



J2SE™ & J2EE™ Performance: Learn How to Write High Performance Java Applications

Matt Hosanee

Partner Engineering

Sun Microsystems



- J2SE Performance
 - Memory allocation and garbage collection
 - Profiling
- J2EE performance
 - Tuning guidelines
- Further Information

J2SE Performance: Memory Allocation & GC

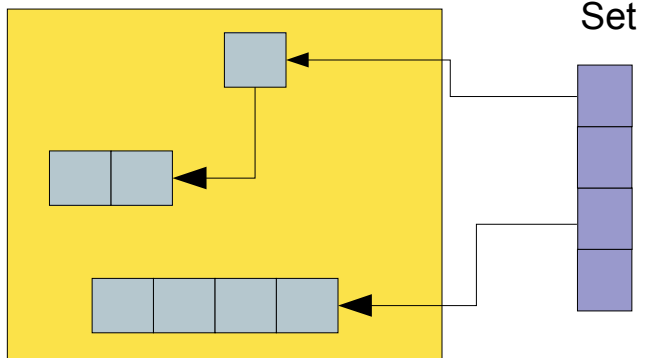
- Age old problems
 - How to allocate space efficiently
 - How to reclaim unused space (garbage) efficiently and reliably
- C (malloc and free)
- C++ (new and delete)
- Java™ (new and Garbage Collection)

- Garbage detection
 - Distinguish live objects from garbage
 - Reference counting
 - Cyclic reference problem
- Garbage reclamation
 - Make space available to the running program again

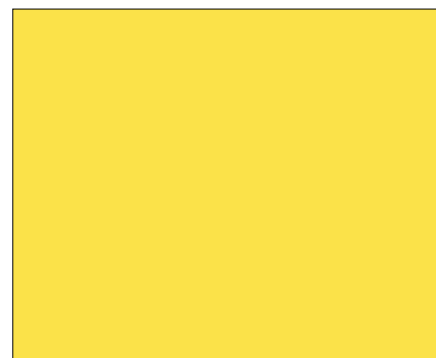
- Most objects are very short lived
 - 80-98% of all newly allocated objects die within a few million instructions
 - 80-98% of all newly allocated objects die before another megabyte has been allocated
- This impacts heavily on choices for GC algorithms

- Copying
- Mark - Sweep
- Mark - Compact
- Incremental
- Generational
- Parallel Copy
- Concurrent
- Parallel Scavenge

From space

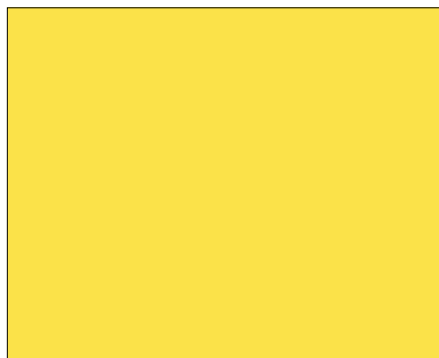


To space

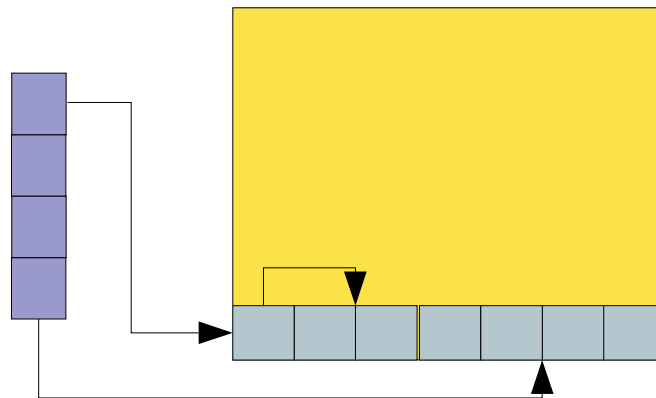


Before

From space



To space



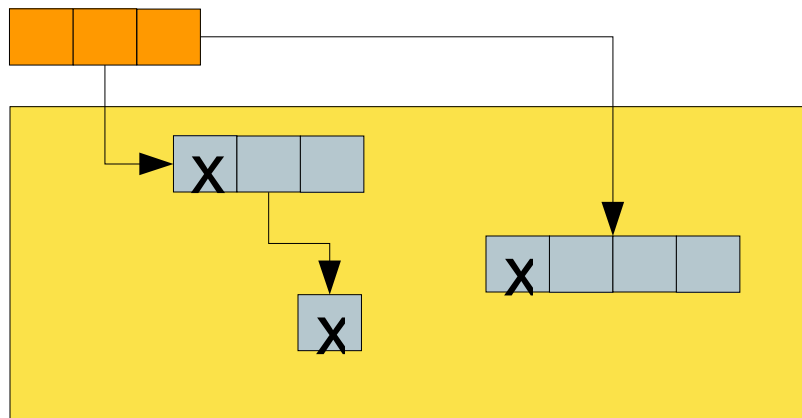
After

- Stop-the-world collector
- Very Efficient
 - Traverses object list and copies objects in a single cycle
 - Simultaneous detection and reclamation
- GC pause is directly proportional to total size of live objects
 - Bigger semi-spaces improve efficiency
 - Less frequent GC, more dead objects

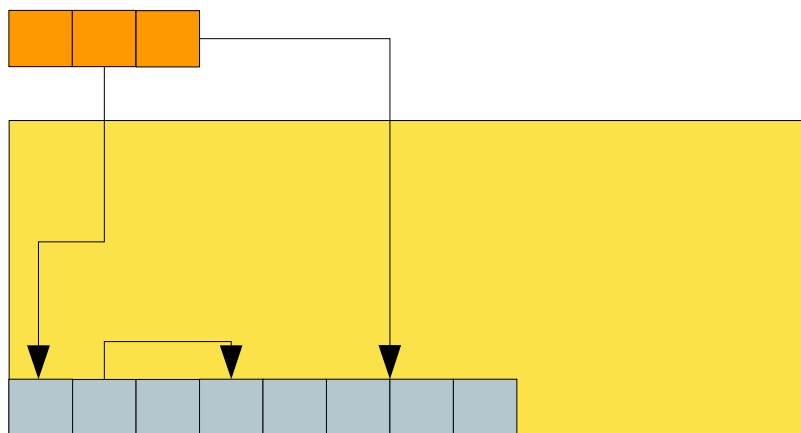
- Stop-the-world collector
- Distinguish live objects from garbage
 - Traverse graph of pointer relationships
 - Mark objects that can be reached
- Reclaim the space
 - Heap space is “swept” for marked areas
 - Free space is added to a free list, ready for use

- Different-sized objects cause fragmentation
 - Multiple free lists for different-sized blocks
- Cost of collection proportional to size of heap
 - Not just live objects
- Locality of reference
 - New objects get interleaved with old objects
 - Bad for VM-based operating systems

Mark – Compact GC



Before

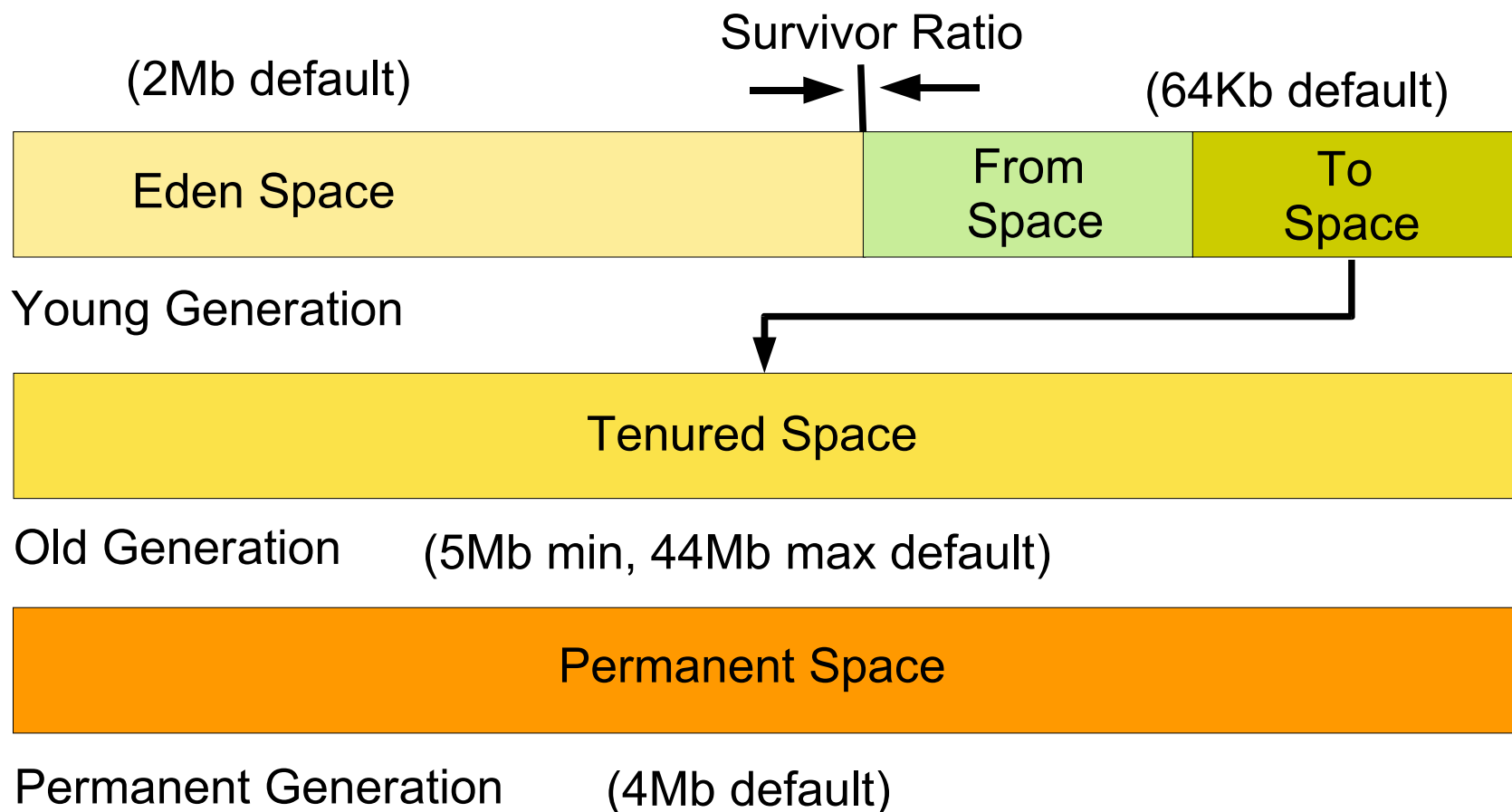


After

- Eliminates fragmentation issue of Mark-Sweep
- Allocation becomes stack-based
- Order of objects maintained
 - Locality of reference
- Requires multiple passes to complete
 - Mark live objects
 - Compute new location
 - Update pointers

- Stop-the-world impacts performance
 - Big heap, big pauses (00's – 000's ms)
- Interleave units of GC work with application work
- Problem is that references change while GC runs
 - Get floating garbage

- Old objects tend to live for a long time
 - GC can spend lots of time analysing and copying the same objects
- Generational GC divides heap into multiple areas (generations)
 - Objects segregated by age
 - New objects die more quickly, GC more frequent
 - Older generations collected less frequently
 - Different generations use different algorithms



Eden = NewSize –

$((\text{NewSize} / (\text{SurvivorRatio} + 2)) * 2)$

From Space = $(\text{NewSize} - \text{Eden} / 2)$

To Space = $(\text{NewSize} - \text{Eden}) / 2)$

- -XX:NewSize
- -XX:MaxNewSize
- -XX:NewRatio
- -XX:SurvivorRatio

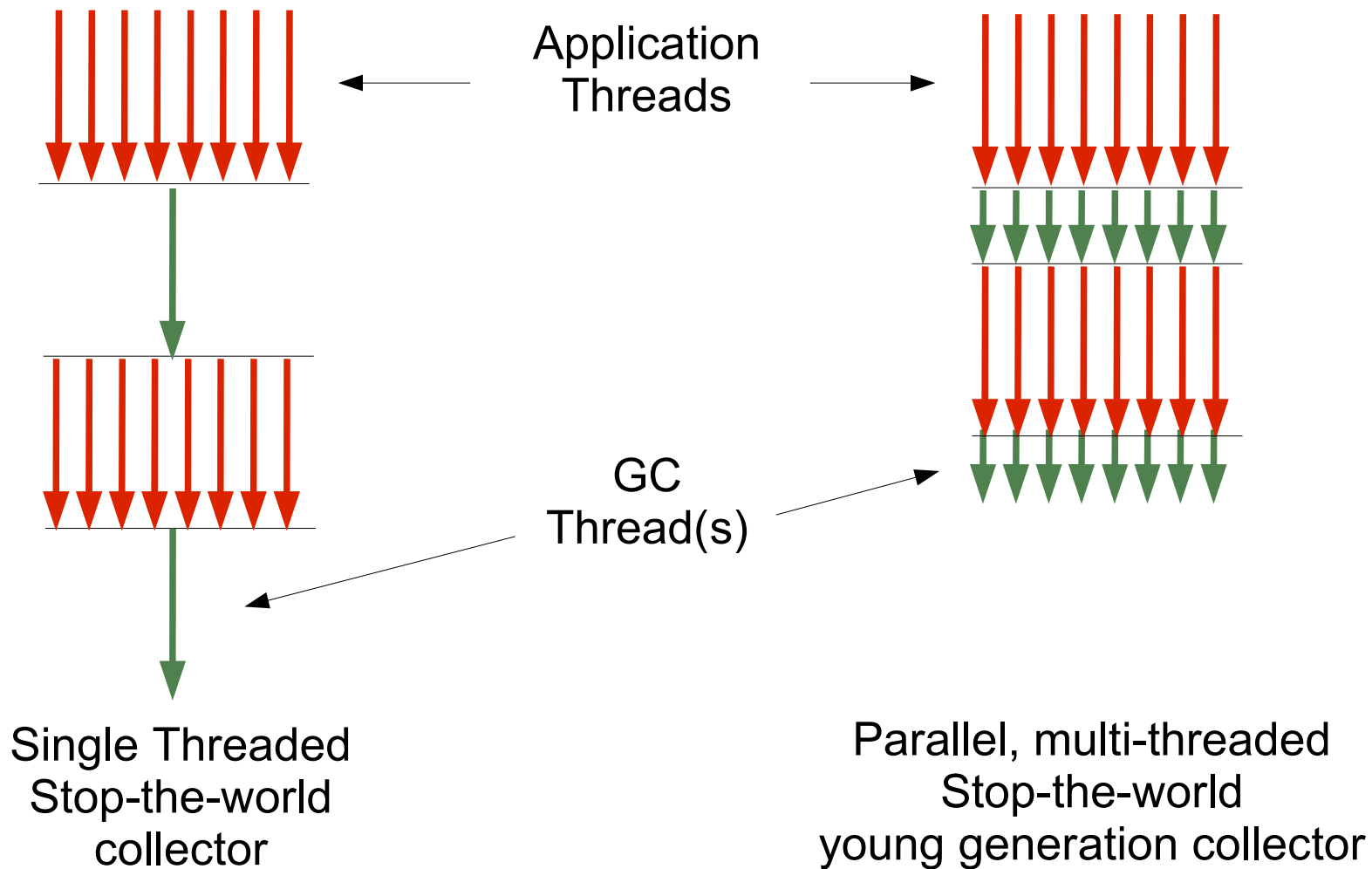
- Tenured generation
 - Objects with long lifetime
- -XX:OldSize
- -XX:MinHeapFreeRatio
- -XX:MaxHeapFreeRatio

- Used to hold class files
- Default size is 4Mb

- -XX:PermSize
- -XX:MaxPermSize
- -Xnoclassgc

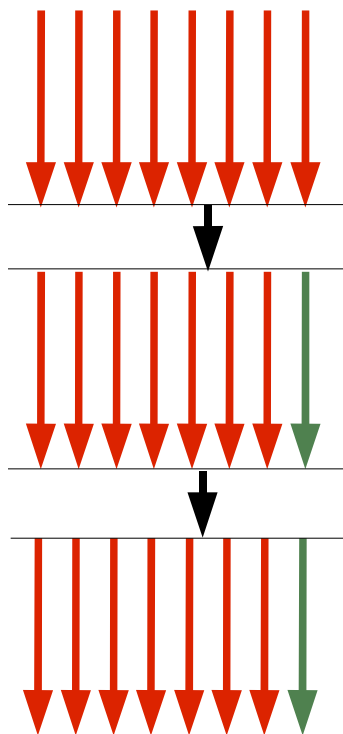
- Problem: multiple threads creating objects
 - All trying to access eden simultaneously
 - Multiple CPU machine: contention
- Solution: Thread local allocation
 - -XX:UseTLAB
 - -XX:TLABSize=<size-in-bytes>
 - -XX:ResizeTLAB
 - -XX:MaxTLABRatio

- Works on young generation
- Similar to copy-collector
 - Still stop-the-world
- Allocates as many threads as CPUs
 - Algorithm optimized to minimize contention
- Maximize work throughput
 - Work stealing



- `-XX:+UseParNewGC`
 - Default copy collector will be used on single CPU machines
- `-XX:ParallelGCThreads=<num>`
 - Default is number of CPUs
 - Can be used to force the parallel copy collector to be used on single a CPU machine
 - Reduce for multiple JVMs on MP machine

- XX:+UseConcMarkSweepGC
- XX:CMSInitiatingOccupancyFraction=*n*



Application Threads

Stop-the-world initial mark phase

Concurrent mark phase

Concurrent precleaning phase

Stop-the-world re-mark phase

Concurrent sweep phase

Reset phase

- Stop-the-world
- Similar to parallel-copy collector
- Aimed at large young spaces (12-80Gb)
 - Larger than old generation
- Scales well with more CPUs
- Adaptive tuning policy
 - Survivor ratio
- Promotion undo to prevent out of memory

- Works on old generation
- `-XX:+UseParallelGC`
- `-XX:ParallelGCThreads=<num>`
 - Control number of threads
- `-XX:+UseAdaptiveSizePolicy`
 - Automatically sizes the young generation and selects optimum survivor ratio

- Rate of object creation
- Object life spans
 - Temporary, intermediate, long
- Types of object
 - Size, complexity
- Relationships between objects
 - Difficulty of determining and tracking object references

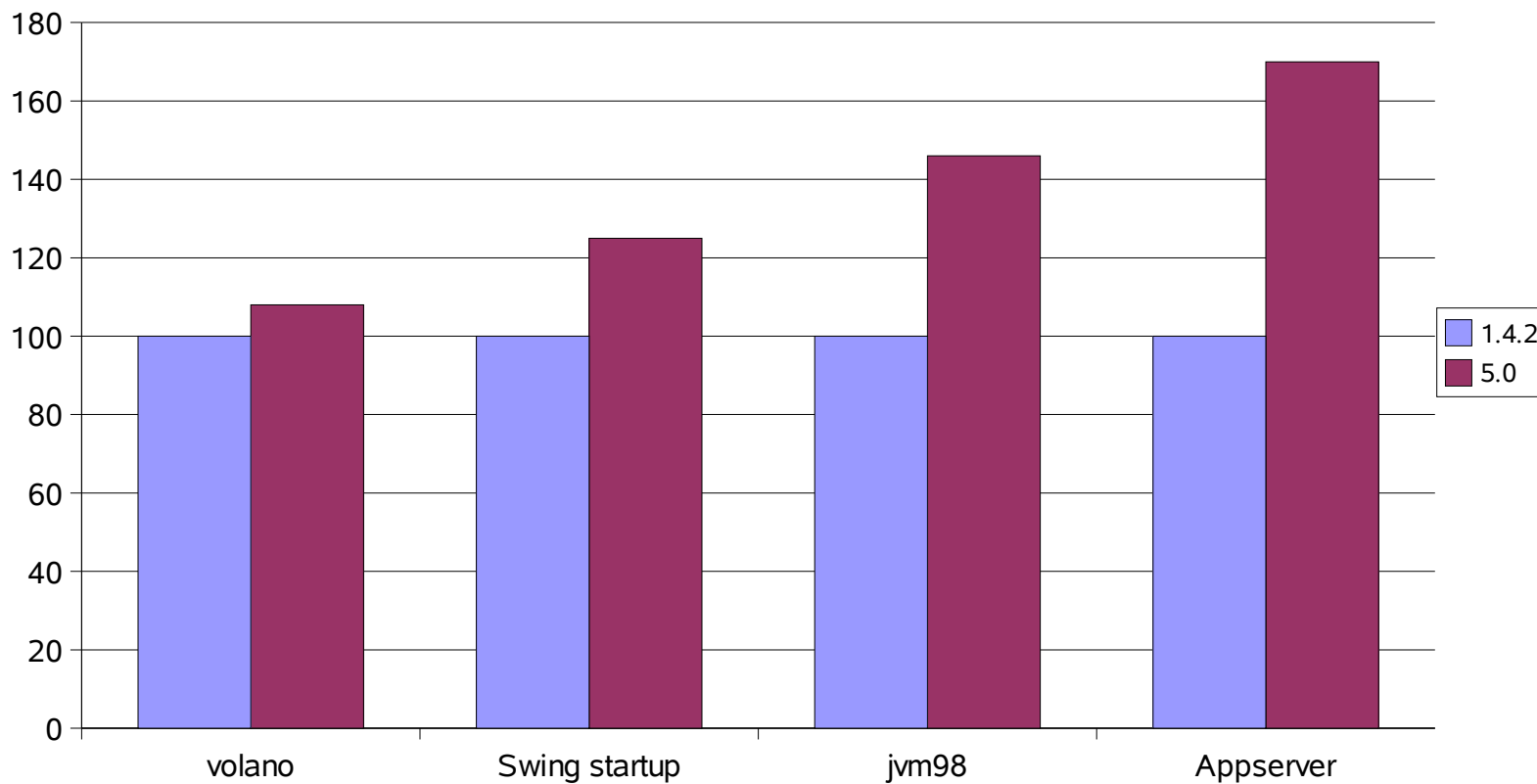
- Profile, profile, profile!
- Use profile data to determine factors affecting performance
- Modify parameters to optimize performance
- Repeat

- Simplest approach
 - -verbose:gc
 - -Xrunhprof
 - -XX:+PrintGCDetails
 - -XX:+PrintGCTimeStamps
 - -XX:+PrintHeapAtGC
 - Warning: very verbose

- jconsole (shipped with J2SE 5)
 - Graphical heap/GC data in realtime
- JFluid (now included with NetBeans IDE)
 - Targeted profiling (low impact)
- jvmstat (from Sun website)
 - similar functionality to jconsole
- GCPortal (from Sun website)
 - Processes verbose:gc output
 - Creates web page with graph/tuning hints

- Always upgrade to the latest version of the JDK/JRE
 - Sun is always working to improve performance
 - Sun is always working to reduce the number of 'undocumented features'

Solaris Sparc



- Can be good for heavy weight objects
 - Database connections/threads
 - Reduce frequency of young GC
- Can also be bad
 - Pooling can be more expensive than creation/collection
 - Can violate good OO design principles

- -XX:PretenureSizeThreshold
 - Allocate space directly in old heap
- Promote all live objects
 - No tenuring of objects in survivor spaces
 - Good for apps with few intermediate objects
- -XX:MaxTenuringThreshold=0
 - Number of times object is copied between survivor spaces
- -XX:SurvivorRatio=100
 - All young generation is allocated to eden space

- Extremely important to GC performance
- Factors to consider
 - Young GC frequency/collection time
 - Ratio and number of short, intermediate and long life objects
 - Promotion size
 - Old GC frequency/collection times
 - Old heap fragmentation/locality problems

- Fragmentation is not an issue
 - Locality of reference could be
- Maximize collection of temporary objects
 - Reduces promotion & tenuring
- Minimize frequency of GC
- Rule of thumb: make it as large as possible
 - Given acceptable collection times

- Ensure heap fits in physical memory
 - Paging and locality of reference issues
- Undersized heap can lead to fragmentation
- Oversized heap increases collection times
 - Locality of reference problems
 - Use ISM and Variable page sizes to alleviate

- Designed for use on big memory Solaris machines
 - Don't use if memory requirements will cause paging
 - JDK1.3.1 introduced support for heaps > 2Gb
 - ISM uses larger page sizes (4Mb rather than 8Kb)
 - Locks pages into memory (no paging to disk)
 - -XX:+UseISM (Solaris Only)
 - -XX:+UsePermISM (Solaris Only)
 - -XX:+UseMPSS (Solaris 9 Only)
 - Need to change shm parameters in /etc/system

- `-XX:+UseAgressiveHeap`
 - Must have min of 256MB RAM
 - Overall heap will be around 3850Mb
 - Thread allocation area 256MB
 - GC deferred as long as possible
 - Do not use `-Xms` or `-Xmx` with this
 - May cause stack space to run out
 - Use `-Xss` to compensate
 - Not suited to multi-app servers

- Allocate more memory to the JVM
 - 64Mb default is often too small
- Set `-Xms` and `-Xmx` to be the same
 - Increases predictability, improves startup time
- Set Eden/Tenured space ratio
 - Eden >50% is bad
 - Except for parallel scavenge collector
 - Eden = 33%, Tenured = 66% seems to be good
- Consider disabling explicit GC
 - `-XX:+DisableExplicitGC`

- Auto-detected
- 2 CPU, 2GB mem (except windows)
 - Uses server compiler
 - Uses parallel collector
 - Initial heap size is 1/64 of physical memory up to 1GB
 - Max heap size is 1/4 of physical memory up to 1GB

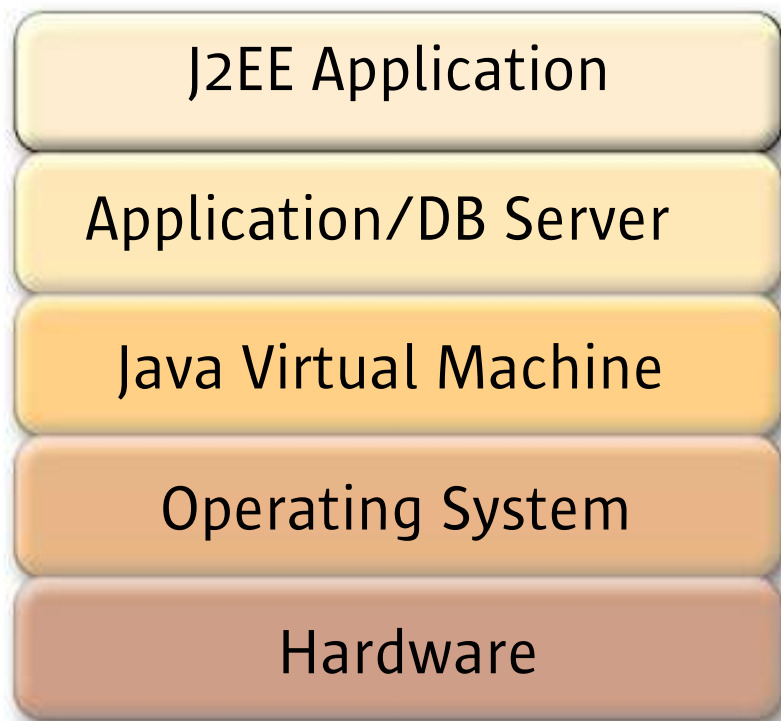
- Maximum pause time goal
 - `-XX:MaxGCPauseMillis=<nnn>`
 - This is a hint, not a guarantee
 - GC will adjust parameters to try and meet goal
 - Can adversely effect applicaiton throughput
- Throughput goal
 - `-XX:GCTimeRatio=<nnn>`
 - GC Time : Application time = $1 / (1 + nnn)$
 - e.g. `-XX:GCTimeRatio=19` (5% of time in GC)

J2EE Performance: Tuning Guidelines

- Startup time
- Peak/sustained performance
- Perceived performance
 - Performance as per the user
- Scalability
 - Readiness of application to handle increasing load without requiring change in design

- Ask yourself these questions
- Is it an external issue?
 - Database systems, messaging systems, etc.
- Is it an operating environment issue?
 - Memory, disk, CPU, network
- Is it an application issue?
 - Resulting from bad design or bad coding
 - Over-design is a bad design

The key to your performance problem can lie in any of these layers. Hence, you have to be able to tune any of these layers.



→ Covered in J2SE Performance part

→ Out of scope. However, we will provide generic guidelines.

- Avoid shared modified class variables in servlets
 - Removes synchronized blocks of code
- Session creation is expensive
 - Invalidate sessions no longer needed
 - Use `<%page session=false%>` directive
 - prevents automatic session creation in JSP
 - Do not store large object graphs in HttpSession
 - avoid forced Java serialization
 - Do not use HttpSession as cache for transactional data
 - Use “read-only” entity beans if available

- Cache EJB references to avoid JNDI lookups
 - Cache EJB home objects in the servlet's `init()`
- Cache bean specific resources in `setSessionContext()` or `ejbCreate()`
 - Release resources in `ejbRemove()` method
- Remove stateful session beans when not needed
 - Avoid passivation = disk I/O
- Use pass-by-reference for remote interfaces if possible
 - Sun Java System Application Server allows pass-by-reference semantics in Sun deployment descriptor

- Ensure EJB stubs generated without nolocals stubs switch by rmic
 - Generate stubs optimized for same-process clients and servers
 - Helps performance where EJB clients (Servlets/JSP) and EJBs are co-located in the same JVM
 - rmic options can be changed using administrative tools provided by your container

	Pooling	Caching
Stateless Session Beans	X	
Stateful Session Beans		X
Entity beans (BMP/CMP)	X	X
Message-driven beans	X	

- Pooling
 - Instances of beans of same type without identity
 - Improves performance by reducing bean class instance creation time
- Caching
 - Used when number of concurrent users of beans exceeds that of maximum allowable number of bean instances

- **Steady pool size** – initial and minimum number of beans that must be maintained in a pool
- **Pool resize quantity** – number of beans to be created or deleted when the pool is being resized by the server
- **Maximum pool size** – maximum pool size
 - Can specify 0 to denote unbounded pool
 - CAUTION – Can overload JVM heap
- **Pool idle timeout** – maximum time that bean is allowed to remain idle in the pool after which bean is destroyed

- Ensure initial and max values of pool size are representative of normal and peak loads
- Setting very large initial or max pool size
 - ineffective use of system resources when app does not have much concurrent load
 - Leads to very long GC pauses
- Setting small initial or max pool size
 - Causes lot of object creation and GC overhead

- **Cache resize quantity** – number of beans created or deleted when cache is being serviced
- **Maximum cache size** – maximum number of beans in a cache
 - 0 denotes unbounded cache.
- **Cache idle timeout** – maximum time stateful session bean or entity bean allowed to be idle in the cache, after which the bean will be passivated
- **Removal timeout**
 - Applicable only to stateful session beans
 - Max time period stateful session bean is allowed to remain passivated before being removed from persistent store

- Client will not be able to access bean after its state is removed
- **Victim selection policy**
 - Applicable only to stateful session beans
 - Algorithm for selecting beans to be removed from the cache
 - Several algorithms
 - Not recently used
 - Least recently used
 - First in, first out

- Use beans from cache as much as possible
 - Bean in cache represents ready state
 - Bean has identity associated with it
 - Beans moving out of cache have to be passivated or destroyed
 - If passivated, bean must be activated to return to cache
 - Any request using cached beans avoids overhead
 - Creation
 - Setting identity
 - Possibly activation.

- Cache too big
 - Memory consumed by beans affects available heap
 - Can mean longer and more frequent, full GC
 - Application server might run out of memory
- Cache too small
 - lots of passivation and activation
 - serialization and de-serialization
 - heavy load on CPU and disk I/O
- Periodic cleaner removes beans from cache that have reached cache idle timeout period

- Bigger cache for frequently used beans
- Entity bean cache and pool sizes should be larger as compared to session beans
- Use lazy loading to optimize memory and network bandwidth consumption
 - To code BMP for lazy loading, put relevant SQLs in the appropriate getXXX() methods.
 - For CMP, most of the containers support lazy loading

- Lazy Loading Caveat
 - Used incorrectly can cause multiple network round-trips
 - Closely monitor data that clients access frequently
 - Load that data when loading the bean
- Mark entity beans as read-only or read-mostly
 - Read-only – avoid unnecessary `ejbStore()` calls at end of each transaction
 - Read-mostly – use if data changes infrequently
 - Ensures that container calls `ejbLoad()` on bean after refresh timeout period

- Tune commit options for entity beans
 - most app servers support commit options B and C
 - Commit option controls action taken by container when the transaction completes

- Option B
 - When transaction completes, the bean is kept in the cache with its identity
 - Next invocation for same primary key can be serviced by same bean instance from cache
 - After calling `ejbLoad()` to synchronize state with database

- Option C
 - When transaction completes, `ejbPassivate()` is called and the bean is disassociated from its primary key
 - This bean is returned to the free pool.
 - Next invocation for same primary key must:
 - Get a bean instance from free pool
 - Set the primary key on this instance
 - Call `ejbActivate()` on the instance.
 - Call `ejbLoad()` to synchronise with database

- Option B avoids `ejbActivate()/ejbPassivate()` calls
 - in most cases will offer better performance
- Option C better if beans in cache are rarely used
 - Container puts bean back into free pool enabling re-use

- How to choose between option B and C
- Look at cache-hits value using your EJB server monitoring tools
 - Cache-hits high compared to cache-misses use option B
 - Otherwise use commit option C
 - Still need to tune cache initial, maximum and resize quantities to achieve optimal performance
 - Option B: Bigger cache, smaller entity bean pool
 - Option C: Bigger entity bean pool, smaller cache

- Message-driven beans are pooled
 - All stateless session bean pool guidelines apply
 - Large MDB pool
 - Better throughput under high traffic conditions
- MDBs consume JMS messages
 - Select the right acknowledgment mode
 - Use `auto_acknowledge` mode to avoid duplicates
 - Avoids ineffective use of network bandwidth
 - Throughput may suffer when lots of messages arrived together
 - Use `dups_ok_acknowledge`

- For large amounts of data
 - e.g. searching a large table
 - Use JDBC directly rather than entity beans
 - If using JDBC 3.0, use `CachedRowSet`
 - Disconnected rowset functionality
 - Boosts performance by not maintaining connection to database while traversing the target set of data
- To ensure that connections are returned to the pool, always close the connection after use

- In BMP use prepared SQL statements
 - For high statement cache hit ratios
 - Each open statement corresponds to an open database cursor
 - Ensure close is called when no longer required
- Use batch updates where possible
 - Use `executeUpdate()`
 - Turn auto commit off `setAutoCommit(false)`
 - Transaction only committed when `commit()` called
 - Otherwise a roundtrip to the database is required for each `executeUpdate()` call

- Batch retrievals
 - Use if component issues separate SQL statements that fetch large amounts of data
 - Set fetch size on Statement to a large number
 - Reduces calls via JDBC driver to database
 - Also consider system resources when setting fetch size

- Use appropriate Statement API
 - **Statement** – to execute SQLs with no input/output parameters
 - **PreparedStatement** – to execute SQLs with input parameters only
 - **CallableStatement** – to execute SQLs with input/output parameters

- Transactions should not encompass user input or user think time
 - avoids holding resources unnecessarily
- CMT usually provides better performance
- Explicitly declare transaction attributes
 - E.g. use 'NotSupported' or 'Never' for non-transactional EJB methods
- For large transaction chains use 'TX_REQUIRED'
 - Ensures all EJB methods in the call chain use the same transaction

- Use lowest cost locking available for database
 - Appropriate for required level of transaction consistency
- Use XA capable data sources
 - only when two or more data sources are involved in the transaction
- Use two connection pools
 - If a database participates in both global and local transactions
 - One for global and one for local
 - Use appropriate pool in your application

- Isolation levels help maintain integrity of concurrently accessed data
- Isolation levels are set at the database connection or connection pool level
 - Given isolation level setting applies to all database access via that connection pool

- Use lowest possible isolation level
- **READ_UNCOMMITTED**
 - When beans represent operations on data which are not critical from integrity standpoint
 - Zero locking, Zero cost
- **READ_COMMITTED**
 - Applications that always read committed data
 - Cost: database server locks the data, returns it to your application, and then releases the lock on data

- REPEATABLE_READ
 - Applications that intend to always read and re-read the same data
 - Cost: others can concurrently read data being accessed but cannot modify it
 - Others can add new data
- SERIALIZABLE
 - Applications that want to hold exclusive access to data
 - Cost: others cannot read or modify the data being accessed concurrently
 - Other requests for data being accessed by your application will be serialized by the database

- Connection queue setting
 - Refers to number of sessions in the queue and average delay before connection is accepted
 - Current, peak and limit settings
 - If peak value is close to limit, increase maximum connection limit
 - avoids dropping connections under heavy load
- Acceptor threads setting
 - These place connections in queue where they are picked up by worker threads
 - Rule of thumb: one acceptor thread per CPU

- Persistent (Keep-alive) connections setting
 - Supports the ability to send multiple requests across a single HTTP session
 - Avoids overloading server with numerous connections
 - Tune number of threads in a keep-alive system
 - Tune keep-alive timeout
 - Specify time in seconds before an idle keep-alive connection is closed by server
 - Tune maximum number of “waiting” keep-alive connections in your server

- Cache static content
 - Improves response of server for static content
 - Tune max number of cache entries allowed in server
 - Tune max age
 - Ensures valid state of cached information
 - Optimizes usage of cached content

- Figuring out why your system runs slow
- Symptoms can include
 - Low CPU utilisation in a tier
 - One or more processors busy
 - High system CPU time in one tier
 - Unusual disk activity at business or data tier
 - Poor response time distribution
- Use tools to get more data
 - OS tools like `truss`, `*stat`
 - Profiling tools like Sun Studio

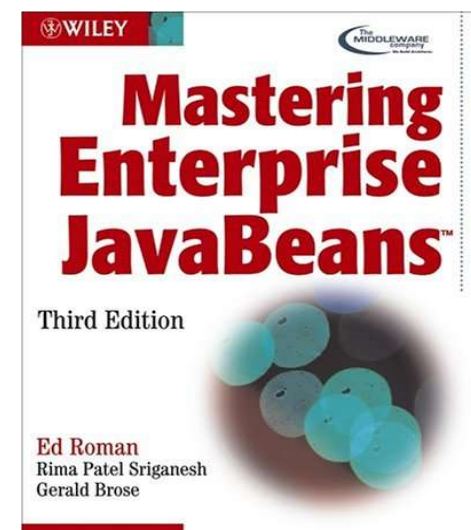
- Avoid excessive directories in the server CLASSPATH
 - Improves class loading time.
- Package application related classes into JAR file
- Set recompile reload interval
 - Prevents JSP recompilation
 - In Sun Java System Application server this value is -1
- Deploy applications that do not contain EJBs as WAR instead of EAR files

- Understanding the virtual machine will help you tune performance
- Use profiling tools to find bottlenecks
- Adapt HotSpot parameters to your application
- Always use the latest JRE
- Sun is always improving Java performance

- J2EE application performance is dependent upon many layers
- Use monitoring tools at all levels
 - Application
 - Application server
 - JVM
 - System layer
- Good performance is also equally dependent on good coding

- java.sun.com/blueprints/performance
- java.sun.com/products/hotspot
- profiler.netbeans.org/index.html
- developers.sun.com/dev/coolstuff/jvmstat
- developer.java.sun.com/developer/technicalArticles/Programming/GCPortal

- java.sun.com/performance
- javaperformancetuning.com
- Mastering Enterprise Java Beans, Ed Roman
 - Design best practices





J2SE™ & J2EE™ Performance: Learn How to Write High Performance Java Applications

Matt Hosanee

Partner Engineering

Sun Microsystems

