Running Java on Linux on System z

Session # 12378

Marcel Mitran
Chief Architect, IBM JVM on System z
mmitran@ca.ibm.com

February 6th, 2013
IBM’s statements regarding its plans, directions, and intent are subject to change or withdrawal without notice at IBM’s sole discretion. Information regarding potential future products is intended to outline our general product direction and it should not be relied on in making a purchasing decision. The information mentioned regarding potential future products is not a commitment, promise, or legal obligation to deliver any material, code or functionality. Information about potential future products may not be incorporated into any contract. The development, release, and timing of any future features or functionality described for our products remains at our sole discretion.
Content

- IBM Java on Z
  - History, overview and roadmap

- IBM J9 Virtual Machine and IBM Testarossa JIT
  - IBM Monitoring and Diagnostic Tools for Java

- Java 7 SR3
  - zEC12 exploitation and performance
IBM and Java

- **Java is critically important to IBM**
  - Fundamental infrastructure for IBM’s software portfolio
  - WebSphere, Lotus, Tivoli, Rational, Information Management (IM)

- **IBM is investing strategically for Java in Virtual Machines**
  - As of Java 5.0, single JVM support
    - JME, JSE, JEE
  - New technology base (J9/TR Compiler) on which to deliver improved performance, reliability, serviceability

- **IBM also invests in, and supports public innovation in Java**
  - OpenJDK, Eclipse, Apache (XML, Aries, Derby, Geronimo, Harmony, Tuscany, Hadoop …)
  - Broad participation in relevant open standards (JCP, OSGi)
IBM’s Approach to Java Technology

- **IBM Java Technology** (openJDK, others)

- *Listen to and act upon market requirements*

- *World class service and support*

- *Available on more platforms than any other Java implementation*

- *Highly optimized*

- *Embedded in IBM’s middleware portfolio and available to ISV partners*

**Quality Engineering**
- Performance
- Security
- Reliability
- Serviceability

**Production Requirements**
- IBM Software Group
- IBM eServer
- ISVs
- IBM Clients
JVM Architectural Overview

User Code

Java API
e.g. Java6/Java7

Java Runtime Environment
e.g. J9 R26

Operating Systems

AIX | Linux | Windows | z/OS
---|---|---|---
PPC-32 x86-32 PPC-32 zArch-31 x86-32 zArch-31
PPC-64 x86-64 PPC-64 zArch-64 x86-64 zArch-64

= User Code
= Java Platform API
= VM-aware
= Core VM
Key Differences between Oracle and IBM Java

- IBM and Oracle use the same reference implementation of Java Class Libraries (e.g. OpenJDK)
  - Key differences to be aware of:
    1. Security: Standards do not impose strong separation of interest
    2. ORB: OMG CORBA standard rules
    3. XML: Xerces/Xalan used by both vendors as of Java5, although different levels may be used.

- IBM uses the J9/TR runtime, Oracle uses Hotspot
  - Different JIT/GC/VM tuning and controls
  - Tooling is distinct (e.g. IBM’s Health Center)
IBM continues to invest aggressively in Java for System z, demonstrating a rich history of innovation and performance improvements.

http://www.centerline.net/review/#/3332_B

Timelines and deliveries are subject to change.
Java Road Map

Language Updates

Java 5.0
- New Language features:
  - Autoboxing
  - Enumerated types
  - Generics
  - Metadata

Java 6.0
- Performance Improvements
- Client WebServices Support

Java 7.0
- Support for dynamic languages
- Improve ease of use for SWING
- New IO APIs (NIO2)
- Java persistence API
- JMX 2.x and WS connection for JMX agents
- Language Changes

Java 8.0**
- Language improvements
- Closures for simplified fork/join

IBM Java Runtimes

IBM Java 5.0 (J9 R23)
- Improved performance
  - Generational Garbage Collector
  - Shared classes support
  - New J9 Virtual Machine
  - New Testarossa JIT technology
  - First Failure Data Capture
  - Full Speed Debug
  - Hot Code Replace
  - Common runtime technology
  - ME, SE, EE

IBM Java 6.0 (J9 R24)
- Improvements in
  - Performance
  - Serviceability tooling
  - Class Sharing
  - XML parser improvements
  - z10™ Exploitation
  - DFP exploitation for BigDecimal
  - Large Pages
  - New ISA features

IBM Java 6.0.1/Java7.0 (J9 R26)
- Improvements in
  - Performance
  - GC Technology
  - zEC12™ Exploitation
  - Transactional Execution
  - Runtime Instrumentation
  - Flash 1Meg pageable LPs
  - 2G large pages
  - Hints/traps
  - Data Access Accelerator
  - Cloud: Multi-tenancy/Virtualization

IBM Java7.0SR3/Java.Vnext**
- Improvements in
  - Performance
  - GC Technology
  - zEC12™ Exploitation
  - Transactional Execution
  - Runtime Instrumentation
  - Flash 1Meg pageable LPs
  - 2G large pages
  - Hints/traps
  - Data Access Accelerator
  - Cloud: Multi-tenancy/Virtualization

**Timelines and deliveries are subject to change.
Java 7.0 – What to look for

New I/O
- Meets the increasingly I/O intensive demands of data mining and analytics workloads
- Significant performance and footprint gains from async I/O

Concurrency Libraries
- Exploit larger multi-core systems, such as next generation Power and System z, by providing better scalability, higher throughput and lower total cost of ownership from server consolidations

Dynamic language support
- Leverage the advantages of a single runtime for dynamic language applications written in PHP, Groovy, jRuby and jython

Language improvements
- Improved efficiency through simplified day-to-day programming tasks
- Protect developer commitment to, and customer/ISV investment in, the Java ecosystem.
Java8-Beta Program

• Provides Java SE 8 compatibility, while exploiting the unique capabilities of IBM platforms to achieve performance and usability improvements
  • To provide early technology access during the development cycle
  • To assist Java 8 in satisfying customer requirements
  • To provide feedback to IBM

• New in IBM SDK, Java Technology Edition, Version 8:
  • Compatibility with the new Java SE 8
  • Leveraging new IBM hardware (e.g. IBM zEnterprise EC12)
  • Improved performance for workload optimized runtimes, which delivers better application throughput without changes to application code
  • Enhanced support for Cloud & Multi-tenancy environments
  • Improved efficiency of manipulating native data records/types directly from Java code

• Managed and Open Beta
Java8: Language Innovation -- Lambdas

New syntax to allow concise code snippets and expression

• Useful for sending code to java.lang.concurrent
• On the path to enabling more parallelisms

Collections.sort(people, new Comparator<Person>() {
   public int compare(Person x, Person y) {
       return x.getLastName().compareTo(y.getLastName());
   }
});

people.sort(comparing(Person::getLastName));

Java8: Data Access Accelerator

A Java library for bare-bones data conversion and arithmetic

Operates directly on byte arrays
No Java object tree created
Orchestrated with JIT for deep platform opt.
Avoids expensive Java object instantiation
Library is platform and JVM-neutral

Marshalling and Un-marshalling
Transform primitive type (short, int, long, float, double) ↔ byte array
Support both big/little endian byte arrays

Packed Decimal (PD) Operations
Arithmetic: +, -, *, /, % on 2 PD operands
Relation: >, <, >=, <=, ==, != on 2 PD operands
Error checking: checks if PD operand is well-formed
Other: shifting, and moving ops on PD operand

Decimal Data Type Conversions
Decimal ↔ Primitive: Convert Packed Decimal(PD), External Decimal(ED), Unicode primitive
types (int, long)
Decimal ↔ Decimal: Convert between dec. types (PD, ED, UD)
Decimal ↔ Java: Convert dec. types (PD, ED, UD) ↔
BigDecimal, BigInteger

Current Approach:
```java
byte[] addPacked(array a[], array b[]) {
    BigDecimal a_bd = convertPackedToBd(a[]);
    BigDecimal b_bd = convertPackedToBd(b[]);
    a_bd.add(b_bd);
    return (convertBDtoPacked(a_bd));
}
```

Proposed Solution:
```java
byte[] addPacked(array a[], array b[]) {
    DAA.addPacked(a[], b[]);
    return (a[]);
}
```
Looking Ahead: PackedObjects with IBM Java

PackedObjects

Experimental feature in the IBM JVM. Introduces a new Java type that implements an explicit object model which tightly packs fields allowing for natural and efficient direct mapping of structured data.

Goals

• Allow for explicit source-level representation of structured data in Java
• Improve serialization and I/O performance
• Allow direct access to “native” (off-heap) data

http://www.slideshare.net/mmitran/ibm-java-packed-objects-mmit-20121120
http://duimovich.blogspot.ca/2012/11/packed-objects-in-java.html
Speak to me in ‘Java’

- Java only speaks ‘Java’…
  - Data typically must be copied/re-formatted onto/off Java heap
  - Costly in path-length and footprint
On-Heap PackedObject

- Allows controlled layout of storage of data structures on the Java heap
- Reduces footprint of data on Java heap
- No (de)serialization required
Off-Heap PackedObject

- Enable Java to talk directly to the native data structure
  - Avoid overhead of data copy onto/off Java heap
  - No (de)serialization required
Looking Ahead: Cloud with IBM Java

- **Multi-tenancy support** will allow multiple applications to run in a single shared JVM for high-density deployments.
  - *Win*: Footprint reduction enabled by sharing runtime and JVM artifacts while enforcing resource consumption quotas
  - *Platform Coverage*: 64-bit, balanced GC policy only
  - *Ergonomics*: Single new command-line flag (-Xmt = multi tenancy)

- **Runtime Adjustable Heap Size** (-Xsoftmx)
  - JMX beans allow for dynamically adjusting heap size
  - Allows users to take advantage of hot-add of memory

- **Hypervisor, Virtual Guest, and Extended-OS JMX Beans**
  - Allows applications to detect and identify the installed hypervisor and query attributes of LPAR
  - Provides richer access to operating system performance statistics
Multitenancy: Isolation and Density

Share-nothing (maximum isolation)
1+ GB / tenant
Application
Middleware
OS Images
Hardware

Shared hardware
1+ GB / tenant
Application
Middleware
OS Images
Hardware

Shared OS
100’s MB / tenant
Application
Middleware
OS Images
Hardware

Shared Process
10’s MB / tenant
Application
Middleware (e.g. WAS)
OS Images
Hardware

Share-everything (maximum sharing)
10’s KB / tenant
Application
Tenant

Isolation
‘Mission critical’ apps

Density

‘free’ apps

Timelines and deliveries are subject to change.
IBM Java Runtime Environment

- IBM’s implementation of Java 5, Java 6 and Java 7 are built with IBM J9 Virtual Machine and IBM Testarossa JIT Compiler technology
  - Independent clean-room JVM runtime & JIT compiler

- Combines best-of breed from embedded, development and server environments… from a cell-phone to a mainframe!
  - Lightweight flexible/scalable technology
  - World class garbage collection – gencon, balanced GC policies
  - Startup & Footprint - Shared classes, Ahead-of-time (AOT) compilation
  - 64-bit performance - Compressed references & Large Pages
  - Deep System z exploitation – zEC12/z196/z10/z9/z990 exploitation

- Millions of instances of J9/TR compiler
IBM Testarossa JIT Compiler – Introduction

- IBM Testarossa JIT is IBM’s Production JIT on all Platforms since SDK5
- Developed at the IBM Toronto Lab
- The Toronto Lab has 30+ years of expertise in compilation and optimization technologies

- Close relationships with:
  - Research: productizing innovative ideas and experimental technologies. (Tokyo/Watson Research Lab)
  - Hardware: best possible performance with the underlying system and processor. (Poughkeepsie, Austin, xSeries)
  - IBM Middleware: work with DB2®, WAS to provide strong performance (SVL, Toronto, Raleigh)
IBM Testarossa JIT –
Dynamic, adaptive, optimizing compiler

• **Dynamic**
  - Triggered at runtime based on projected profitability of compilation
  - Compiled methods can be freely intermixed with interpreted callers/callees
  - May have multiple versions of methods built with different levels of optimization

• **Adaptive**
  - Sensitive to need for program to have CPU (e.g. throttled during startup, runs on asynchronous thread)
  - Able to profile program to retrieve common control paths or data values
  - Profile information used in subsequent re-optimizing compilation step

• **Optimizing**
  - Comprehensive collection of conventional optimizations
    - control flow simplification, data flow analysis, etc
  - Speculative and Java-specific optimizations
    - de-virtualization, partial inlining, lock coarsening, etc
  - Deep exploitation of System z micro-architecture
**IBM Testarossa JIT – Compilation Strategy**

**Goals**
- Focus compilation CPU time where it matters
  - Stager investment over time to amortize cost
- Methods start as interpreted
  - Interpreter does first level profiling
- After N invocations methods get compiled at ‘warm’ level
- Sampling thread used to identify hot methods
- Methods may get recompiled at ‘hot’ or ‘scorching’ levels
- Transition to ‘scorching’ goes through a temporary profiling step
  - Global optimizations are directed using profiling data
  - Hot paths through methods are identified
    - register allocation, branch straightening, etc
  - Values/types are profiled, hot paths are specialized/versioned
  - Virtual calls are profiled, hot targets are in-lined
IBM Testarossa JIT – System z Support

- Idioms are recognized in Java source/bytecodes
- Bytecodes converted to CISC instructions**
- CISC Instructions:
  - TROT, TRTO, TRTT, TROO (TR = Translate, O = One Byte, T = Two bytes)
  - SRST (search string)
  - MVC (move character)
  - XC (exclusive-or)
  - CLC (compare-logical)

- Example:

```java
while (i < end) {
    value = table[arrB[i+offsetB]];
    if (value == termChar) break;
    arrA[i+offsetA] = value;
    ++i;
}
```

IBM J9 Garbage Collector

- IBM J9 VM garbage collector family

<table>
<thead>
<tr>
<th>Policy</th>
<th>Recommended usage</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>optThroughput</td>
<td>optimized for throughput</td>
<td>default in Java5 and Java6</td>
</tr>
<tr>
<td>optAveragePause</td>
<td>optimized to reduce pause times</td>
<td></td>
</tr>
<tr>
<td>gencon</td>
<td>optimized for transactional workloads</td>
<td>default in Java601/Java7</td>
</tr>
<tr>
<td>subPools</td>
<td>optimized for large MP systems</td>
<td>deprecated in Java601/Java7</td>
</tr>
<tr>
<td>balanced</td>
<td>optimized for large heaps</td>
<td>added in Java601/Java7</td>
</tr>
</tbody>
</table>

- Why have many policies? Why not just “the best”?  
  - Cannot always dynamically determine what tradeoffs the user/application are willing to make

  - **Pause time vs. Throughput**
    - Trade off frequency and length of pauses vs. throughput

  - **Footprint vs. Frequency**
    - Trade off smaller footprint vs. frequency of GC pauses/events
IBM J9 Garbage Collector: \texttt{-Xgcpolicy:optthruput}

The default policy in Java5 and Java6.

Used for applications where raw throughput is more important than short GC pauses. The application is stopped each time that garbage is collected.

\begin{itemize}
  \item Thread 1
  \item Thread 2
  \item Thread 3
  \item Thread n
\end{itemize}

\textit{Picture is only illustrative and doesn’t reflect any particular real-life application. The purpose is to show theoretical differences in pause times between GC policies.}
IBM J9 Garbage Collector: `-Xgcpolicy:optavgpause`

Trades high throughput for shorter GC pauses by performing some of the garbage collection concurrently. The application is paused for shorter periods.

*Picture is only illustrative and doesn’t reflect any particular real-life application. The purpose is to show theoretical differences in pause times between GC policies.*
IBM J9 Garbage Collector: -Xgcpolicy:gencon

• Best of both worlds
  • Throughput + Small Pause Times
  • Shown most value with customers

• Two types of collection:
  • Generational nursery (local) collection
  • Partially concurrent nursery & tenured (global) collection

• Why a generational + concurrent solution?
  • For most workloads objects die young
    • Generational allows a better return on investment (less effort, better reward)
    • Performance can be close or even better than standard configuration
  • Reduce large pause times
    • Partially concurrent with application thread (application thread is ‘taxed’)
    • Mitigates cost of object movement, and cache misses
IBM J9 Garbage Collector: -Xgcpolicy:gencon

Default policy in Java6.0.1/Java7

Handles short-lived objects differently than objects that are long-lived. Applications that have many short-lived objects can see shorter pause times with this policy while still producing good throughput.

Picture is only illustrative and doesn't reflect any particular real-life application. The purpose is to show theoretical differences in pause times between GC policies.
IBM J9 Garbage Collector: A closer look into gencon

• Heap is split into two areas:
  – Objects created in the nursery (a small but frequently collected area)
  – Objects that survive a number of collections are promoted to tenured area (less frequently collected)

• Nursery is further split into two spaces: ‘allocate’ and ‘survivor’
• A collection in the nursery (scavenge) copies objects from the ‘allocate’ space to the ‘survivor’ space
  – Reduces fragmentation, improves data locality, speeds up future allocations
• If an object survive X number of scavenges it gets promoted to the ‘tenure’ space

The division between allocate and survivor space is dynamic.
It will be adjusted depending on the survival rate.
IBM J9 2.6 Technology Enhancements: Garbage Collection: Balanced Policy

**Improved responsiveness in application behavior**
- Reduced maximum pause times to achieve more consistent behavior
- Incremental result-based heap collection targets best ROI areas of the heap
- Native memory aware approach reduces non-object heap consumption

**Next generation technology expands platform exploitation possibilities**
- Virtualization – Group heap data by frequency of access, direct OS paging decisions
- Dynamic reorganization of data structures to improve memory hierarchy utilization (performance)

**Recommended deployment scenarios**
- Large (>4GB) heaps
- Frequent global garbage collections
- Excessive time spent in global compaction
- Relatively frequent allocation of large (>1MB) arrays

**Input welcome: Help set directions by telling us your needs**
IBM J9 Garbage Collector: Tuning

- GC Tuning documentation
  - [http://proceedings.share.org/client_files/SHARE_in_San_Jose/S1448Kl161816.pdf](http://proceedings.share.org/client_files/SHARE_in_San_Jose/S1448Kl161816.pdf)

- GC and Memory Visualizer – Views on verbose GC

- Typical configuration
  - Pick a policy based on desired application behaviour
  - Tune heap sizes (use tooling)
  - Helper threads (-Xgcthreads)
  - Avoid finalizers
  - Don’t use System.gc()
  - Lots of other tuning knobs, suggest try hard to ignore, to avoid over-tuning

- Memory leaks are possible even with a garbage collector
What is IBM Support Assistant?

- **IBM Support Assistant**
  - A free application that simplifies and automates software support
  - Helps customers analyze and resolve questions and problems with IBM software products.
  - Includes rich features and serviceability tools for quick resolution to problems

- **Meant for diagnostics and problem determination**
  - Not a monitoring tool

**Find Information**
Easily find the information you need including product specific information and search capabilities.

**Analyze Problem**
Diagnose and analyze problems through serviceability tools, collection of diagnostic artifacts, and guidance through problem determination.

**Manage Service Request**
Effectively submit, view and manage your service requests enhanced with automated collection of diagnostic data.
IBM Monitoring and Diagnostic Tools for Java - Health Center

What problem am I solving
• What is my JVM doing? Is everything ok?
• Why is my application running slowly?
• Why is it not scaling?
• Am I using the right options?

Overview
• Lightweight live monitoring tool with very low overhead
• Understand how your application is behaving, diagnose potential problems with recommendations.
• Visualize garbage collection, method profiling, class loading, lock analysis, file I/O and native memory usage
• Suitable for all Java applications running on IBM’s JVM
IBM Monitoring and Diagnostic Tools for Java - GCMV

What problem am I solving
• How is the Garbage Collector (GC) behaving? Can I do better?
• How much time is GC taking?
• How much free memory does my JVM have?

Overview
• Analyse Java verbose GC logs, providing insight into application behaviour
• Visualize a wide range of garbage collection data and Java heap statistics over time
• Provides the ability to detect memory leaks and optimized garbage collection
• Recommendations use heuristics to guide you towards GC performance tuning

Tuning recommendation
- The garbage collector seems to be compacting excessively. On average 45% of each pause was spent compacting the heap. Compaction occurred on 40% of collections. Possible causes of excessive compaction include the heap size being too small or the application allocating objects that are larger than any contiguous block of free space on the heap.
- The garbage collector is performing system (full) GCs. 5 out of 145 collections (3.44%) were triggered by System.gc() calls. The use of System.gc() is generally not recommended since they can cause long pauses and do not allow the garbage collection algorithm to optimise themselves. Consider inspecting your code for occurrences of System.gc().
- The mean occupancy in the nursery is 7%. This is low, so the gencon policy is probably an optimal policy for this workload.
- The mean occupancy in the tenured area is 14%. This is low, so you have some room to shrink the heap if required.

Summary
<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation failure count</td>
<td>140</td>
</tr>
<tr>
<td>Concurrent collection count</td>
<td>8</td>
</tr>
<tr>
<td>Forced collection count</td>
<td>5</td>
</tr>
<tr>
<td>GC Mode</td>
<td>gencon</td>
</tr>
<tr>
<td>Global collections - Mean garbage collection pause (ms)</td>
<td>1.85</td>
</tr>
<tr>
<td>Global collections - Mean interval between collections (minutes)</td>
<td>0.13</td>
</tr>
<tr>
<td>Global collections - Number of collections</td>
<td>5</td>
</tr>
<tr>
<td>Global collections - Total amount reclaimed (MB)</td>
<td>33.1</td>
</tr>
<tr>
<td>Largest memory request (bytes)</td>
<td>127764</td>
</tr>
<tr>
<td>Minor collections - Mean garbage collection pause (ms)</td>
<td>48.2</td>
</tr>
<tr>
<td>Minor collections - Mean interval between collections (ms)</td>
<td>7105</td>
</tr>
<tr>
<td>Minor collections - Number of collections</td>
<td>1-90</td>
</tr>
</tbody>
</table>
IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer

What problem am I solving
• Why did I run out of Java memory?
• What’s in my Java heap? How can I explore it and get new insights?

Overview
• Tool for analyzing heap dumps and identifying memory leaks from JVMs
• Works with IBM system dumps, heapdumps and Sun HPROF binary dumps
• Provides memory leak detection, footprint analysis and insight into wasted space
• Objects by Class, Dominator Tree Analysis, Path to GC Roots, Dominator Tree by Class Loader
• Provides SQL like object query language (OQL)
zEC12 – More Hardware for Java

Continued aggressive investment in Java on Z
Significant set of new hardware features
tailed and co-designed with Java

**Hardware Transaction Memory (HTM)** *(no zVM)*
Better concurrency for multi-threaded applications
eg. ~2X improvement to juc.ConcurrentLinkedQueue

**Run-time Instrumentation (RI)**
Innovation new h/w facility designed for managed runtimes
Enables new expanse of JRE optimizations

**2GB page frames** *(only on z/OS)*
Improved performance targeting 64-bit heaps

**Pageable 1MB large pages using flash** *(only on z/OS)*
Better versatility of managing memory

**New software hints/directives**
Data usage intent improves cache management
Branch pre-load improves branch prediction

**New trap instructions**
Reduce over-head of implicit bounds/null checks

Engineered Together—IBM Java and zEC12 Boost Workload Performance

Up-to 60% improvement in throughput amongst Java workloads measured with zEC12 and Java7SR3
Hardware Transactional Memory (HTM)

- **Allow lockless interlocked execution of a block of code called a ‘transaction’**
  - **Transaction**: Segment of code that appears to execute ‘atomically’ to other CPUs
    - Other processors in the system will either see **all-or-none** of the storage up-dates of transaction
  - **How it works:**
    - TBEGIN instruction starts speculative execution of ‘transaction’
    - Storage conflict is detected by hardware if another CPU writes to storage used by the transaction
    - Conflict triggers hardware to roll-back state (storage and registers)
      - transaction can be re-tried, or
      - a fall-back software path that performs locking can be used to guarantee forward progress
    - Changes made by transaction become visible to other CPUs after TEND

---

**Example:**

**CPU 0: Tran A**

```
TBEGIN
...
load Y
load X
...
TEND
```

CPU 0 can only see \((X=Y=0)\) or \((X=Y=1)\), cannot see \((X=1,Y=0)\) or \((X=0,Y=1)\)

**CPU 1: Tran B**

```
X = Y = 0;
TBEGIN
X = 1
store X
Y = 1
store Y
TEND
```

Storage conflict: Tran A will abort Tran B will commit changes to X and Y

---

**Note:**

Complete your sessions evaluation online at SHARE.org/SanFranciscoEval
HTM Example: Transactional Lock Elision (TLE)

Threads must serialize despite only reading... just in-case a writer updates the hash

```
read_hash(key) {
  Wait_for_lock();
  read(hash, key);
  Release_lock();
}
```

Lock elision allows readers to execute in parallel, and safely back-out should a writer update hash

```
read_hash(key)
  TRANSACTION_BEGIN
  read hash.lock;
  BRNE serialize_on_hash_lock read (hash, key);
  TRANSACTION_END
```

---

**e·lide** /ihˈlaɪd/ verb (used with object), -lid·ed, -lid·ing.
1. to omit (a vowel, consonant, or syllable) in pronunciation.
2. to suppress; omit; ignore; pass over.
3. Law. to annul or quash.
Transactional Execution: Concurrent Linked Queue

- ~2x improved scalability of juc.ConcurrentLinkedQueue
- **Unbound Thread-Safe LinkedQueue**
  - First-in-first-out (FIFO)
  - Insert elements into tail (en-queue)
  - Poll elements from head (de-queue)
  - No explicit locking required
- **Example usage: a multi-threaded work queue**
  - Tasks are inserted into a concurrent linked queue as multiple worker threads poll work from it concurrently

(new diagram showing operations like En-queue and De-queue with nodes)

(Chart showing benchmark with Java prototype)

New TX-base implementation  
Traditional CAS-base implementation

(Controlled measurement environment, results may vary)
Linux on System z and Java7SR3 on zEC12:
64-Bit Java Multi-threaded Benchmark on 16-Way

Aggregate 60% improvement from zEC12 and Java7SR3
- zEC12 offers a ~45% improvement over z196 running the Java Multi-Threaded Benchmark
- Java7SR3 offers an additional ~10% improvement (-Xaggressive)
Linux on System z and Java7SR3 on zEC12:
64-Bit Java Multi-threaded Benchmark on 16-Way

~12x aggregate hardware and software improvement comparing Java5SR4 on z9 to Java7SR3 on zEC12
LP=Large Pages for Java heap  CR= Java compressed references
Java7SR3 using -Xaggressive + 1Meg large pages

(Controlled measurement environment, results may vary)
WAS on zLinux –
Aggregate HW, SDK and WAS Improvement: WAS 6.1 (Java 5) on z9 to WAS 8.5 (Java 7) on zEC12

~4x aggregate hardware and software improvement comparing WAS 6.1 Java5 on z9 to WAS 8.5 Java7 on zEC12

(Controlled measurement environment, results may vary)
Questions

Marcel Mitran
mmitran@ca.ibm.com
Thank You

© Copyright IBM Corporation 2012. All rights reserved. The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. Product release dates and/or capabilities referenced in these materials may change at any time at IBM’s sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way. IBM, the IBM logo, Rational, the Rational logo, Telelogic, the Telelogic logo, and other IBM products and services are trademarks of the International Business Machines Corporation, in the United States, other countries or both. Other company, product, or service names may be trademarks or service marks of others.
Summary of Links

- Documentation
- zOS SDK
- System z Linux SDK
- GC Tuning documentation
- IBM Support Assistant