




**ORACLE®**

## **Java for Trading Applications**

**Trading Applications Developer Workshop, London 2011**

Sten Garmark

Product Manager, Java Platform Group

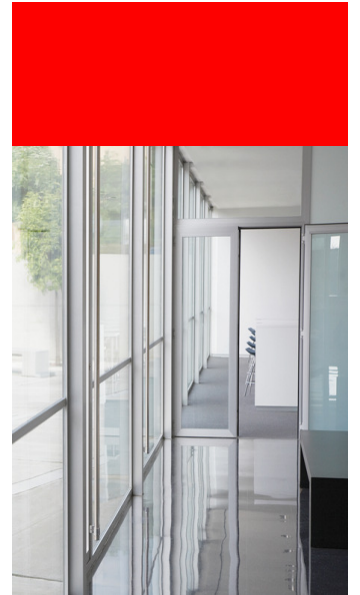


The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.



# Agenda

- Oracle's Java Strategy
- Why Java for Trading Applications?
- JDK 7 & 8 Features and Roadmap
- JVM Convergence & Roadmap
- Garbage Collection and Latency
- Tooling for Low Latency
- Exalogic – Hardware for Java
- The Stack - Engineered Together





# Oracle's Java Strategy

In Order of Priority

## 1. A Vibrant Java Ecosystem

- Oracle depends on Java for its software business. Keeping Java vibrant and competitive ensures business continuity.
- In other words: Our goals are aligned with the community – make Java successful!

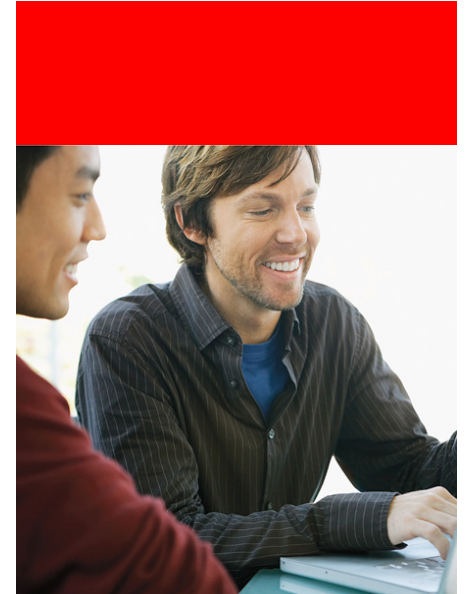
## 2. Generate Revenue

- Primarily indirectly by making Oracle software and hardware more competitive, but also directly through premium products like JRockit Real-Time and Mission Control

## 3. Lower Costs

- Improve language, libraries and tools in such a way that it makes Oracle's thousands of Java developers more productive.
- This will of course also help goal #1 in most cases

# Why Java for Trading Applications?





# Portability

- Flexibility to choose best hardware and OS platform



# State-of-the-art Runtime and Compiler

- Extremely efficient machine code
- Profile guided optimizations
- Engineering partnership with Intel
- On par or better than C/C++



# Time to Market

- Rich standard class library
- Rich ecosystem of commercial and open source components
- Access to bright developers
- Managed memory
  - No SIGSEGV bugs that weeks or months to fix
- Concurrency built in



# Competition means Innovation

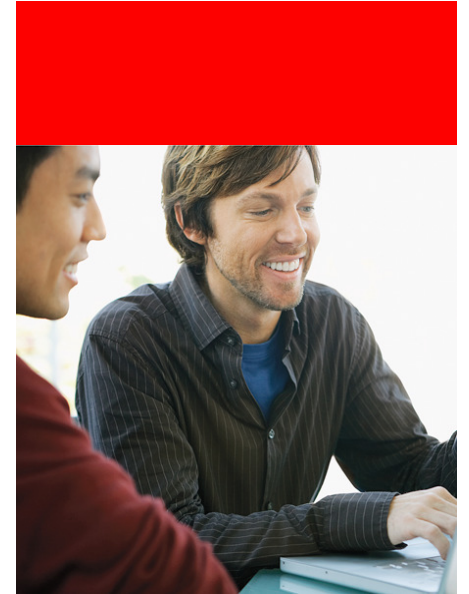
- Multiple vendors at each layer of the stack
- Fiercely competitive




# Language innovation

- Careful innovation on Java
- Aggressive innovation on new languages
- Platform optimizations for non-Java
  - InvokeDynamic bytecode in JDK 7
- Innovation feeding back into Java

# JDK 7 & 8 Features and Roadmap





Disclaimer: The JDK roadmap contents reflect Oracle's current assumption on what Java SE 7 and 8 will contain. It will be updated to reflect content changes decided by the JCP Expert Groups.



# JDK 7 Highlights

- Coin – Small language enhancements (JSR 334)
- Dynamic Language Support (JSR 292)
- Concurrency and Collections updates (JSR 166y)
- Network and File System (JSR 203)
- Security
- Internationalization
- Other Miscellaneous Enhancements
- JVM Convergence



# Coin - Small Language Enhancements (JSR 334)

- Small Language changes for Java
- Simplifies every-day coding for most developers
- Six new language features:
  - Strings in Switch
  - Better Integral Literals
  - Improved Exception handling
  - Improved Type Inference for Generic Instance Creation (Diamond Operator)
  - Automatic Resource Management (a.k.a try-with-resources)
  - Simplified Varargs Method Invocation
- Will as a proof of concept be applied to a subset of the core libraries



# Dynamic Language Support (JSR 292)

- Introduces invokedynamic byte code
- Enhanced interoperability with dynamically typed languages
- Improve performance for dynamic languages on the the jvm like JRuby, Jython and Rhino
  - Members of the development mailing list report 10x faster than invocation through reflection APIs. \*
- Limited value to pure Java Programs
  - But it is a step towards Lambda in JDK 8

(\*) Source: <http://www.cs.iastate.edu/~design/vmil/2010/papers/p06-kaewkasi.pdf>



# Concurrency and Collections Updates (JSR 166y)

## Lightweight Fork/Join Framework

- Framework for automating parallel divide-and-conquer
  - Programmer specifies problem decomposition, framework does the rest
- Automatic load balancing across dozens or hundreds of cores
- Suitable for many searching, sorting, and computational problems



# Concurrency and Collections Updates (JSR 166y)

## Other Enhancements

- Flexible Reusable Synchronization Barrier & Enhanced Blocking Buffer
  - Prerequisite to the fork/join framework
  - Useful standalone for advanced programmers
- Thread-Local PRNG (Pseudo Random Number Generator)
  - Avoids serialization bottleneck
- Thread Pool improvements
  - Better scalability and performance



# Network and File System

NIO.2 (JSR 203)

- Addition to filesystem interface
  - Bulk access to file attributes
  - Change notification
  - Escape to filesystem-specific APIs
  - Service-provider interface for pluggable filesystem implementations
- Socket channel API
  - Addresses multicasting, socket binding associated with channels, and related issues
- Asynchronous I/O API
  - Enables mapping to I/O facilities, completion ports, and various I/O event port mechanisms



# Network and File System

## Other Enhancements

- Redirection for Subprocess
  - stdin/stdout/stderr/etc
- SDP (Sockets Direct Protocol)
- SCTP (Stream Control Transmission Protocol) support
- NIO.2 Filesystem Provider for ZIP and JAR Archives
  - Useful in itself, as well as good NIO.2 proof-of-concept
- More encodings for Zip filenames and comments
  - [641 votes on bugparade](#)
- Improved Windows IPv6 support
  - Windows Vista and later



# Security

- Native ECC (Elliptic Curve Cryptography)
  - Alternative to RSA (public-key) system
  - Equivalent security with smaller key sizes, faster computations, lower power consumption, memory and bandwidth savings
- TLS (Transport Layer Security) 1.2
- Stronger pseudorandom functions, additional (stronger) hash/signature algorithms, enhanced key exchange
- ASLR (Address space layout randomization)
  - More difficult for an attacker to predict target addresses
- DEP (Data Execution Prevention) – Windows Only
  - Prevent an application or service from executing code from a non-executable memory region



# Internationalization

- Unicode 6.0
- IETF BCP47 and UTR35
  - Script codes needed for distinctions among languages that use different writing systems
  - Three-letter base language codes; three-digit region codes
- Separate user locale and user interface locale
- Currency data enhancements
  - Switch from hard coded data to a model similar to the one used for time zone data.
  - Easier to keep currency data up to date



# Other Enhancements

## Client & Graphics

- Added Nimbus Look & Feel to the standard
  - Much better –modern- look than what was previously available
- Platform APIs for Java 6u10 Graphics Features
  - Shaped and translucent windows
- JXLayer core included in the standard
  - Formerly SwingLabs component
  - Allows easier layering inside of a component
- Optimized Java2D Rendering Pipeline for X-Windows
  - Allows hardware accelerated remote X



# Other Enhancements

- Better font configuration on Unix
  - Now uses standard Unix mechanism to find fonts.
- JAXP 1.4.4, JAXWS 2.2 and JAXB 2.2
- JDBC 4.1, Rowset 1.1
- Upgrade class-loader architecture
  - Modifications to the ClassLoader API and implementation to avoid deadlocks in non-hierarchical class-loader topologies
- Close URLClassloaders
  - Allows applications to proactively clean up classloaders, freeing up native resources and unlocking JAR files
- Javadoc support for stylesheets



# JDK 7 Platform Support

- Windows x86
  - Server 2008, Server 2008 R2, 7 & 8 (when it GAs)
  - Windows Vista, XP
- Linux x86
  - Oracle Linux 5.5+, 6.x
  - Red Hat Enterprise Linux 5.5+, 6.x
  - SuSE Linux Enterprise Server 10.x, 11.x
  - Ubuntu Linux 10.04 LTS, 11.04
- Solaris x86/SPARC
  - Solaris 10.9+, 11.x
- Apple Mac OSX x64
  - will be supported post-GA, detailed plan TBD

*Note: These configurations are the ones primarily tested by Oracle, and for which we provide commercial support.*



# JDK 8 – Fall/Winter 2012

## Features from “Plan B”

- **Modularization**
  - Language and VM Support
  - Platform Modularization
- **Project Lambda**
  - Lambda Expressions
  - Default Methods
  - Bulk Data Operations
- **Annotations on Java types (JSR 308)**
- **More Small Language Enhancements**
  - Project Coin part 2

## Other Things On Oracle’s Wish List\*

- **Serialization fixes**
- **Multicast improvements**
- **Java APIs for accessing location, compass and other “environmental” data (partially exists in ME)**
- **Improved language interop**
- **Faster startup/warmup**
- **Dependency injection (JSR 330)**
- **Include select enhancements from Google Guava**
- **Small Swing enhancements**
- **More security/crypto features, improved support for x.509-style certificates etc**
- **Internationalization: non-Gregorian calendars, more configurable sorting**
- **Date and Time (JSR 310)**
- **Process control API**

\* Many of these will undoubtedly NOT make JDK 8.

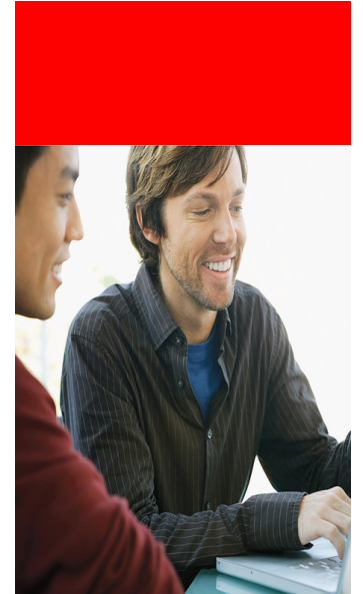


# Your Feedback

JDK 7 & 8 Features and Roadmap

- What will you use?
- What would you like to see added (or removed)?

# JVM Convergence & Roadmap





# JVM Convergence

HotSpot + JRockit

- One JVM with existing HotSpot and JRockit features
- HotSpot and JRockit team merged and hiring
- Multi-year effort
- Most investment will go into OpenJDK
- JRockit premium features will remain premium

# JVM Convergence

Project “HotRockit”

⦿ = Premium Feature

## JDK 7 GA

- Java SE 7 Support
- Rebranding
- Partial PermGen removal (stretch goal)
- Improved JMX Agent
- Command line serviceability tool (jcmd)
- Limited JRockit Mission Control Console support ⦿

## JDK 7 Update X

- Performance
- Initial JRockit Flight Recorder Support ⦿

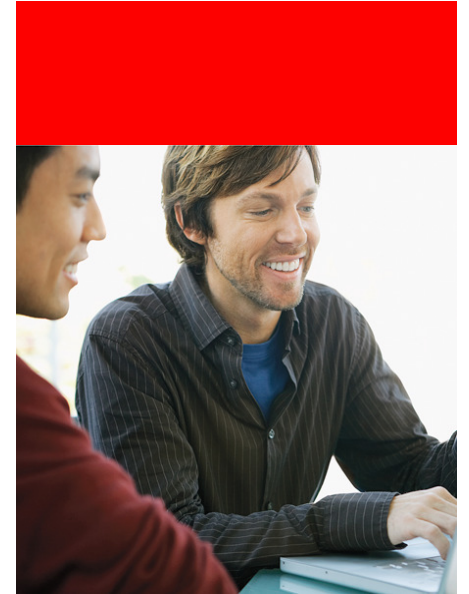
## JDK 7 Update Y

- More performance
- Improved command line serviceability (jcmd)
- G1 complete as CMS replacement
- Non-contiguous heaps
- Complete JRockit Flight Recorder Support ⦿
- JRockit Virtual Edition Support ⦿
- JRockit Real-Time ⦿

## JDK 8 GA

- Java SE 8 Support
- All performance features from JRockit ported
  - I/O Performance
  - JIT Optimizations
- All serviceability features from JRockit ported
  - Compiler controls
  - Verbose logging
- JRockit Mission Control Memleak Tool Support ⦿

# Garbage Collection and Latency





# Basic Memory Management Functions

- Provide efficient allocation
- Find unreachable objects, reclaim their memory and make it available for allocation
- Manage fragmentation



# The Garbage Collection Trade-Offs

- Throughput
- Latency
- Memory Usage



# Soft Real-Time vs Hard Real-Time

- Hard Real-Time
  - Absolute guarantees on latency
  - High overhead (end-to-end latency, throughput, footprint)
  - E.g. pace maker
- Soft Real-Time
  - “Many nines” guarantees on latency
  - Reasonable overhead (end-to-end latency, throughput, footprint)
  - E.g. algorithmic trading, Telco application, RFID processing



# Garbage Collection Induced Latency

- Stop the world pause
  - Perceived as a “pause”
- Java thread doing work to maintain consistency and state
  - Affect end-to-end latency but is not perceived as “pause”
- Non Java threads doing GC related work concurrently as Java threads are running
  - Minimal impact on Java threads but will affect HW caches
  - If CPUs are saturated Java threads will be preempted
- Latency = time to process transaction + GC pause



# JRockit Real-Time

- Drop-in solution for soft real-time Java SE
- Deterministic Garbage Collector (DetGC)
- Designed from the ground up for soft real-time
- Low and predictable GC induced latency
- Decreases frequency and severity of latency spikes
- High throughput
- Massive JVM engineering effort

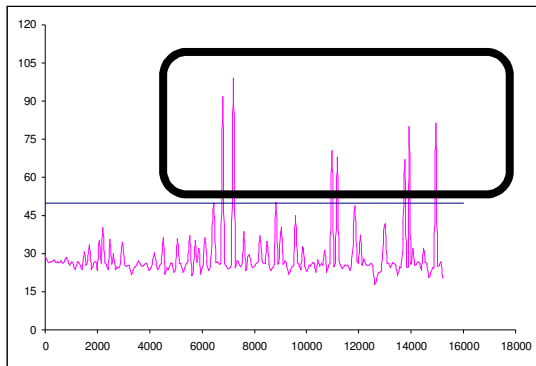


# JRockit Real-Time

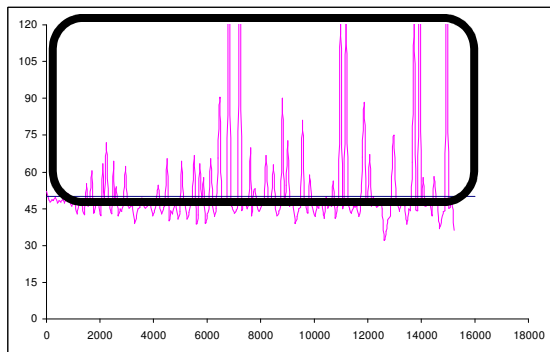
- Easy to deploy – no code changes
  - `java -Xms1024m -Xmx1024m -XgcPrio:deterministic – XpauseTarget=20`
- Configure pause target

# Benefits of Deterministic GC

## Traditional Java

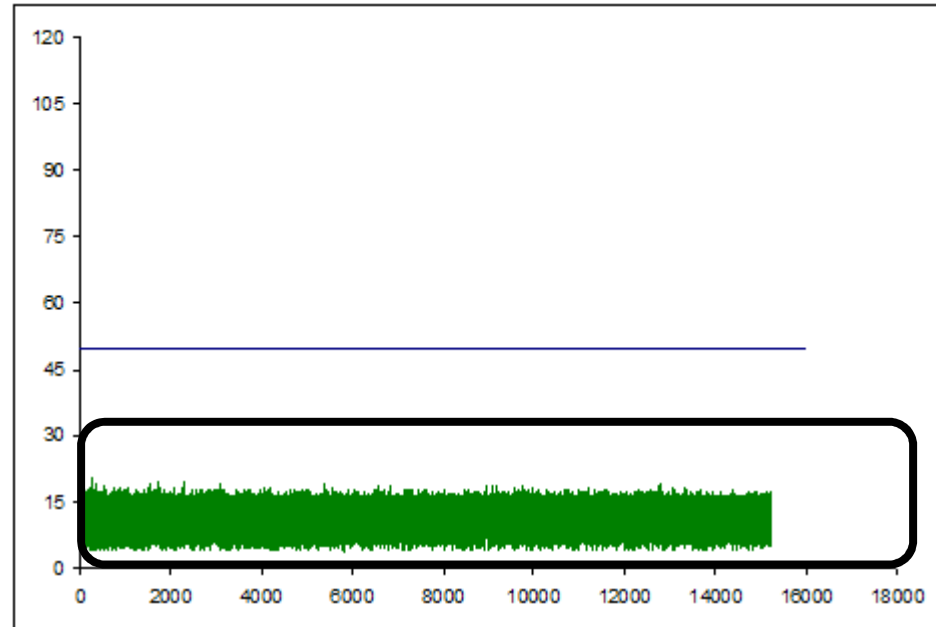


**During Low Load:** GC spikes and occasional timeouts visible



**During High Load:** GC pauses can result in unacceptable response times

## JRockit Real Time



JRRT Makes garbage collection *deterministic*.  
Allowing for the guarantee of SLAs.



# JRokit Real Time

## Typical Opportunities

### 1. Slow response leads to lost revenue

- Trading Application: Respond too slow and you miss the deal
- Trader quote -- “Every millisecond delay means we lose money due to losing deals or increase arbitrage costs.”

### 2. Unpredictable response time leads to less control

- Pricing Engine: Slow response means inability to respond with a price for a securities instrument

### 3. SLA Violation leads to penalties

- Communication Service Provider – Customer SLA’s require immediate, predictable response
- LOB Owner -- “We have a stringent response time SLA to our customers. If we don’t meet it, we have to pay fines.”

### 4. New SLAs lead to Higher Revenues & Market Leadership

- Securities Trading Customers -- Ability to offer new SLAs could lead to new revenue streams
- Market-leading product, consistent offer, results in loyalty, revenues, & market leadership

### 5. Slow response invalidates use case

- RFID: Must be fast or savings by automating process are lost

### 6. Moving to Java saves money

- Port legacy C/C++ apps without worrying about pauses
- Data Center Manager quotes
  - “We have problems with long GC pauses in our Java application”
  - “We are going to build an application which requires millisecond response times, and want to do it in Java”

# ORC 'FIX' latency -- 77 to 7 milli-seconds!

## ORC Software Background

- Provides solutions for the global financial industry in advanced derivatives trading and low latency connectivity
- FIX protocol – De facto standard for electronic trading
- Orc CameronFIX is a market-leading FIX engine used by exchanges, brokers, the buy-side for high-performance FIX connectivity

## Key Differentiator

- Consistent latency performance at 99.995% under extreme loads
- Between electronic trading partners such as exchanges and brokers

## Oracle Solution – JRockit Real Time

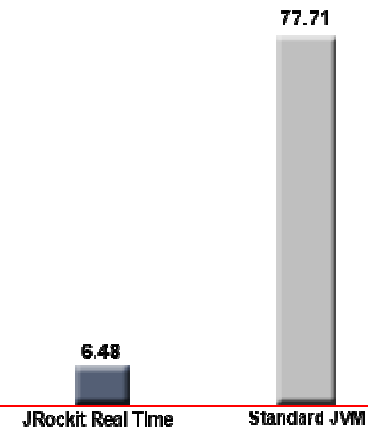
- Run Orc's CameronFIX engine on JRockit Real Time providing consistent low latency

## Results

- Reduction of latency spikes from 77 milliseconds using a competitor JVM to under 7 milliseconds!
- JRockit Real Time on the Sun X4450 provided a 91% reduction in latency at the 99.995 percentile.



Comparison of 99.995 Percentile Latencies





# JRokit Real Time -- Use Case #3 of 3

## A Major Investment Bank

### Customer

- Goal -- Eliminate artificial arbitrage created by slow trade order execution
- Challenge -- Re-writing applications to lower latency was time and cost prohibitive

### Oracle Solution

- Replaced existing Java platform with JRokit Real Time
  - Deterministic technology allowed for bounded garbage collection response times
- Drop-in solution - required no application changes
- Latency response times improved 30%.
- Provided Immediate increase in profits due to securing best price at right time
- Eliminated the “gaming of the system” / artificial arbitrage



# Be Nice To The Deterministic GC (1/3)

- 1. Keep it small and simple
  - Unnecessary complexity makes profiling and tuning difficult
- 2. Choose your heap size with care
  - If you have 900 MB of “live” data and 1 GB of heap you will have very frequent GCs
  - Pause times are dependent on live data at the time of GC, so premature GC leads to longer pause times
  - Profile to find size of data, then set heap to 2-3x that number
- 3. Watch the allocation rate
  - If you allocate 1 GB/second you will cause very frequent GCs
  - High allocation rate can lead to premature GC, see above
  - Avoid unnecessary object creation and copying
  - Reuse large objects (arrays) when possible



# Be Nice To The Deterministic GC (2/3)

- 4. Watch out for “dark matter”
  - Some allocation patterns lead to heap fragmentation
  - Heap fragmentation leads to long pause times
  - Don't
    - Store every 10:th new object in a map
    - Allocate huge numbers of large arrays
  - Choose your heap size with care (see previous page)
  - Reuse large objects (arrays) when possible
- 5. Use reference objects sparingly
  - Finalizers, soft, weak and phantom references require special processing during GC
  - Decrease reference object usage if profiling shows an impact on pause times



## Be Nice To The Deterministic GC (3/3)

- 6. Leave breathing room for the concurrent GC phases
  - If you starve the GC threads, your heap will fill up before the concurrent phase is done
  - Emergency GC (long pause) or out of memory
  - **Keep CPU utilization below ~80%**
- 7. Choose your data structures with care
  - Linked lists lead to pointer chasing during GC
  - Traversing linked data structures cannot be parallelized
  - **Consider using ArrayList instead of LinkedList**
- 8. Don't overdo it
  - The Deterministic GC can take a lot of punishment
  - "...premature optimization is the root of all evil" – Donald Knuth



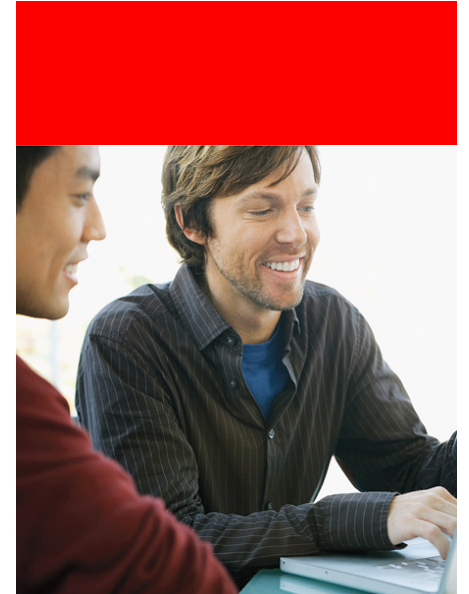
# HotSpot

## Concurrent Mark Sweep

- Low latency GC in HotSpot – Concurrent Mark Sweep
- Generational
  - Objects normally die young
  - 3 spaces in young generation where objects age
- Used by many customers
- Requires careful tuning and GC aware application for soft real time use cases
- Customers achieving consistent millisecond latencies
- May require daily re-start to deal with fragmentation



# Tooling for Low Latency





# Flight Recorder

## Continuous Profiling with Triggers in Production

- Circular Buffer of Profiling Data
- Near zero overhead profiling for tuning & production
  - Records latency of java and JVM events
  - Piggy back on information the JVM
  - Lock free, thread local, HW cache friendly, compact
  - Hardware accelerated timing
  - Highly optimized stack trace collection and storage
  - Efficient instrumentation of JVM code generator and the JVM
- Triggers to save profiling data up to SLA violation
  - e.g. response time too long -> dump 10 minutes of data
- Powerful Tooling - JRockit Mission Control



# Flight Recorder

Continuous Profiling with Triggers in Production

- DEMO



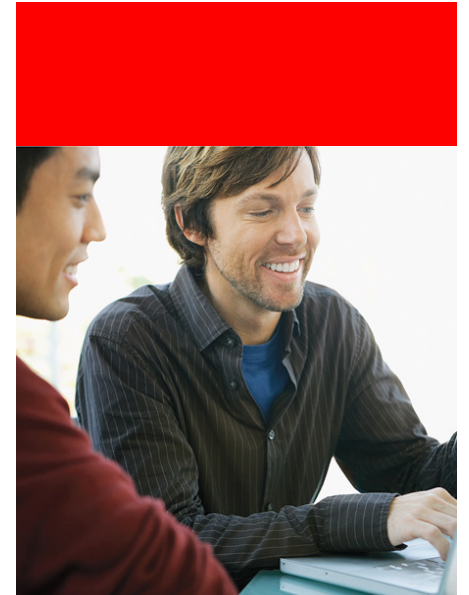
# Your Feedback

## JVM and Tooling

- Scalability preference – Many JVMs or one BIG
- Heap sizes?
- Thread local GC
  - Transaction API hints to GC
  - Mostly Transaction local allocation & reachability
- Public/standard JFR API and middleware integration?
- DTrace?
- Enterprise Manager integration?
- Replace `java.util.logging` somewhere?
- Flight Recorder infrastructure for other logging



# Exlogic Hardware for Java



# Exalogic Elastic Cloud Compute Nodes

## High Performance Compute Nodes

2.93 GHz Xeon Cores	360
1333 MHz RAM	2.8 TB
FlashFire SSD	960 GB

- **CPU, RAM and IO balanced**
  - Optimal Java performance
- **Fully redundant power, disk**
- **Hot swappable**
- **Industry standard**
- **Oracle Enterprise Linux and Oracle Solaris factory installed**



# Exalogic Elastic Cloud Network

NM2 GW and NM2 36P InfiniBand Switch/Ethernet Gateways

## Exalogic IO Fabric

QDR InfiniBand	40 Gb/s
Latency (MPI Ping)	1.2 $\mu$ s

- **Lossless switched fabric**
- **Channel-based architecture**
  - Quality of Service and security
  - Fault tolerance and failover
  - Extreme Scalability

## Datacenter Network Integration

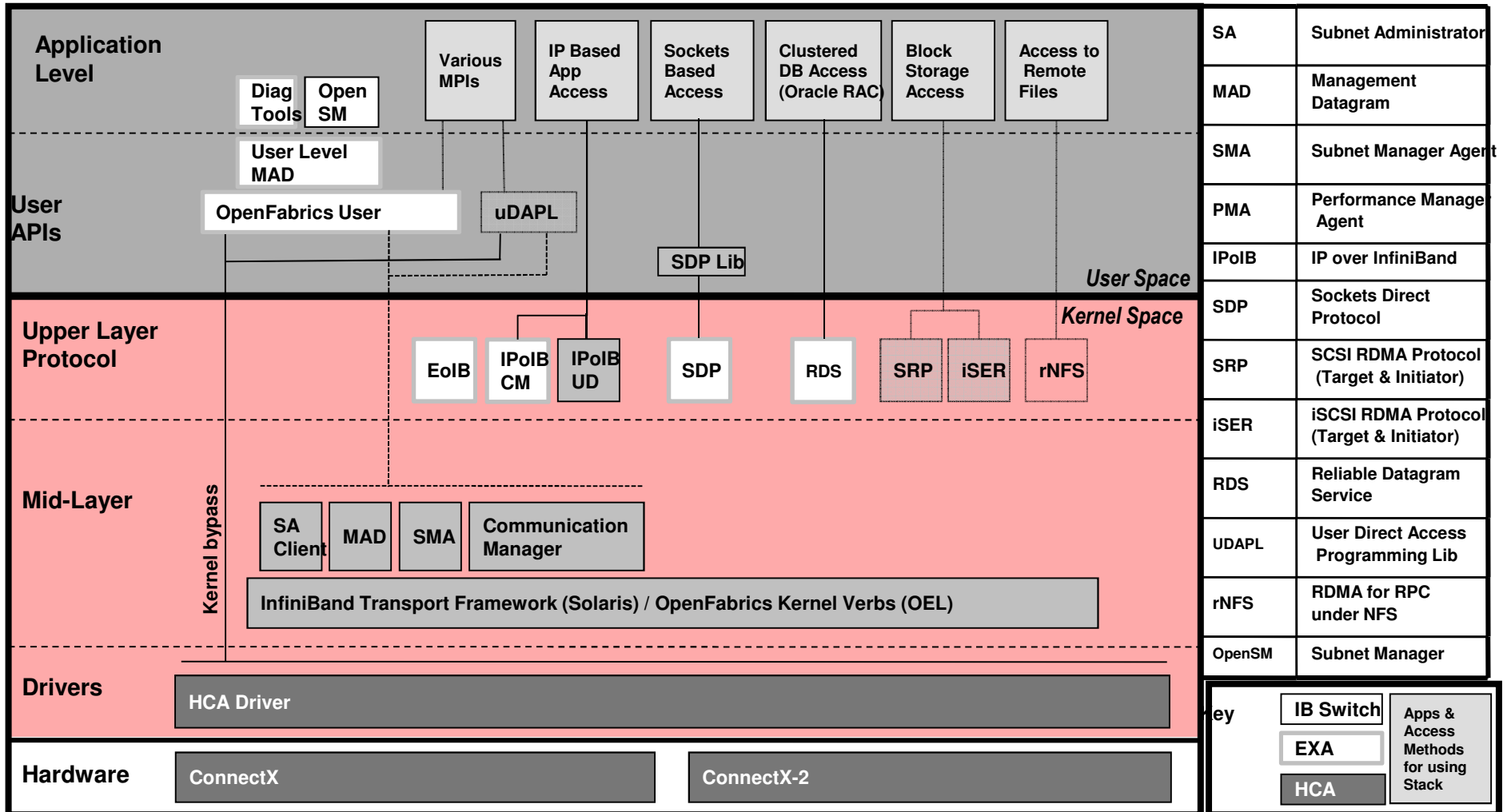
Service Network (Fiber)	10 GbE
Management Network	GbE



ORACLE

# The InfiniBand Stack

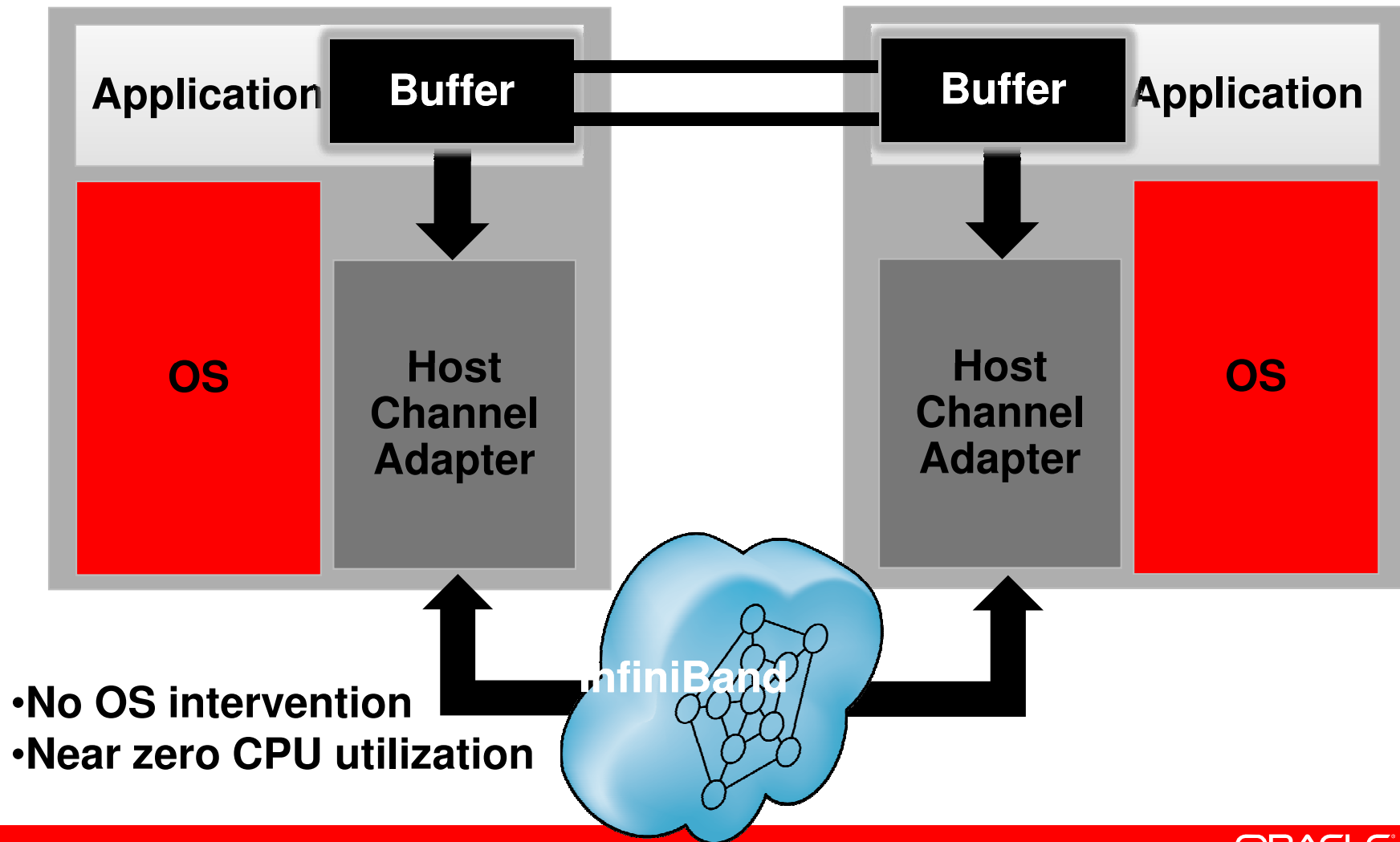
ORACLE  
SOLARIS



ORACLE

# Remote Direct Memory Access (RDMA)

Direct access by remote application

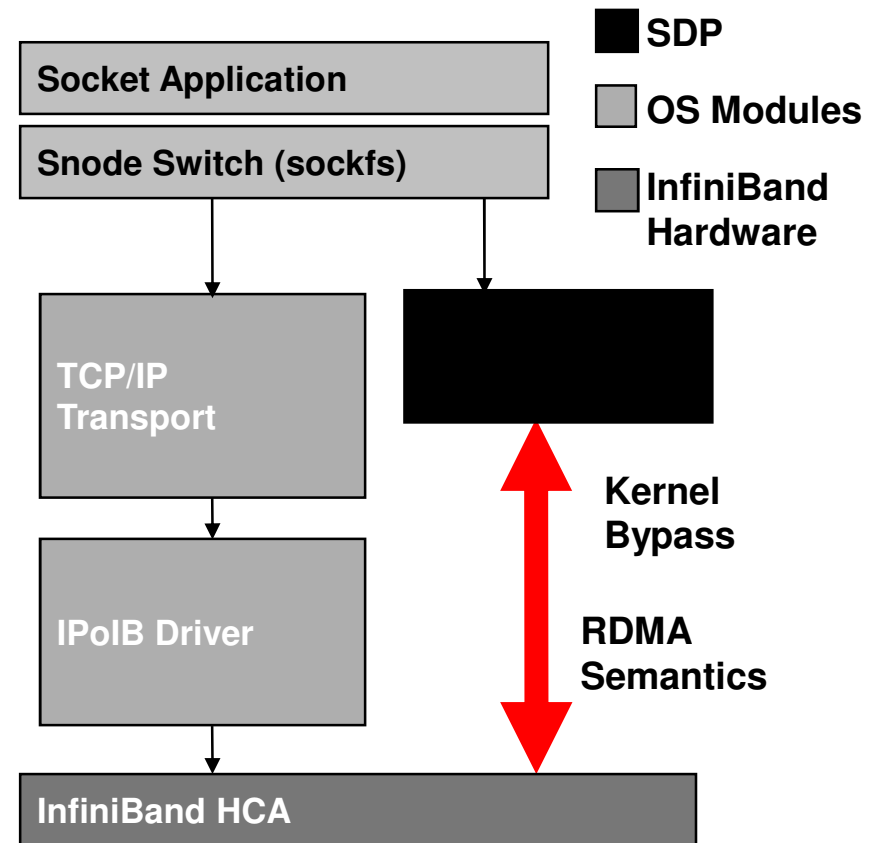


ORACLE

# The InfiniBand Stack

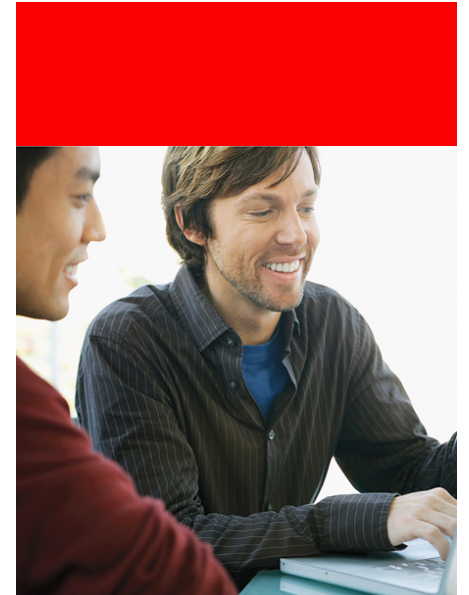
## Sockets Directs Protocol (SDP)

- Access to IB via Socket API
- IBTA standard wire protocol
- SOCK\_STREAM transport
- Leverages InfiniBand Capabilities
  - Transport Offload – Reliable Connection
  - Zero Copy – Using RDMA
  - Kernel Bypass
  - Delivers low latency
- Functional in IPoIB subnets
  - IP address resolution dependency on IPoIB





# The Stack Engineered Together






# Coherence on JRRT on Exalogic

- Setup
  - 2 Exalogic nodes = 1/16<sup>th</sup> full Exalogic machine ~40GB cache
  - Exalogic node = 3 Coherence storage nodes with 20GB heap
  - 1/3 heap storage, 1/3 backup and 1/3 free
  - Drivers (8 JVMs) execute 50 put/get/remove per transaction
  - InfiniBand communication
  - Generational Concurrent vs Deterministic GC
- Results
  - Max GC pause 2,000 ms -> 15 ms (DetGC + Coherence opt)
  - Average end-to-end transaction: 3.4 ms (JVM InfiniBand opt)
  - Throughput reduced by only 4%



# **Hardware and Software Engineered to Work Together**

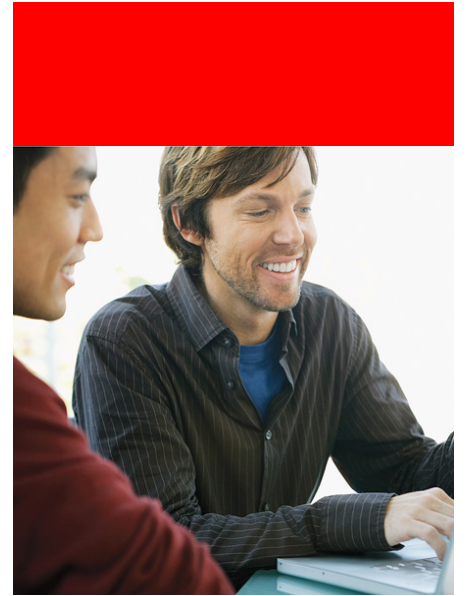


The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.





# Supporting Slides





# Higher Revenues, Customer Sat

## FXall Background and Challenges

- Leading provider of automated/workflow-based trading solutions for foreign exchange and treasury products
- Market makers are stretching limits with higher #s, quote frequency
- More Clients (takers), shorter reaction time, seeking more liquidity options
- System Requirements -- Java based -- Need to maintain server latency within 20ms
  - One message with high latency may lose the deal!



## Solution – Oracle JRockit Real Time

- Two weeks of POC
- Multiple trades exceeding 100ms barrier -- Sun JVM too unpredictable
- Excellent tooling -- Sun JVM provided minimal/ no diagnostic tools for latency bottlenecks
- Target 11M trades/ day, but latency reqts stay same -- Sun JVM wont scale without extensive recoding

## Results

- Significantly Faster -- **15x** faster than Sun, at **60% higher load!**
  - Numbers overall more consistent (*deterministic*)
- Significantly better use of resources -- Heap sizes reduced from 3GB to 0.5GB (512MB)

## Business Benefits

- Higher \$\$ -- Lower-latency => clients get better live prices, resulting in higher likelihood of trades
- More customers, Faster -- With scalability addressed, able to onboard more market-makers, large clients
- Improved Developer Effectiveness -- Developers spend more time on solving business issues