the requirements of the JVM specification.
- An instance of a JVM is an implementation running in a process that executes a computer program compiled into Java bytecode

HotSpot VM High Level Architecture



HotSpot VM High Level Architecture

- The JIT compiler, client or server, is pluggable.
- Choice of garbage collector: Serial GC, Throughput, Concurrent, or G1.
- The HotSpot VM Runtime provides services and common APIs to the HotSpot JIT compilers and HotSpot garbage collector.
- In addition the HotSpot VM Runtime provides basic functionality to the VM such as a launcher, thread management, Java Native Interface, and so on.

HotSpot VM Runtime

- Command Line Options
- VM Life Cycle
- VM Class Loading
  - Class Loading Phases
  - Class Loader Delegation
  - Bootstrap Class Loader
  - Type Safety
  - Class Metadata in HotSpot
  - Internal Class Loading Data
- Byte Code Verification
- Class Data Sharing
- Interpreter
- Exception Handling
- Synchronization
- Thread Management

HotSpot VM Runtime

- The VM Runtime provides the core functionality of the HotSpot VM.
- The HotSpot VM Runtime encompasses many responsibilities, including parsing of command line arguments, VM life cycle, class loading, byte code interpreter, exception handling, synchronization, thread management, Java Native Interface, VM fatal error handling, and C++ (non-Java) heap management

Command Line Options

- The HotSpot VM Runtime parses the many command line options and configures the HotSpot VM based on those options.
- A number of command line options and environment variables can affect the performance characteristics of the HotSpot VM.
- Some of these options are consumed by the HotSpot VM launcher such as
  - the choice of JIT compiler
  - choice of garbage collector;
- some are processed by the launcher and passed to the launched HotSpot VM where they are consumed such as Java heap sizes.

Command Line Options

- **3** main categories of command line options
  - **Standard options**
    - expected to be accepted by all Java Virtual Machine implementations as required by the *Java Virtual Machine Specification*
  - **Non standard options**
    - Nonstandard command line options are not guaranteed to be supported in all JVM implementations, nor are they required to be supported in all JVM implementations.
    - Nonstandard command line options are also subject to change without notice between subsequent releases of the Java SDK
  - **Developer Options**
    - Developer command line options often have specific system requirements for correct operation and may require privileged access to system configuration parameters.

How to provide command line VM Options?

- Command line options control the values of internal variables in the HotSpot VM, all of which have a type and a default value.
- Nonstandard command line options begin with a **-X** prefix.
- Developer command line options in the HotSpot VM begin with a –XX prefix.
- For developer command line options (-XX options) with boolean flags, a **+** or **-** before the name of the options indicates a true or false value, respectively, to enable or disable a given HotSpot VM feature or option.
- For example, -XX:+AggressiveOpts sets a HotSpot internal boolean variable to true to enable additional performance optimizations.

VM Life Cycle

- The HotSpot VM Runtime is responsible for launching the HotSpot VM and the shutdown of the HotSpot VM.
- The component that starts the HotSpot VM is called the launcher.
- There are several HotSpot VM launchers.
  - The most commonly used launcher is the *java command* on Unix/Linux and on Windows the java and *javaw commands*.
  - It is also possible to launch an embedded JVM through the JNI interface, JNI_CreateJavaVM.
  - In addition, there is also a network-based launcher called javaws, which is used by Web browsers to launch applets.
  - The trailing "ws" on the javaws is often referred to as "web start." Hence the term "Java web start" for the javaws launcher.

HotSpot VM Runtime

Class Loading

Java Class Loaders

- Java ClassLoader loads a java class file into java virtual machine.

Types (Hierarchy) of Java Class Loaders

- Java class loaders can be broadly classified into below categories:
- Bootstrap Class Loader
  - Bootstrap class loader loads java's core classes like java.lang, java.util etc. These are classes that are part of java runtime environment. Bootstrap class loader is native implementation and so they may differ across different JVMs.
- Extensions Class Loader
  - JAVA_HOME/jre/lib/ext contains jar packages that are extensions of standard core java classes. Extensions class loader loads classes from this ext folder. Using the system environment propery java.ext.dirs you can add 'ext' folders and jar files to be loaded using extensions class loader.
- System Class Loader
  - Java classes that are available in the java classpath are loaded using System class loader.

More on Class Loaders

- You can see more class loaders like java.net.URLClassLoader, java.security.SecureClassLoader etc.
- Those are all extended from java.lang.ClassLoader
- These class loaders have a hierarchical relationship among them.
- Class loader can load classes from one level above its hierarchy. First level is bootstrap class loader, second level is extensions class loader and third level is system class loader.

Byte Code Verification

- Java Virtual Machine cannot guarantee that the code was produced by a trustworthy javac compiler.
- It must reestablish type-safety through a process at link time called bytecode verification.
- If any violations are found, the Java Virtual Machine throws a **VerifyError** and prevents the class from being linked.

Class Data Sharing

- Class data sharing is a feature introduced in Java 5 that was intended to reduce the startup time for Java applications, in particular small Java applications, as well as reduce their memory footprint.
- During subsequent Java Virtual Machine invocations, the shared archive is memory-mapped into the JVM, which saves the cost of loading those classes and allowing much of the JVM's metadata for these classes to be shared among multiple JVM processes.

Class Data Sharing

- Class data sharing reduces the start up time for JVM
- produces better results for smaller applications because it eliminates a fixed cost of loading certain Java SE core classes.
- The smaller the application relative to the number of Java SE core classes it uses, the larger the saved fraction of startup time.
- In the HotSpot VM, the class data sharing implementation introduces new Java subspaces into the permanent generation space that contains the shared data.
- The **classes.jsa** shared archive is memory mapped into these spaces in permanent generation at HotSpot VM startup time. Subsequently, the shared region is managed by the existing HotSpot VM memory management subsystem.

Interpreter

Interpreter

- The HotSpot VM interpreter is a template based interpreter.
- The HotSpot VM Runtime generates the interpreter in memory at JVM startup using information stored internally in a data structure called a TemplateTable.
- The TemplateTable contains machine dependent code corresponding to each bytecode.
- A template is a description of each bytecode.
- The HotSpot VM's TemplateTable defines all the templates and provides accessor functions to get the template for a given bytecode.
- The template table generated in memory can be viewed using what is called a HotSpot "debug" VM and the non product flag -XX:+PrintInterpreter.

HotSpot VM Runtime

Exception Handling

Exception Handling

- Java Virtual Machines use exceptions to signal that a program has violated the semantic constraints of the Java language.
- An exception causes a nonlocal transfer of control from the point where the exception occurred, or was thrown, to a point specified by the programmer, or where the exception is caught.
- The HotSpot VM interpreter, its JIT compilers, and other HotSpot VM components all cooperate to implement exception handling.
- There are two general cases of exception handling; either the exception is thrown or caught in the same method, or it is caught by a caller. The latter case is more complicated and requires stack unwinding to find the appropriate handler

Synchronization

Synchronization

- *synchronization* is described as a mechanism that prevents, avoids, or recovers from the inopportune interleavings, commonly called *races,* of concurrent operations.
- The HotSpot VM provides Java monitors by which threads running application code can participate in a mutual exclusion protocol.
- A Java monitor is either locked or unlocked, and only one thread may own the monitor at any one time.
- Only after acquiring ownership of a monitor may a thread enter a critical section protected by the monitor.
- In Java, critical sections are referred to as *synchronized blocks* and are delineated in code by the synchronized statement.

HotSpot VM Runtime

Thread Management

Thread Management

- Thread management covers all aspects of the thread life cycle, from creation through termination along with the coordination of threads within the HotSpot VM.
- This involves management of threads created from Java code, regardless of whether they are created from application code or library code, native threads that attach directly to the HotSpot VM, or internal HotSpot VM threads created for other purposes.
- While the broader aspects of thread management are platform independent, the details vary depending on the underlying operating system.

Threading Model

- The threading model in the Hotspot VM is a one-to-one mapping between Java threads, an instance of **java.lang.Thread**, and native operating system threads.
- A native operating system thread is created when a Java thread is started and is reclaimed once it terminates.
- The operating system is responsible for scheduling all threads and dispatching them to an available CPU.
- The relationship between Java thread priorities and operating system thread priorities is a complex one that varies across systems.

Thread Creation and Destruction

- There are two ways for a thread to be introduced in the HotSpot VM; either by executing
  - Java code that calls the start() method on a java.lang.Thread object,
  - or by attaching an existing native thread to the HotSpot VM using JNI.
- Internally to the HotSpot VM there are a number of objects, both C++ and Java, associated with a given thread in the HotSpot VM. These objects, both Java and C++, are as follows:
  - A java.lang.Thread instance that represents a thread in Java code.
  - A C++ JavaThread instance that represents the java.lang.Thread instance internally within the HotSpot VM.
    - It contains additional information to track the state of the thread.
- A JavaThread holds a reference to its associated java.lang.Thread object, as an ordinary object pointer, and the java.lang.Thread object also stores a reference to its Java Thread as a raw int.
- A JavaThread also holds a reference to its associated OSThread instance.
- An OSThread instance represents an operating system thread and contains additional operating-system-level information needed to track thread state.
- The OSThread also contains a platform specific “handle” to identify the actual thread to the operating system.

Thread States

- The HotSpot VM uses a number of different internal thread states to characterize what each thread is doing.
- This is necessary both for coordinating the interactions of threads and for providing useful debugging information if things go wrong.
- From the HotSpot VM perspective the possible states of the main thread are
  - **New thread.** A new thread in the process of being initialized
  - **Thread in Java.** A thread that is executing Java code
  - **Thread in vm.** A thread that is executing inside the HotSpot VM
  - **Blocked thread.** The thread is blocked for some reason (acquiring a lock, waiting for a condition, sleeping, performing a blocking I/O operation, and so on)

Thread States

- For debugging purposes additional state information is also maintained for reporting by tools, in thread dumps, stack traces, and so on.
- This is maintained in the internal HotSpot C++ object OSThread. Thread states reported by tools, in thread dumps, stack traces, and so on, include
  - **MONITOR_WAIT.** A thread is waiting to acquire a contended monitor lock.
  - **CONDVAR_WAIT.** A thread is waiting on an internal condition variable used by the HotSpot VM (not associated with any Java object).
  - **OBJECT_WAIT.** A Java thread is performing a java.lang.Object.wait() call.

Internal VM Threads

- **VM thread.** A singleton C++ object instance that is responsible for executing VM operations.
- **Periodic task thread.** A singleton C++ object instance, also called the WatcherThread, simulates timer interrupts for executing periodic operations within the HotSpot VM.
- **Garbage collection threads.** These threads, of different types, support the serial, parallel, and concurrent garbage collection.
- **JIT compiler threads.** These threads perform runtime compilation of bytecode to machine code.
- **Signal dispatcher thread.** This thread waits for process directed signals and dispatches them to a Java level signal handling method.
- All these threads are instances of the internal HotSpot C++ Thread class, and all threads that execute Java code are internal HotSpot C++ **JavaThread** instances.
- The HotSpot VM internally keeps track of all threads in a linked-list known as the Threads list and is protected by the Threads lock—one of the key synchronization locks used within the HotSpot VM.

HotSpot VM Runtime

C++ Heap Management

C++ Heap Management

- HotSpot VM also uses a C/C++ heap for storage of HotSpot VM internal objects and data.
- Within the HotSpot VM and not exposed to a user of the HotSpot VM, a set of C++ classes derived from a base class called Arena is used to manage the HotSpot VM C++ heap operations.
- The Arena base class and its subclasses provide a rapid C/C++ allocation layer that sits on top of the C/C++ malloc/free memory management routines.
- Each Arena allocates memory blocks (internally the HotSpot VM refers to them as Chunks) from three global ChunkPools.
- Each ChunkPool satisfies allocation requests for a distinct range of allocation sizes.

HotSpot VM Runtime

Java Native Interface (JNI)

Java Native Interface

- The Java Native Interface, referred to as JNI hereafter, is a native programming interface.
- It allows Java code that runs inside a Java Virtual Machine to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly language.