High Performance Java Technology in a Multi-Core World

David Dagastine

Paul Hohensee

VM Technologies

Sun Microsystems, Inc.
http://java.sun.com

TS-2885

What To Expect

Learn about today's multi-core architectures and the Java™ Virtual Machine (JVM™ Tool) technologies and software optimization techniques to make your application run its fastest.

The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java™ platform.

Agenda

Multi-Core Computer Architectures

JVM Tool Technologies and Optimizations

Application and JVM Tool Tuning

Demo—Identifying Common Bottlenecks

Agenda

Multi-Core Computer Architectures

JVM Tool Technologies and Optimizations

Application and JVM Tool Tuning

Demo—Identifying Common Bottlenecks

Definitions

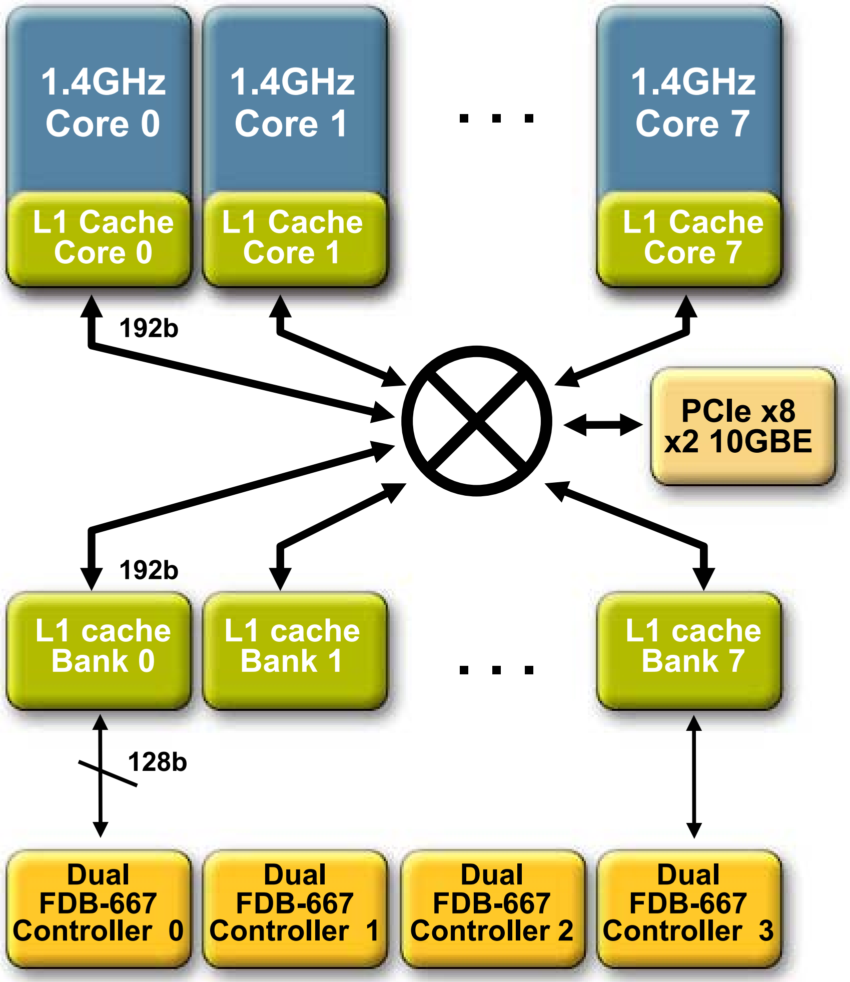
- CMT—Chip multithreading
  - Multiple hardware threads of execution (a.k.a. strands) per chip, through multiple cores, multiple threads per core or a combination of both
  - SPARC® US-T1 processor and US-T2 processor (aka Niagara 1 and 2)
  - Rock
- CMP—Chip multiprocessing
  - Multiple cores per chip
  - Intel® Core2
  - AMD Opteron™
  - US-IV+

Definitions

- SMT—Simultaneous multithreading
  - Issue multiple instructions from multiple threads in one cycle on one core
- HT—Hyperthreading
  - Two-thread SMT
  - IBM™ Power
  - Intel Xeon

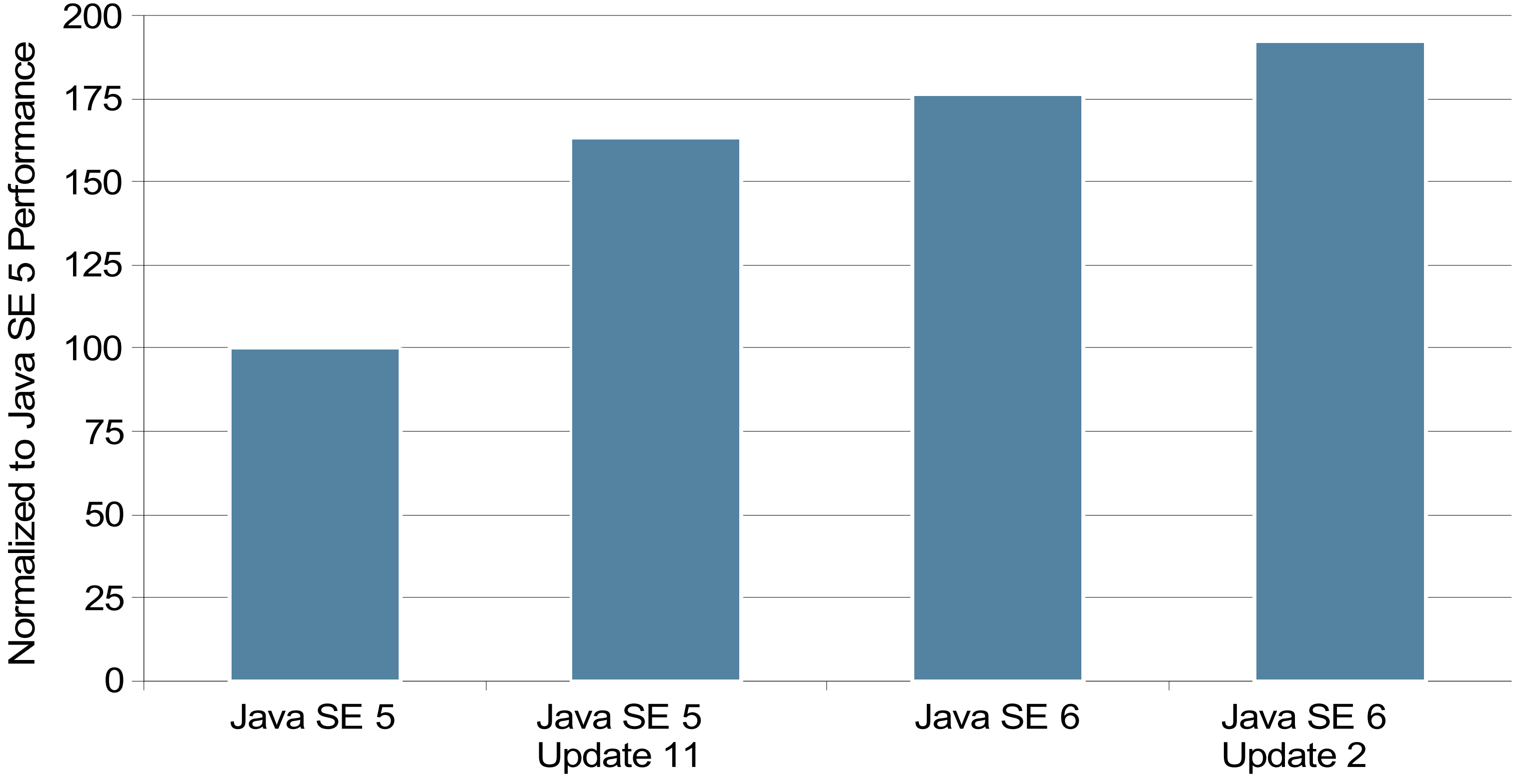
Niagara-2 System Diagram

- 8 SPARC processor cores, 8 threads each
- Shared 4MB L2, 8-banks, 16-way associative
- Four dual-channel FBDIMM memory controllers
- Two 10/1 Gb Enet ports with onboard packet classification and filtering
- One PCI-E x8 1.0 port



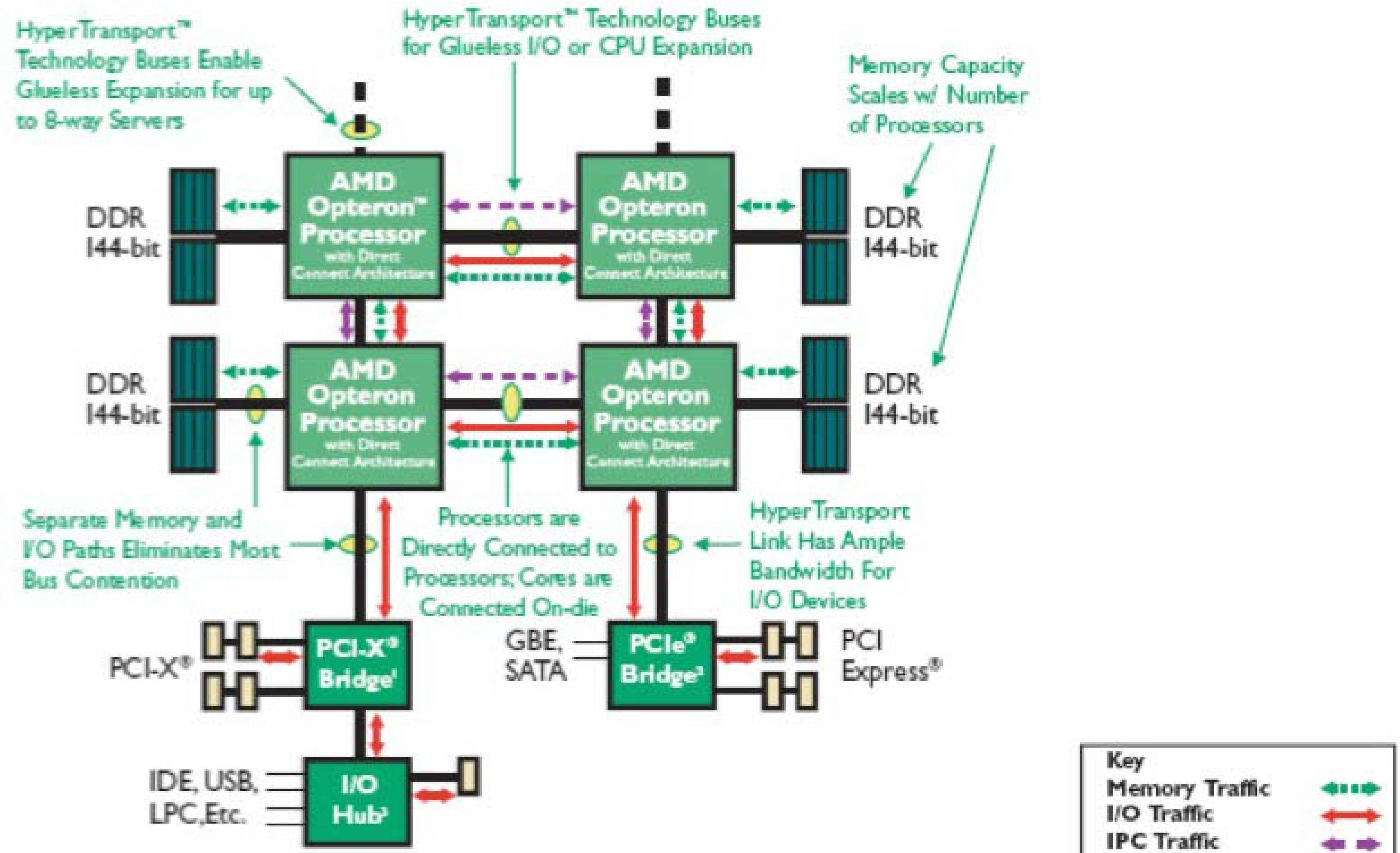
JVM Tool Performance on UltraSPARC® T1

SPECjbb2005 on Sun Fire™ T2000 Server: 1 x 32 x1.2 Ghz US-T1



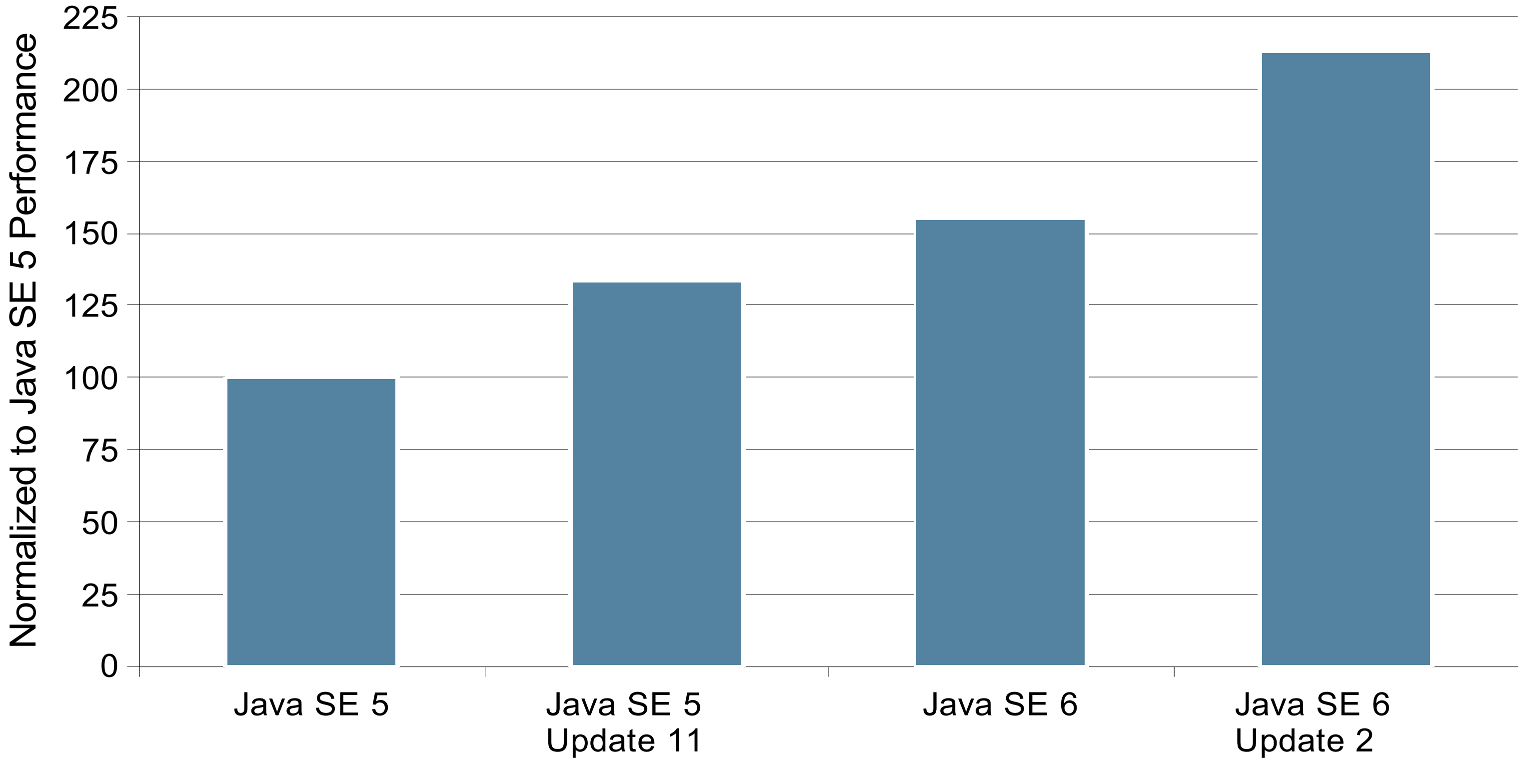
Source: Sun Microsystems, Inc.
Run on a Sun Fire T2000 Server: 1 x 1.2Ghz US-T1, 16GB RAM
SPECjbb2005 are trademarks of the Standard Performance Evaluation Corporation. For the latest SPECjbb2005 benchmark results, visit http://www.spec.org/osg/jbb2005.
Java SE = Java Platform, Standard Edition (Java SE platform)

Four Socket AMD Opteron Overview



JVM Tool Performance on AMD Opteron

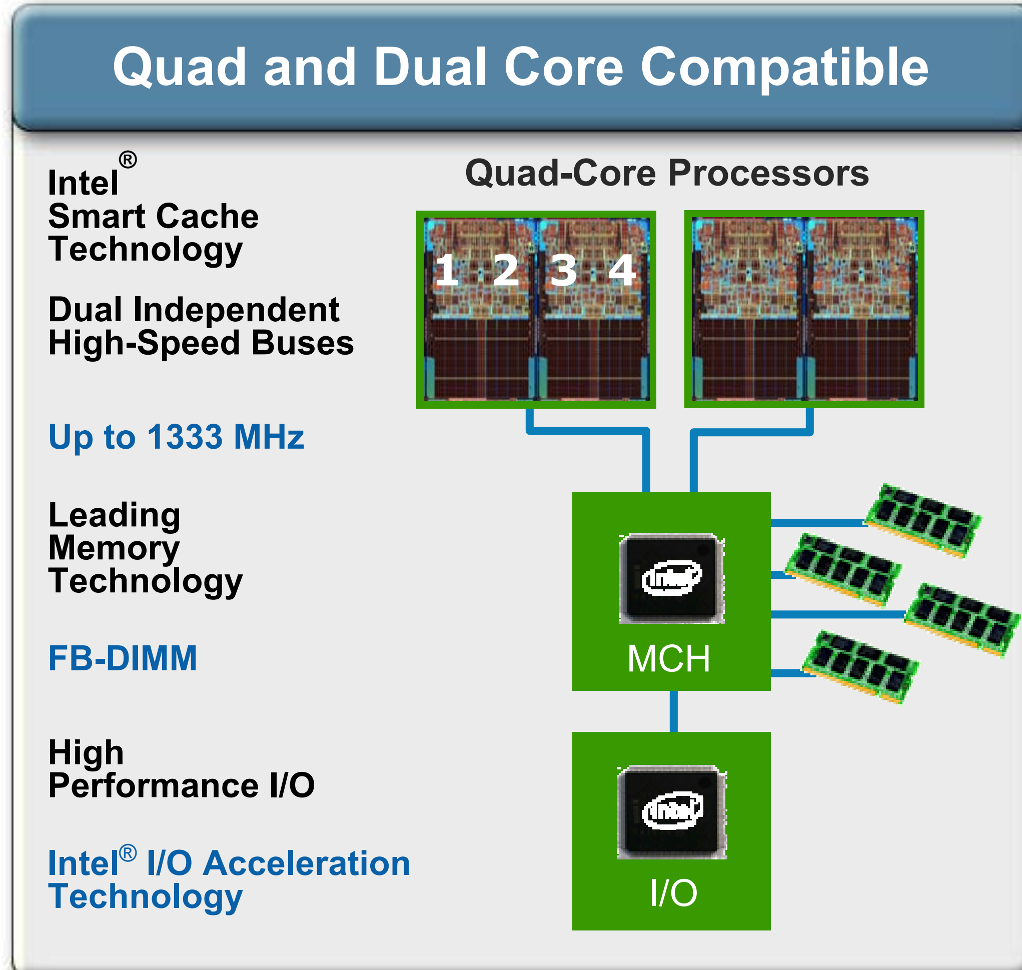
SPECjbb2005 on Sun X4600: 4 x 2 x 2.8 Ghz Opteron



Source: Sun Microsystems, Inc.
Run on a Sun Fire X4600: 4 x 2.6Ghz AMD Opteron Dual-core , 64GB RAM
SPECjbb2005 are trademarks of the Standard Performance Evaluation Corporation. For the latest SPECjbb2005 benchmark results, visit http://www.spec.org/osg/jbb2005.

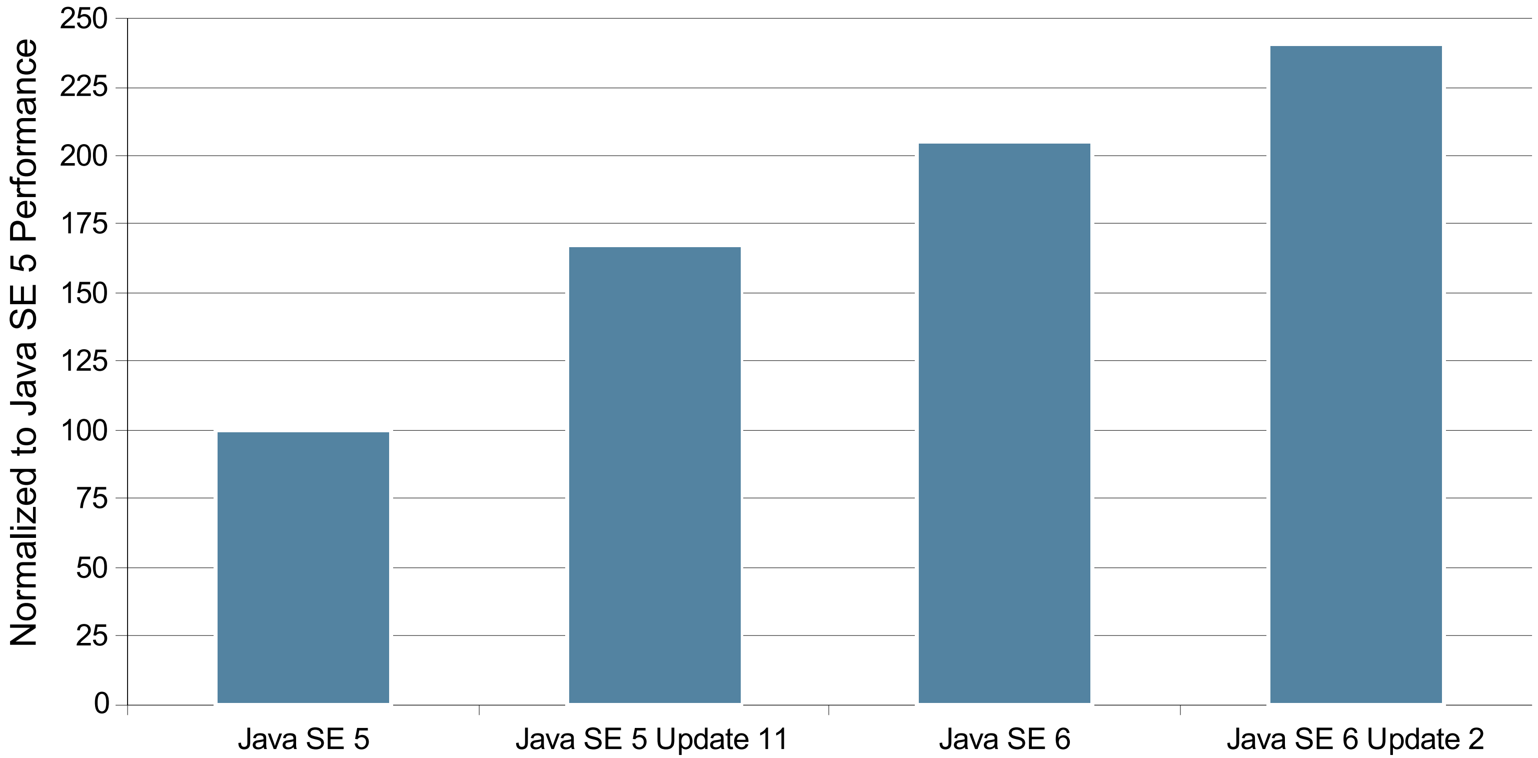
Quad-Core Intel Xeon

- Intel Wide Dynamic Execution
- Intel Advanced Digital Media Boost
- Intel Smart Memory Access
- Intel Intelligent Power Capability
- Large 8mb L2 Cache
- Socket Compatible: Dual-core to quad-core
- Power Efficient



JVM Tool Performance on Intel Xeon

SPECjbb2005 on 2 x 2 x 2.6 Ghz Intel Xeon



Source: Sun Microsystems, Inc.
Run on 2 x 2.6Ghz Intel X5355 Core 2 Quad , 8GB RAM
SPECjbb2005 are trademarks of the Standard Performance Evaluation Corporation. For the latest SPECjbb2005 benchmark results, visit http://www.spec.org/osg/jbb2005.

Agenda

Multi-Core Computer Architectures

JVM Tool Technologies and Optimizations

Application and JVM Tool Tuning

Demo—Identifying Common Bottlenecks

Optimization for Multi-Core Systems

- Leverage Plentiful Hardware Threads
  - Increase Throughput
    - Java platform (and JVM tools) are inherently multi-threaded
    - Threading using the Java platform is easy!
    - java.util.concurrent
    - Parallel Garbage Collection
  - Improve determinism
    - Concurrent Garbage Collection
    - Sun Java Real-Time System (Java RTS)
  - Both
    - Concurrent/Parallel Garbage Collection
    - Concurrent/Parallel Dynamic Compilation
    - Concurrent/Parallel Classloading

Optimization for Multi-Core Systems

- All those hardware threads pound memory, so must optimize memory system use
- Overcome latency (time to fetch data from memory) and bandwidth (amount of data transferred between memory and processor in a given time) limitations
- Processor/memory affinity
  - Always run a given software thread on the same hardware thread
  - Keeps data “close” to processor (caches warm)
  - OS does its best
    (with your help: binding, processor sets)

Optimization for Multi-Core Systems

- Number of simultaneously active software threads should be ≥ number of hardware threads
  - But may be less due to memory system limitations
  - Plan to use all the hardware threads
  - Include non Java threads in the count: Concurrent GC, native threads
- Minimize writes to shared data
  - Processor must acquire data ownership, which usually means a write to plus a read from long-latency memory
  - Synchronization requires write to shared lock word
  - Reads of shared data are okay

Optimization for Multi-Core Systems

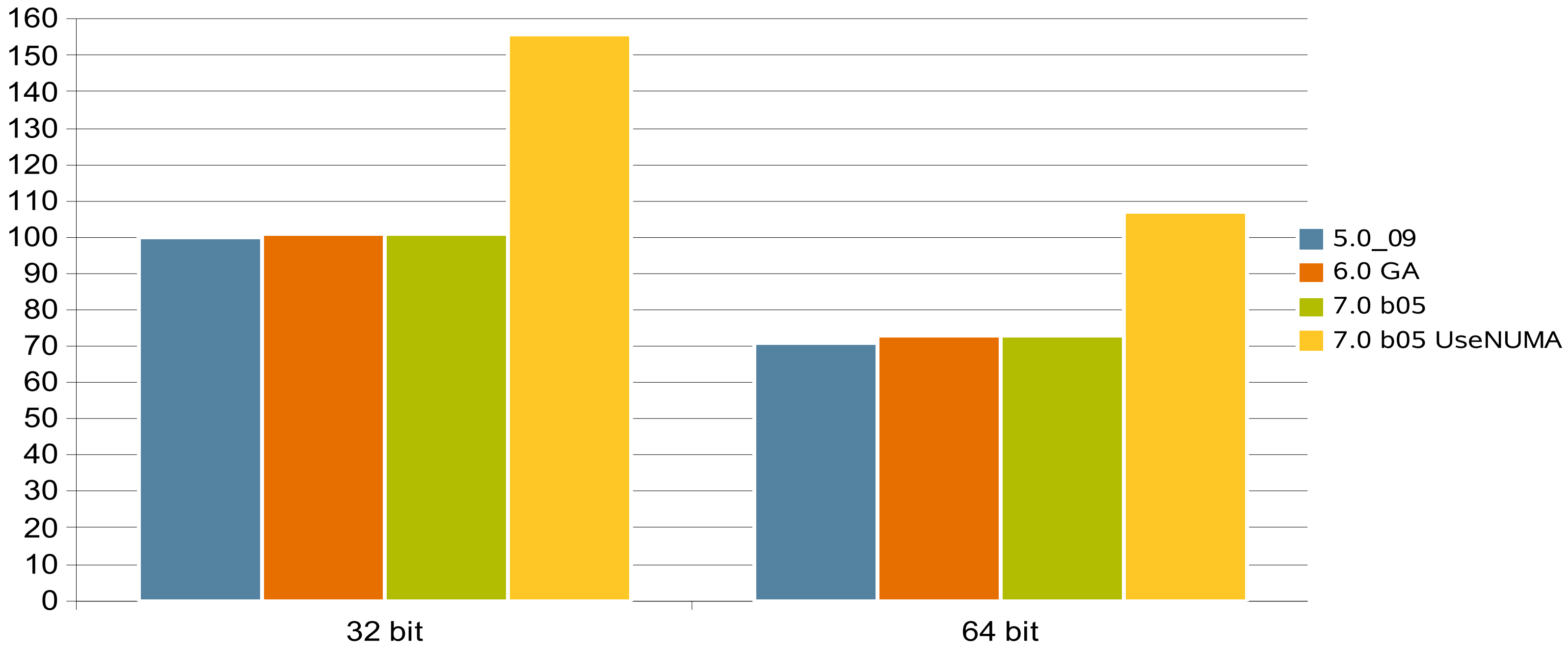
- How important are memory system optimizations for your hardware?
- Optimization effectiveness depends on latency/ bandwidth ratios in the memory hierarchy
  - Shared cache(s), local and remote memory
- Likely effectiveness, most to least
  - Sun US-VI+ (E25K) L1, L2, L3, *local memory, remote memory
  - Intel Core2 L1, *L2, memory
  - AMD Opteron L1, L2, *local memory, remote memory
  - Sun US-T1 L1, *L2, memory

JVM Tool Optimizations: Affinity

- Thread-Local Allocation Buffers (TLABs)
  - Java threads allocate objects in thread-private memory
  - Otherwise app serializes on shared heap access
- Parallel Thread-Local Allocation Buffers (PLABs)
  - GC threads copy live objects to thread-private memory
- NUMA-aware allocators
  - Chip- and/or board-local allocation regions: TLABs and PLABs writ large
  - Associate Java and GC threads with a region
  - Collect all regions when one becomes full
  - Depends on OS affinity mechanisms

NUMA-aware SPECjbb2005

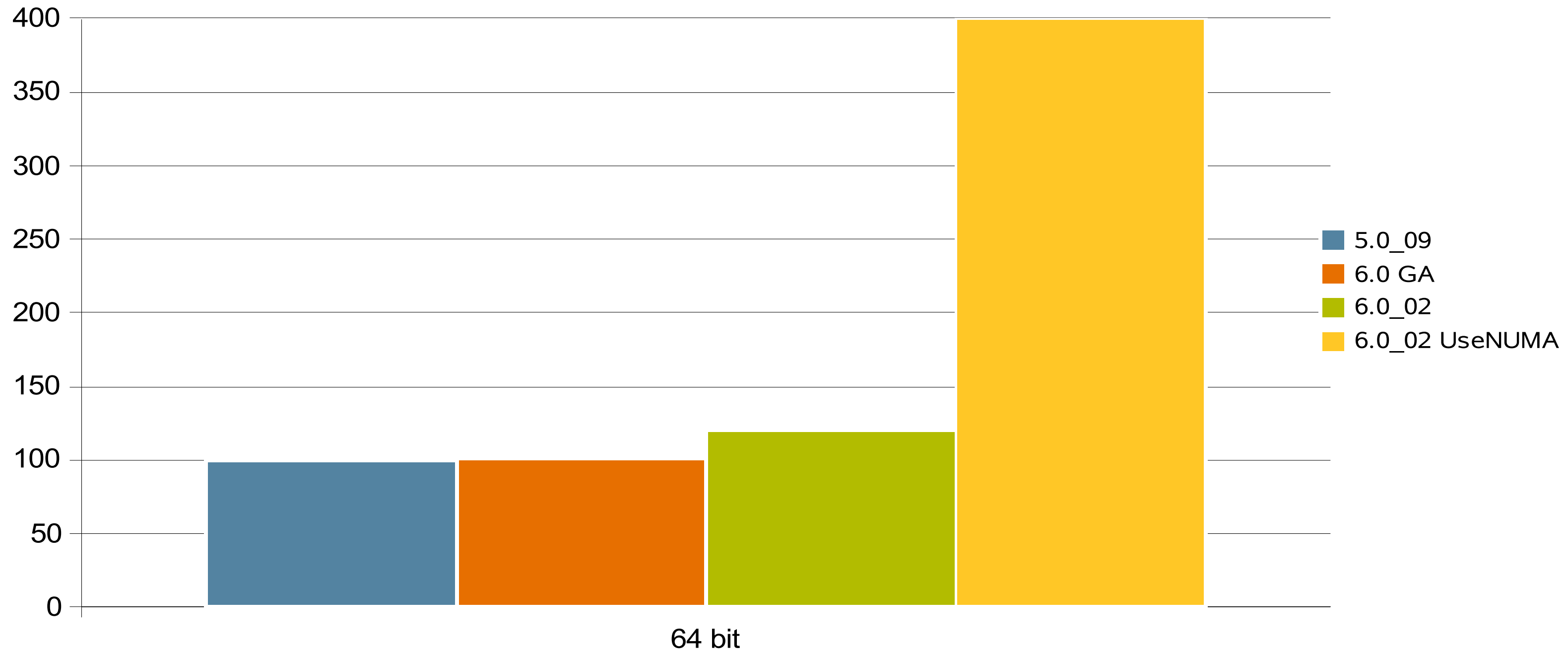
16 core, 8 chip Opteron: Sun X4600, Solaris™ 10u3



Source: Sun Microsystems, Inc.
SPECjbb2005 are trademarks of the Standard Performance Evaluation Corporation. For the latest SPECjbb2005 benchmark results, visit http://www.spec.org/osg/jbb2005.

NUMA-aware SPECjbb2005

144 core, 72 chip, 18 board UltraSPARC-IV+: Sun Enterprise™ E25K, Solaris 10u3



Source: Sun Microsystems, Inc.
SPECjbb2005 are trademarks of the Standard Performance Evaluation Corporation. For the latest SPECjbb2005 benchmark results, visit http://www.spec.org/osg/jbb2005.

JVM Tool Optimizations: Affinity/Bandwidth

- Copying garbage collectors can scatter objects around memory that were originally allocated next to each other in TLABs
- Objects allocated together are usually accessed together, so scattering causes extra memory traffic
- Object copying order can be
  - Breadth-first: Copy all children, then all children's children
  - Depth-first: Copy first child, then first-child's first child,… then second child,…
  - Some combination of the two
- Which is better is app-dependent, but for most applications depth-first is better

JVM Tool Optimizations: Latency (1)

- Allocation prefetch
  - Prefetch instructions can acquire cache line ownership for a processor in time for later writes
  - Allocate space in cache for the acquired line
  - When allocating objects linearly in TLABs, prefetch a platform-dependent distance ahead of address of the object being allocated
  - Subsequent allocations should find line already cached
  - Sometimes it's a good idea to prefetch multiple cache lines ahead

JVM Tool Optimizations: Latency (2)

- Processors cache virtual-to-physical address translations in Translation Lookaside Buffers (TLBs)
- TLB size is limited, typically 8 to 64 entries
- TLB miss is expensive
  - Requires walking page table in memory
- Modern processors support large pages
  - 2 to 4 mb rather than 4 to 8 kb
  - Can map memory with many fewer TLB entries
- JVM tool can map Java heap and generated code cache with large pages
- Far fewer TLB misses

JVM Tool Optimizations: Bandwidth

- Object field reordering
  - Group frequently accessed fields together so they end up in minimum number of cache lines
  - Often with object header
  - Experience shows that scalar fields should be grouped together separately from object reference fields
- Vectorization
  - Load, operate on and store multiple array elements at once with single machine instructions
  - e.g., use 8- or 16-byte loads and stores to access 4 or 8 char array elements at a time
  - Compiler-generated or tailored assembly code: e.g., System.arraycopy

JVM Tool Optimizations: Latency/Bandwidth (1)

- 64-bit JVM tools enable heaps larger than 4 gb, but are ~20% slower than 32-bit JVM tools
- Essentially all of the difference is due to extra memory system pressure caused by moving 64-bit pointers around
- Solution: Use 32-bit offsets from a Java heap base address instead of 64-bit pointers
- If objects are highly aligned, can use > 4 gb heaps
  - If objects are 8-byte aligned, a 32-bit object offset can represent a 35-bit byte offset => 32 gb Java heap
- On some platforms (x64), resulting 64-bit JVM tool can be faster than 32-bit equivalent!

Agenda

Multi-core Computer Architectures

JVM Tool Technologies and Optimizations

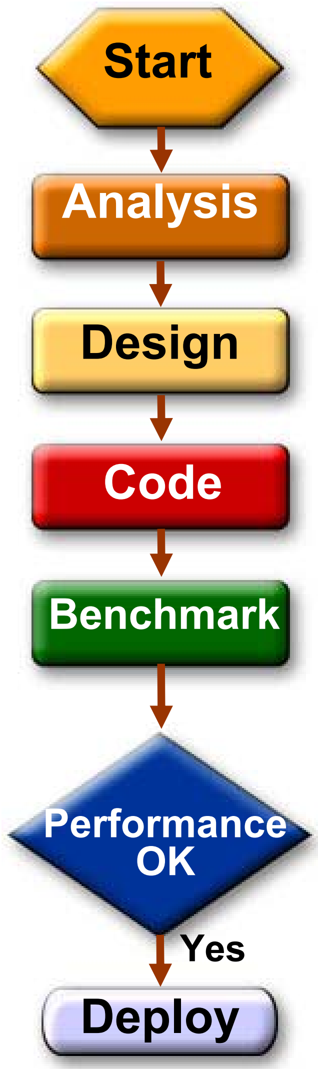
Application and JVM Tool Tuning

Demo—Identifying Common Bottlenecks

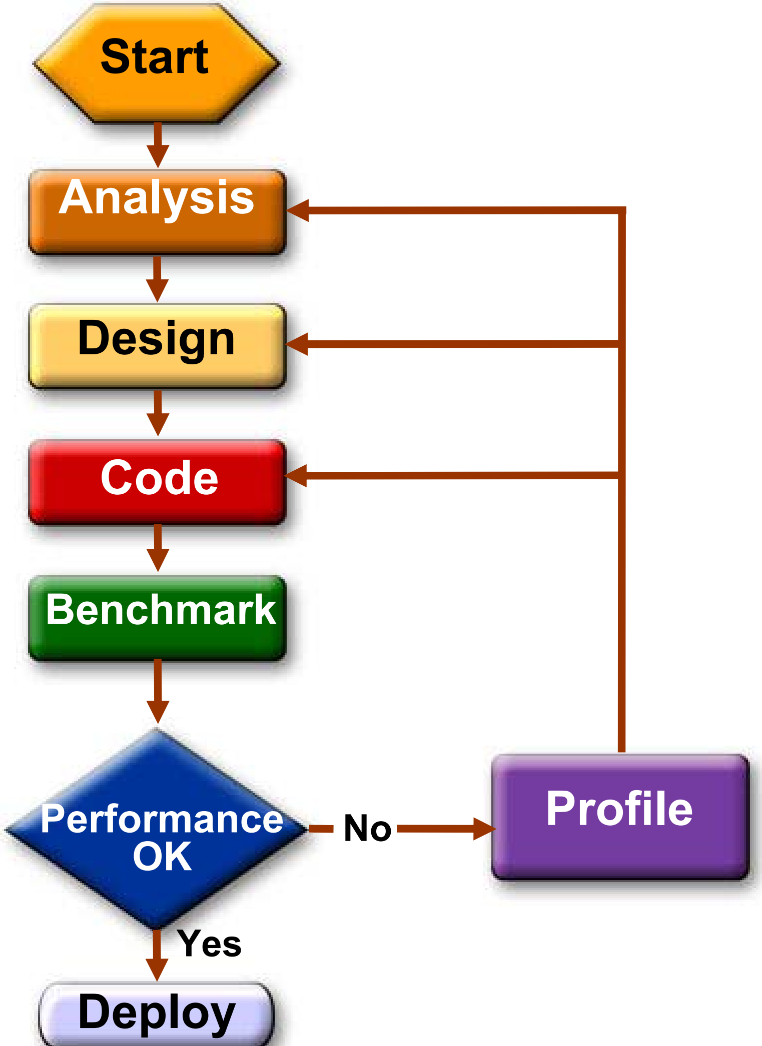
What's Tuning?

- The process of making an app run well on a particular platform
- Use the JVM tool to help
- Trust the JVM tool (but verify)
  - Don't warp your source code to compensate for perceived JVM tool problems
  - JVM tools constantly improve
  - They optimize for the common case
  - Warped source code eventually becomes a performance liability
- http://java.sun.com/performance/reference/whitepapers/tuning.html

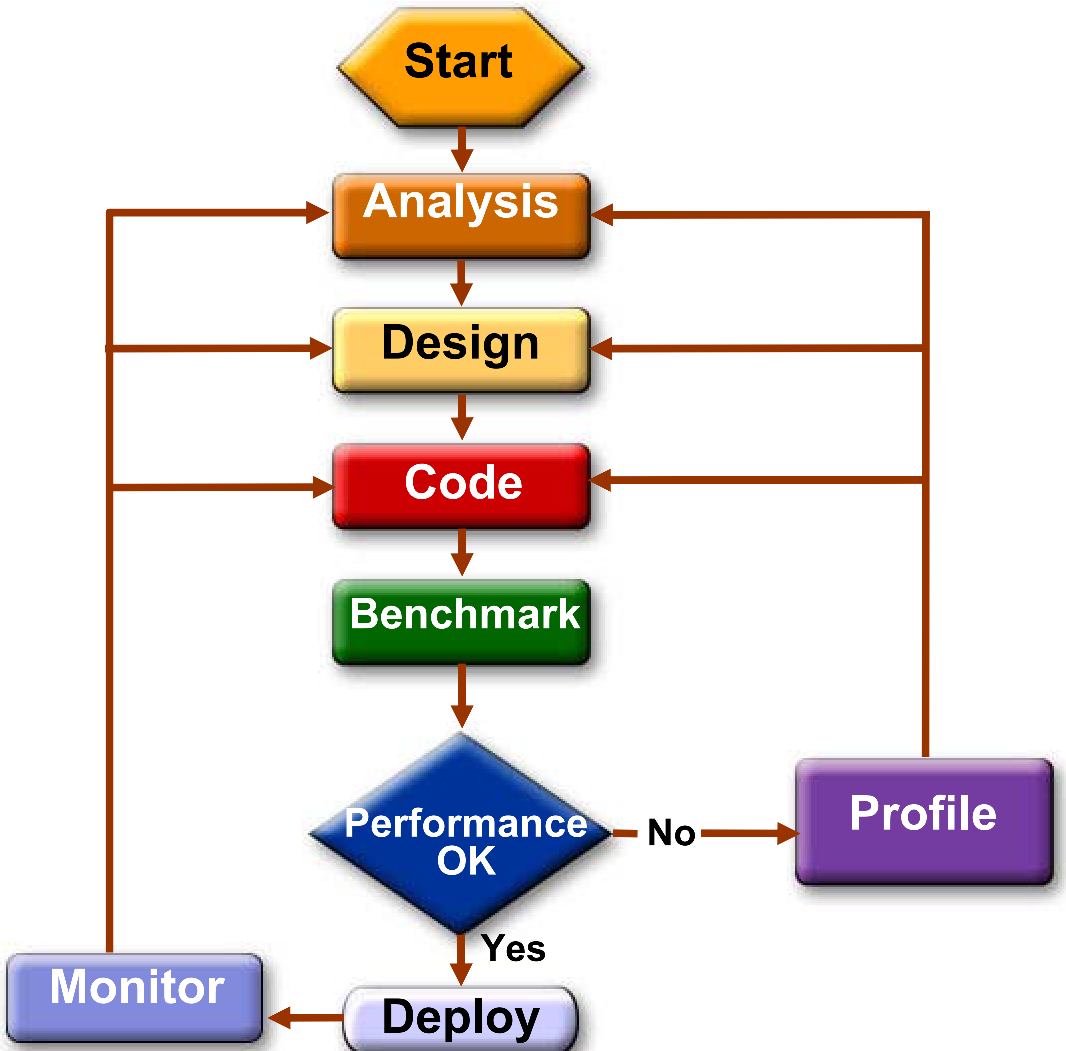
Typical Development Process



Application Performance Process



Application Performance Process



Fastest JVM Tool? Which Benchmark?

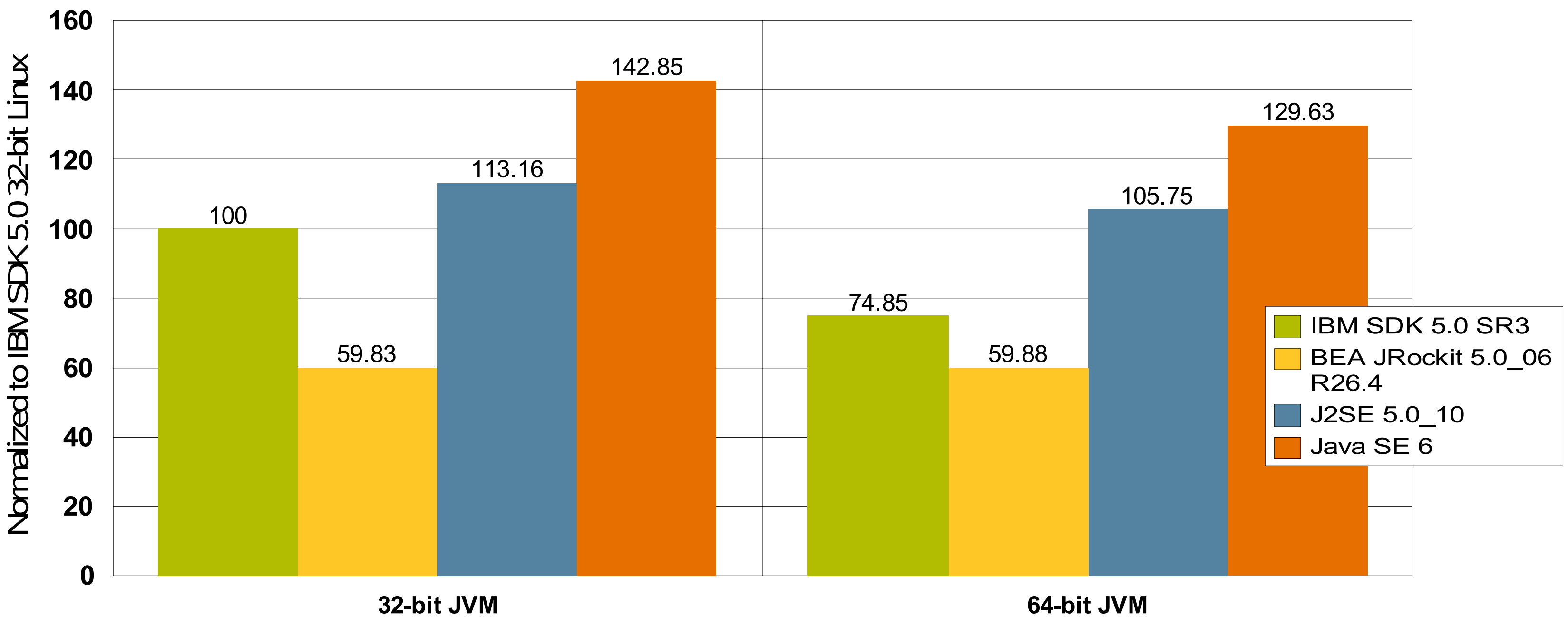
Competitive analysis by % of total submissions

	SPECjbb2005	SPECjAppServer2004
BEA	62%	9%
SUN	22%	47%
IBM	11%	20%
HP	2%	24%

Source: http://www.spec.org

SPECjbb2005: Out-of-Box Performance

Intel Core2 Processor Microarchitecture:
2 x 2 x 3.0 Ghz Intel 5160 (4-core)



Source: Sun Microsystems, Inc., Full Disclosure on http://blogs.sun.com/dagastine
Run on a 2 x 3.0 Ghz Intel 5160 (4-core), 8GB RAM
SPECjbb2005 are trademarks of the Standard Performance Evaluation Corporation. For the latest SPECjbb2005 benchmark results, visit http://www.spec.org/osg/jbb2005.
J2SE = Java 2 Platform, Standard Edition (J2SE™ platform)

Common Bottlenecks

Three general categories

- Excessive Allocation
  - Increased pressure on GC and memory systems
- Synchronization
  - Serialization in your application will limit scalability
- Untuned Java heap configuration, including collector selection

Reduce Object Allocation Rate

Steps to identify hot allocation sites

- Profile
  - Netbeans™ Profiler Module (coming next in Demo)
  - HPROF
- Identify alternate strategies
  - Thread-local variables?
- If unable to reduce allocation rate, then tune

JVM Tool Tuning for High Allocation Rates

Steps to tune the JVM Tool

- Observe GC behavior using VisualGC or jconsole
- Tune Java heap and generation sizes
  - Increase overall heap and young generation sizes
  - Large heaps need a parallel collector
- Tune TLABs
  - Not necessary with Sun's HotSpot™ JVM tool
  - Tune allocation prefetch
  - Again, not necessary with HotSpot JVM tool

Identify Synchronization Bottlenecks

Steps to identify hot locks

- Profile
  - Netbeans Profiler Module (Coming next in Demo)
  - HPROF
- OS CPU statistics
  - High mutex spin count
  - High context switch rates
  - Unable to leverage 100% of CPU
- Identify alternate strategies
  - Maintain thread affinity
  - Use java.util.concurrent

Basic Java Platform Heap Tuning

First steps to tuning GC

- Observe GC behavior
  - -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
  - jvmstat, VisualGC, jconsole
- Identify proper heap size
  - -Xms -Xmx
- Identify proper young generation size
  - -Xmn -XX:NewSize= -XX:MaxNewSize=

JVM Tool Throughput Tuning

Tuning parameters and why we use them

- -XX:+UseParallelOldGC
  - Minimize garbage collection impact on throughput
  - Does not target low pause times (Use CMS for that)
- -XX:ParallelGCThreads=<*n*>
  - *Large-scale deployment running multiple JVM tools*
  - *By default, = #hardware threads*
  - *On US-T1, total GC threads on the system should not exceed 20: You might have expected 32*
  - *Experiment needed, mileage will vary*

JVM Tool Throughput Tuning

Tuning parameters and why we use them

- -XX:+UseBiasedLocking(5–10%)
  - On by default in Java SE v.6 platform
  - Bias synchronized object to the thread that created it
  - If the synchronized block is never accessed by another thread, uses cmp+branch, not atomics, to lock/unlock
  - +3% on US-T1; CAS is cheap
- -XX:+AggressiveOpts (+5–10%)
  - New wrapper flag for performance optimizations
  - Features will be enabled by default in upcoming releases
  - Code quality optimizations, not GC

JVM Tool Low Pause Time Tuning

Key parameters for CMS

- -XX:NewRatio=N -Xmn -XX:[Max]NewSize
- -XX:SurvivorRatio=
- -XX:MaxTenuringThreshold=
  - Smaller young generation can put more pressure on CMS old generation
  - Larger young generation can increase young generation pause times
  - Experiment

JVM Tool Low Pause Time Tuning

Key parameters for CMS

- -XX:ParallelCMSThreads=<n>
  - Dynamically set based on ParallelGCThreads
- -XX:ParallelGCThreads=<n>
  - Default: number of hardware threads (ncpus)
  - Try ncpus = ncpus ≤ 8 ? ncpus : ncpus * 5/8;
- -XX:CMSInitiatingOccupancyFraction=<n>
  - Old gen occupancy at which CMS starts collecting
    - Larger values improve throughput and Full GC risk
    - Lower values reduce throughput and Full GC risk

Using Large Pages

How to enable on Solaris

- Enabled by default: It just works
- Default page size is 8k on SPARC,4k on x64
- Large page size on US-III and US-IV systems 4m
- US-IV+ supports 32mb pages
  - -XX:LargePageSizeInBytes=32m
- US-T1 supports 256mb pages
  - -XX:LargePageSizeInBytes=256m
- X86 (Intel and AMD) supports 4mb pages
- X64 (AMD and Intel) supports 2mb pages

Using Large Pages

How to enable on Windows

- Use the local security settings console to “lock pages in memory” for the user running the application
- -XX:+UseLargePages

For more detailed information:

- http://java.sun.com/docs/hotspot/VMOptions.html#largepages

Using Large Pages

How to enable on Linux

Bear with me, this will take awhile

0. 1. Create huge page folder
   - mkdir/mnt/hugepages
1. Mount the huge page file system
   - mount -t hugetlbfs nodev/mnt/hugepages
2. Set permissions for read and write on the folder for the user/users that will use huge pages; By default only root will have access after mounting In this example, all users will be allowed
   - chmod 755/mnt/hugepages
   - chmod 777/mnt/hugepages

Using Large Pages

How to enable on Linux

4. 4. Specify how many pages you want to allocate as large pages:
   - Echo 1500 > /proc/sys/vm/nr_hugepages
5. Verify result:
   - cat /proc/meminfo | grep -E "(HugePage|Hugepage|Mem)"
6. -XX:+UseLargePages

For more detailed information:

- http://java.sun.com/javase/technologies/hotspot/largememory.jsp

NUMA Optimizations

How to enable on Solaris™

- Single JVM tool
  - New in JDK™ 6 Update 2: use the NUMA allocator
    - -XX:+UseNUMA
  - Prior to JDK 6 Update 2:
    - /etc/system: lgrp_mem_default_policy=3
- Multiple JVM tools
  - Use processor sets (See man page for psrset)
  - Significant (5–10%) on x64 and US-IV+ System
  - lgrp_mem_pset_aware=1
    - Default random policy applies only to lgroups with a process' processor set

NUMA Optimizations

How to enable on Linux

- Single JVM tool
  - numactl –interleave
- Multiple JVM tools
  - numactl—cpubind=$node_num—embind=$node_num
- Significant performance gains on x64 systems

NUMA Optimizations

How to enable on Windows

- Single JVM tool
  - AMD Opteron: enable node-interleaving in the BIOS
- Multiple JVM tools
  - Use Processor Affinity (similar to numactl on Linux)
    - Bring up task manager select the processes tab, select a process and right click—You will get a popup, select Processor Affinity
- Significant performance gains on x64 systems

DEMO

Identify Common Bottlenecks

Q&A

Additional Info

Performance References (1)

- General
  - http://java.sun.com/javase/technologies/hotspot/index.jsp
  - http://java/sun/com/javase/technologies/performance.jsp
  - http://java.sun.com/performance/reference/whitepapers
  - http://java.sun.com/performance/reference/whitepapers/5.0_performance.html
- Tuning
  - http://java.sun.com/performance/reference/whitepapers/tuning.html
- GC
  - http://java.sun.com/javase/technologies/hotspot/gc/index.jsp
- java.net
  - http://www.java.net/

Performance References (2)

- Planet JDK
  - http://planetjdk.org
- Individual blogs
  - Dave Dagastine http://blogs.sun.com/dagastine/
  - Dave Dice http://blogs.sun.com/dave/
  - David Holmes http://blogs.sun.com/dholmes/
  - Jon Masamitsu http://blogs.sun.com/jonthecollector/
  - Steve Bohne http://blogs.sun.com/sbohne/
  - Steve Goldman http://blogs.sun.com/fatcatair/
  - Tony Printezis http://blogs.sun.com/tony/
  - Keith McGuigan http://blogs.sun.com/kamg/

JVM Tool Optimizations: Latency/Bandwidth (2)

- Escape analysis
  - An object ***escapes*** the thread that allocated it if some other thread can ever see it
- If an object doesn't escape, we can do
  - Object explosion: Allocate an object's fields in different places
  - Scalar replacement: Store scalar fields in registers
  - Thread stack allocation: Store fields in stack frame
  - Eliminate synchronization
  - Eliminate GC read/write barriers
- Memory system pressure reduced, possibly eliminated

JVM Tool Optimizations: Latency/Bandwidth (3)

- Generic collection classes introduce hidden promotion of scalars to objects, a.k.a. “auto-boxing”
  - HashMap.get(5) is transformed by javac to
  - HashMap.get(Integer.valueOf(5))
- Smart compilers can eliminate the object allocation entirely and/or use the scalar value instead of accessing the object once it’s allocated

JVM Tool Optimizations: Synchronization (1)

- Most locking is uncontended
  - Avoid associating an OS mutex (heavy-weight lock) with each object
  - While uncontended, use light-weight mechanism(s) to enter/exit monitor
  - If contended, fall back to heavy-weight lock
- Detecting contention requires atomic write to shared lock word, usually via compare-exchange instruction
  - Must complete all memory operations to first level of memory shared by all processors
  - Must acquire lock word ownership
  - Takes 10s to many 100s of cycles

JVM Tool Optimizations: Synchronization (2)

- Typically two light-weight mechanisms
- Start with biased locking
  - Avoids lock word contention when a lock is owned by only a single thread over long periods of time
  - Single compare-exchange biases lock toward a thread
  - Thereafter, compare-and-branch for monitor entry/exit
  - Typically very expensive to transfer bias to another thread
- If lock ownership starts changing frequently, but still without contention

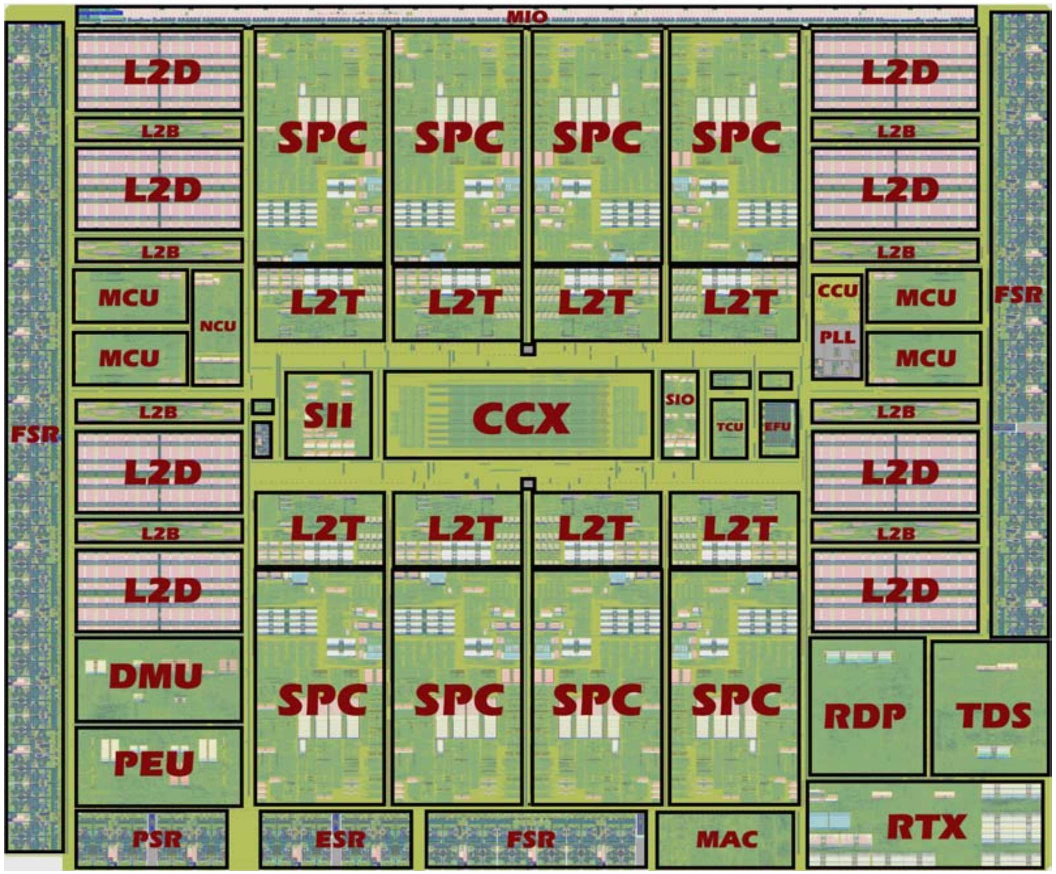
JVM Tool Optimizations: Synchronization (3)

- Switch to compare-exchange for monitor entry/exit
  - More expensive, but still far cheaper than OS lock
- If real contention occurs (One thread wants to acquire a lock held by another), try desperately to avoid heavy-weight lock
  - System calls for monitor entry/exit take 1000s of cycles
- Adaptive spinning
  - Spin awhile, then retry lock acquisition
  - Locked region usually short, lock likely released during spin
  - Platform and execution history-dependent spin time, otherwise cost exceeds benefit

JVM Tool Optimizations: Synchronization (4)/Affinity

- If in spite of all this a lock becomes heavy-weight, bias selection of next thread to acquire lock
- On monitor exit, prefer running a blocked thread that has recently run on the same processor
  - Caches and TLB will be warm
- Locally unfair, but globally efficient
- Relies on OS to guarantee that every thread will eventually run

Niagara2 Overview



- 8 SPARC processor cores, 8 threads each
- Shared 4MB L2, 8-banks, 16-way associative
- Four dual-channel FBDIMM memory controllers
- Two 10/1 Gb Enet ports w/onboard packet classification and filtering
- One PCI-E x8 1.0 port
- 711 signal I/O, 1831 total

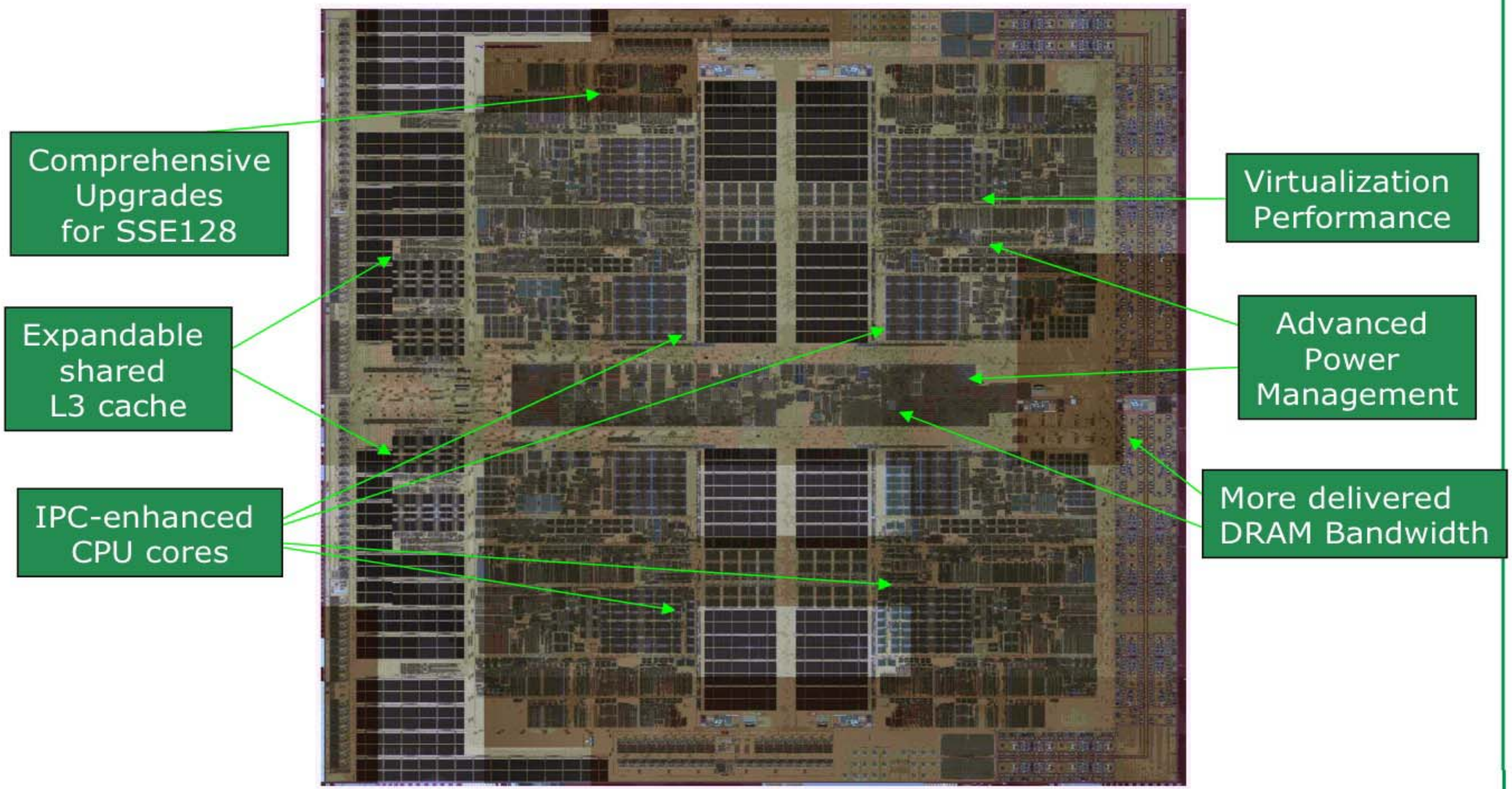
Niagara 2 Details

General purpose, high throughput, heavily threaded

- Niagara-2 combines all major server functions on one chip
- > 2x throughput and throughput/watt vs. UltraSPARC T1
- Greatly improved floating-point performance
- Significantly improved integer performance
- Embedded wire-speed cryptographic acceleration
- Enables new generation of power-efficient, fully-secure datacenters

Source: Please add the source of your data here

AMD Quad-core Overview



High Performance Java Technology in a Multi-Core World

David Dagastine

Paul Hohensee

VM Technologies

Sun Microsystems, Inc.
http://java.sun.com

TS-2885