Marcel Mitran, Senior Technical Staff Member
Java and Compilers for System z
mmitran@ca.ibm.com

Java SDK5, SDK6 and Beyond: A Performance Update

SHARE Session 9997, August 8th, 2011

Agenda

- Overview/Roadmap
- Java on z/OS
- IBM Java Runtime Environment for System z
  - Compressed references
- Java 6.0.1/Java6 R26 performance
- Inter-Language Communication

Appendix

A. IBM Java Consumability
B. zAAPs
C. IBM Testarossa JIT
D. IBM J9 Garbage Collector
E. 64-bit Perfromance
F. IBM System z Hardware and Java
G. Links and Documentation

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**
  - Eclipse, Apache (XML, Aries, Derby, Geronimo, Harmony, Tuscany, Hadoop …)
  - Broad participation in relevant open standards (JCP, OSGi)

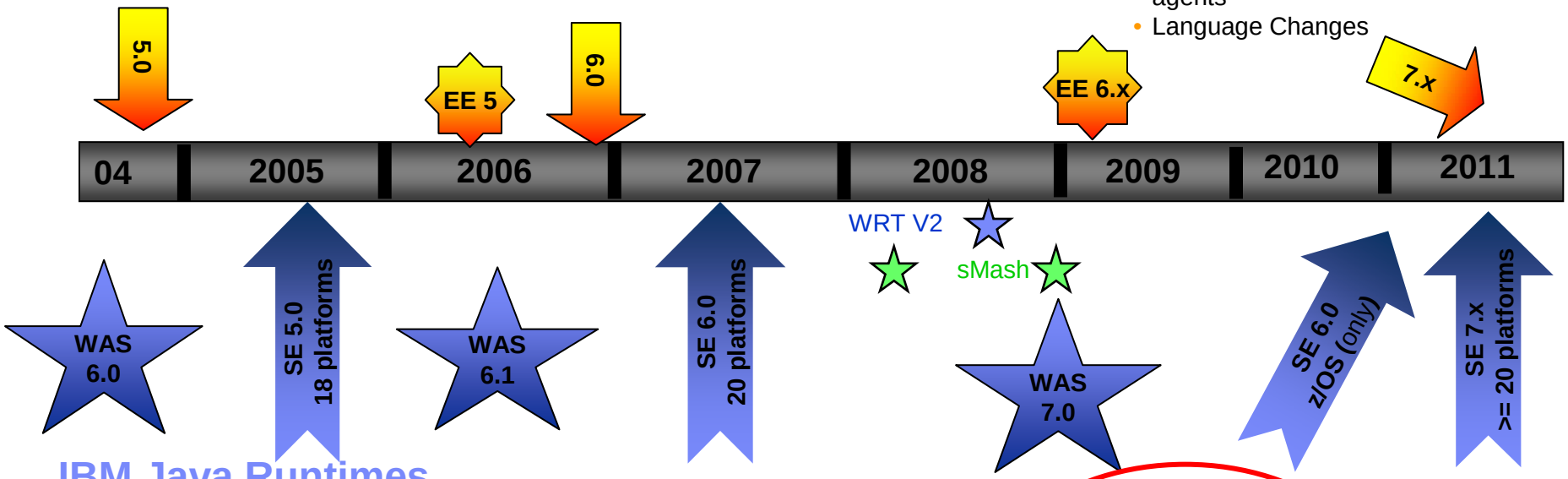
Java Road Map

Oracle Java Runtimes

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

Java 6.0
- Performance Improvements
- Client WebServices Support

Java 7.x
- 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



IBM Java Runtimes

IBM Java 5.0
- 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

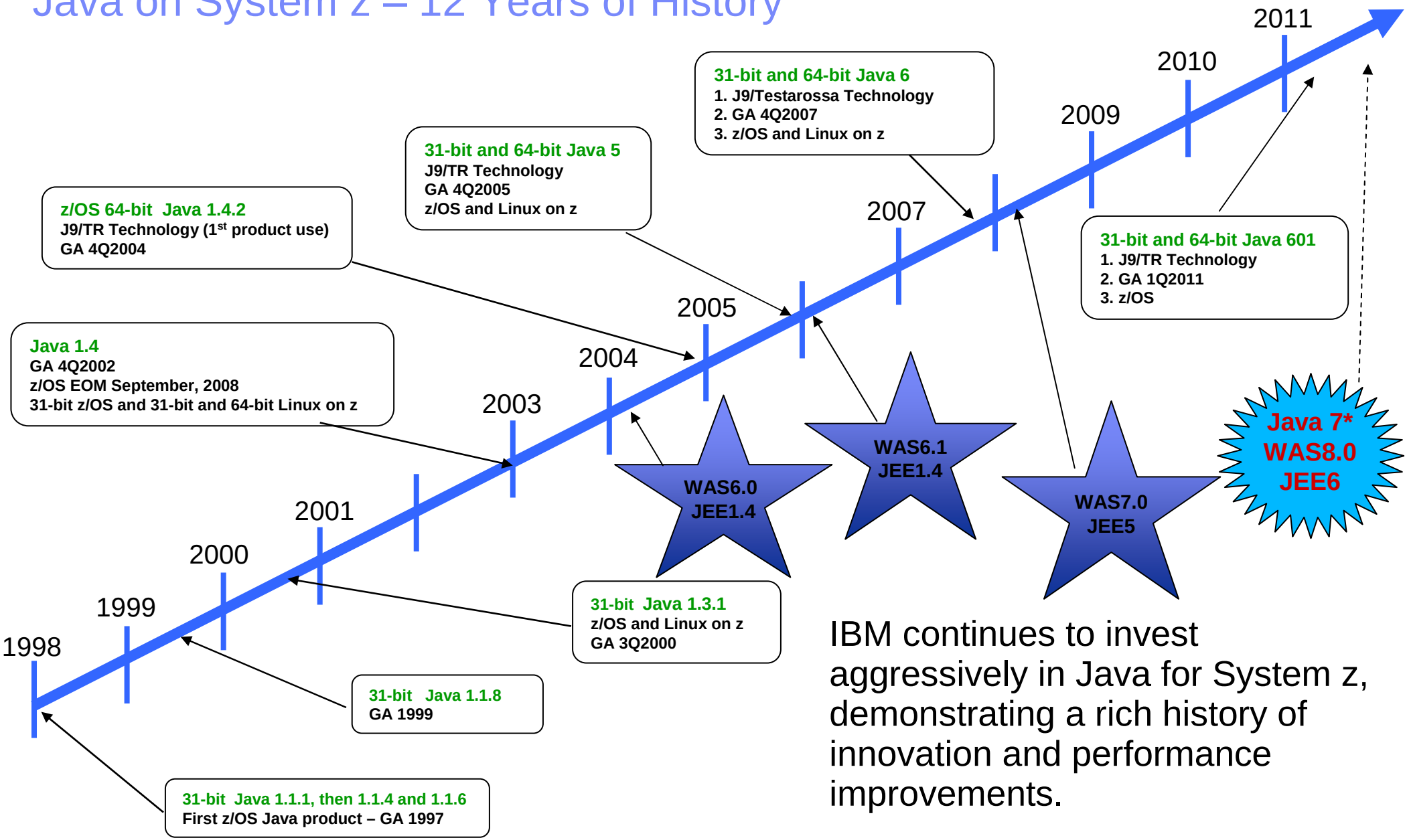
IBM Java 6.0
- Improvements in
  - Performance
  - Serviceability tooling
  - Class Sharing
- XML parser improvements
- z10™ Exploitation
  - DFP exploitation for BigDecimal
  - Large Pages
  - New ISA features

IBM Java 6.0.1
- Improvements in
  - Performance
  - GC Technology
- z196™ Exploitation
  - OOO Pipeline
  - 70+ New Instructions
- JZOS/Security Enhancements

IBM Java 7.x
- Improved serviceability and consumability
- Improved performance
- VM updates in support of language changes

Timelines and deliveries are subject to change.

Java on System z – 12 Years of History

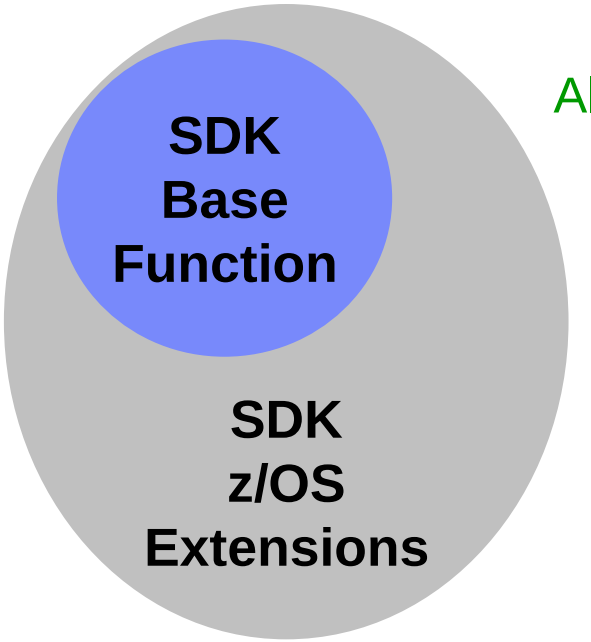


IBM continues to invest aggressively in Java for System z, demonstrating a rich history of innovation and performance improvements.

*** Java7 Beta:** https://www.ibm.com/developerworks/java/jdk/beta/index.html
Testimonials: http://www-01.ibm.com/software/os/systemz/testimonials/

Timelines and deliveries are subject to change.

z/OS – System z Java Extensions



All SDKs support the 'standards', Java on z/OS extends the SDK

- Access to z/OS services
- Access to all types of data
- Access under control of z/OS security mechanisms
- Integration into existing operational infrastructure

Services available in JEE and JSE environments under the restrictions of the container.

System specific extension allow you to write robust middleware and applications that integrate with traditional z/OS operating environment

- Allow for maintaining platform independent design development.
- Platform specific implementations when required
- Allows for operational and resource optimization

e.g. JAAS wrapper of SAF (RACF, ACF2, or TopSecret), Traditional OS dataset access, Cryptographic hardware (Cards and CPACF), z/OS Console (modify and messages), z/OS system logger, JES job submission, DFSORT, SMF, etc.

Java Execution Environments and Interoperability

- Capitalize on pre-existing assets, artifacts, processes, core competencies, platform strengths

- **IBM Java Execution Offerings**
  - Transactional/Interactive
    - WebSphere for z/OS (WAS z/OS)
    - WebSphere Process Server for z/OS (WPS)
    - JCICS
    - IMS Java
    - DB2 Stored Procedures
  - Batch oriented
    - WebSphere Compute Grid (WAS-CG)
      - WAS/JEE runtime extensions
    - JZOS component of z/OS SDK
      - JES/JSE-based environment
    - z/OS V1R13 Java/COBOL Batch Runtime Env.*
      - JES/JSE-based, designed to inter-op with DB2 while maintaining transaction integrity

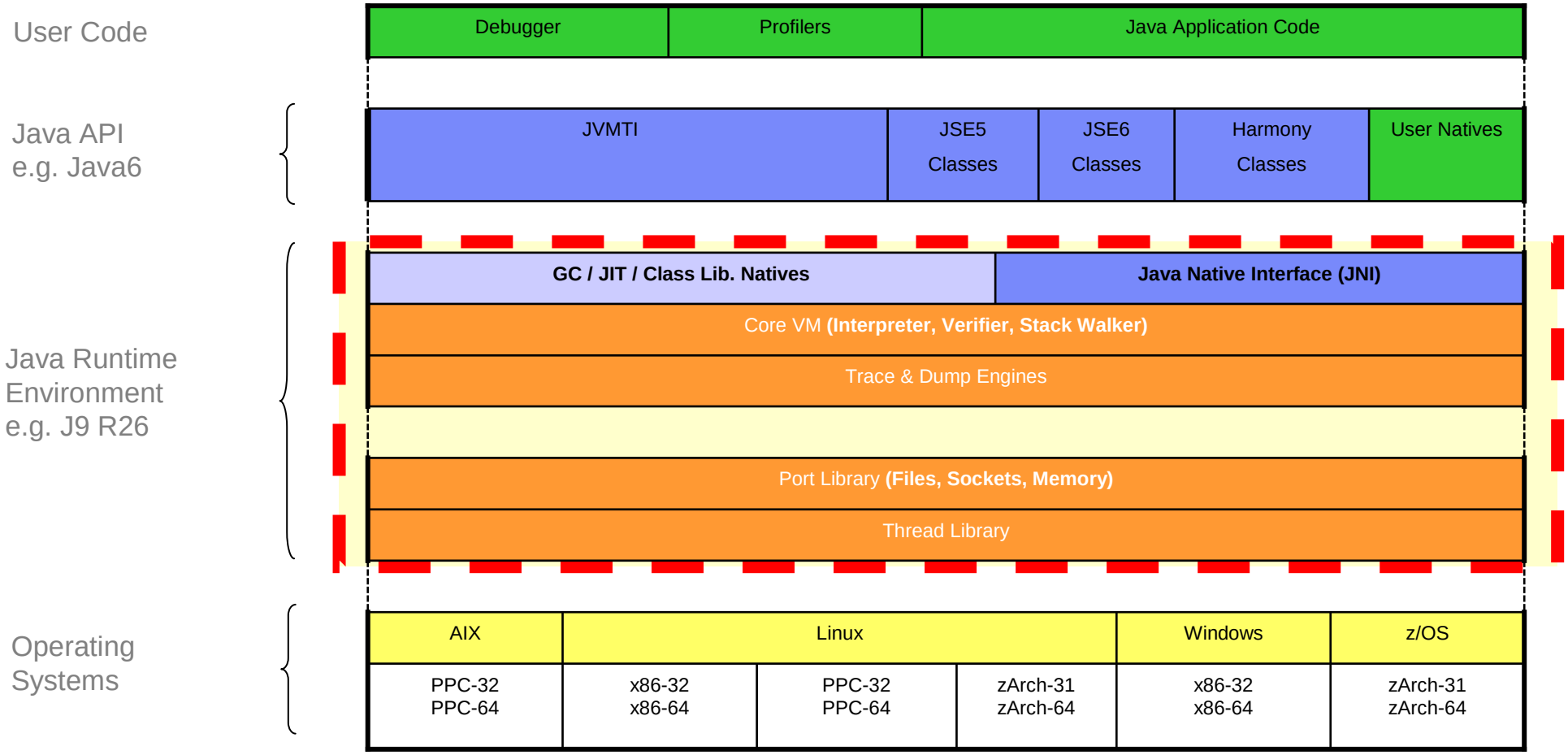
- **Open Source or non-IBM vendor Application Server and Frameworks**
  - Tomcat, JBoss
  - iBatis, Hibernate, Spring
  - Ant

- **COBOL/Native Interoperability**
  - COBOL Invoke maps to JNI
  - RDz and JZOS** have tooling to map COBOL copy books to Java classes
  - JCICS
  - IMS Java, JMP/JBP
  - WAS CG, WOLA
  - etc

* See http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?subtype=ca&infotype=an&supplier=897&letternum=ENUS211-252

** Alphaworks only, and hence currently un-supported

JVM Architectural Overview



Java 6.0.1:

- Also referred to as Java6 R26, ships with WAS8 across platforms, or standalone on z/OS
- Fully compatible/compliant Java6 (JSE6)
- Includes new J9 R26 JRE (replacing J9 R24 in Java6.0.0)
  - Transparent z196 and new optimization exploitation
  - New balanced GC policy

= User Code
= Java Platform API
= VM-aware
= Core VM

IBM Java Runtime Environment

- IBM's implementation of Java 5 and Java 6 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 – z196/z10/z9/z990 exploitation
  - Cost-effective for z - zAAP Ready!

- Millions of instances of J9/TR compiler

Compressed References

- Option to enable compression in 64-bit Java 6 SR4, WAS 7 (SPack3)
  - use –Xcompressedrefs
- Java objects are 8-byte aligned
  - Low 3 bits of object address = 000
- Address range restriction: Java heap allocated in 2^{31} – 2^{35} range (2GB – 32GB virtual)
  - High 29 bits of object address = 000 … 000
  - 32 out 64 bits are 0!
- Main idea is to store 32-bit shifted offset in objects
  - Shift values of 0 through 3 are used
- Maximum allowable heap is 30GB

Compressed References: Technical Details

- **32-bit Object (24 bytes – 100%)**

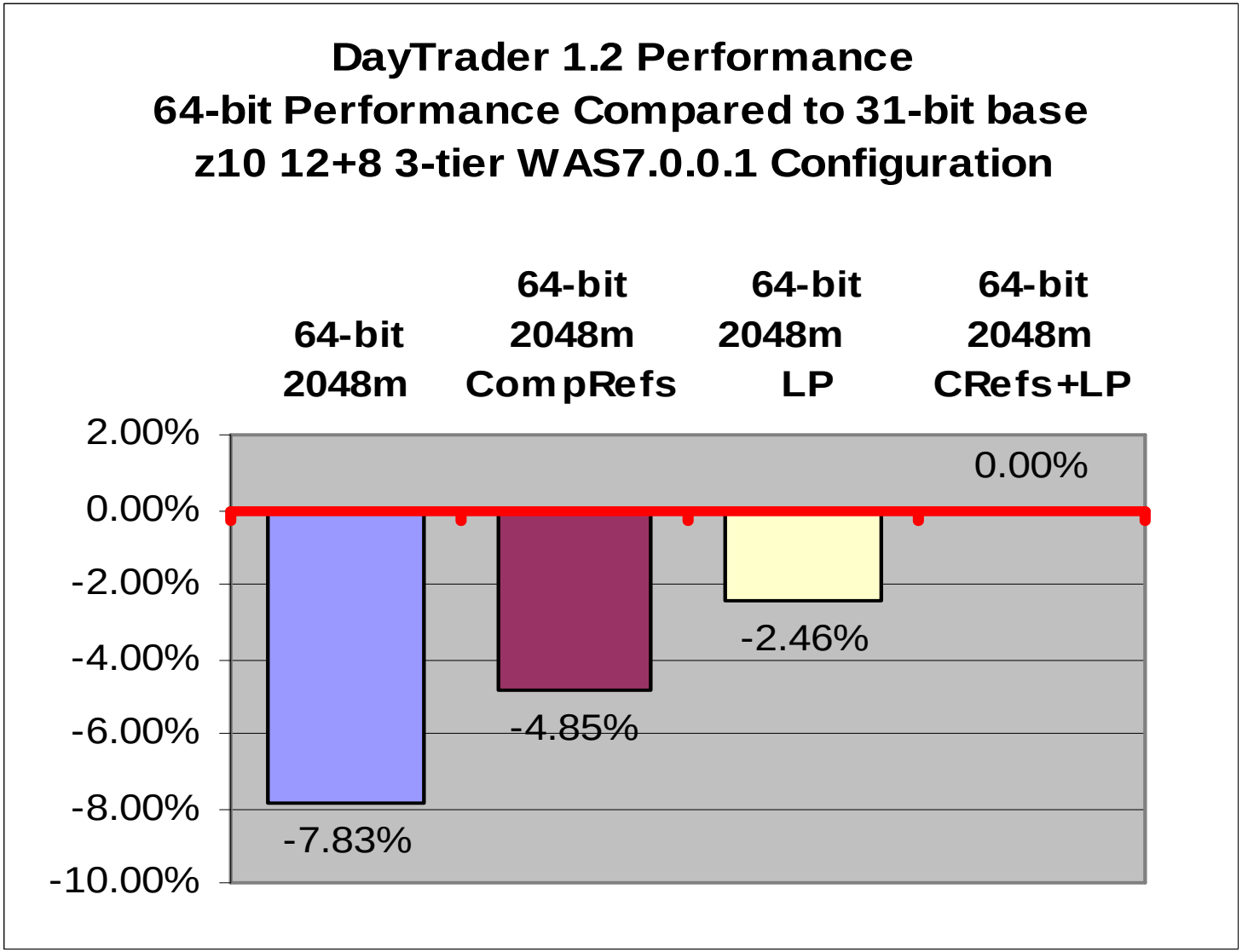
- **64-bit Object (48 bytes – 50%)**

- **64-bit Compressed References (24 bytes – 100%)**

Use 32-bit values (offsets) to represent object fields

With scaling, between 4 GB and 32 GB can be addressed

WebSphere 64- vs. 31-bit Performance



Controlled measurement environment, results may vary.

IBM Java 6.0.1 on z/OS

z196 and Java6.0.1: Engineered Together

- Up-to 2.1x improvement to Java throughput
- Reduced footprint
- Tighter integration with z/OS facilities
- Improved responsiveness in application behavior

J9 R2.6 Virtual Machine

- Significant enhancements to JIT optimization technology
- z196 exploitation of instructions and new pipeline
- New Balanced GC policy to reduce max pause times
- Default GC policy changed to gencon

z/OS Unique Enhancements

- JZOS 2.4.0
- z/OS Java unique security enhancements

Performance

- 2.1x improvement to multi-threaded workload
- 1.93x improvement to CPU-intensive workload

IBM Testarossa JIT - Deep System z Exploitation

Out-of-Order/Super-scalar Instruction Scheduler

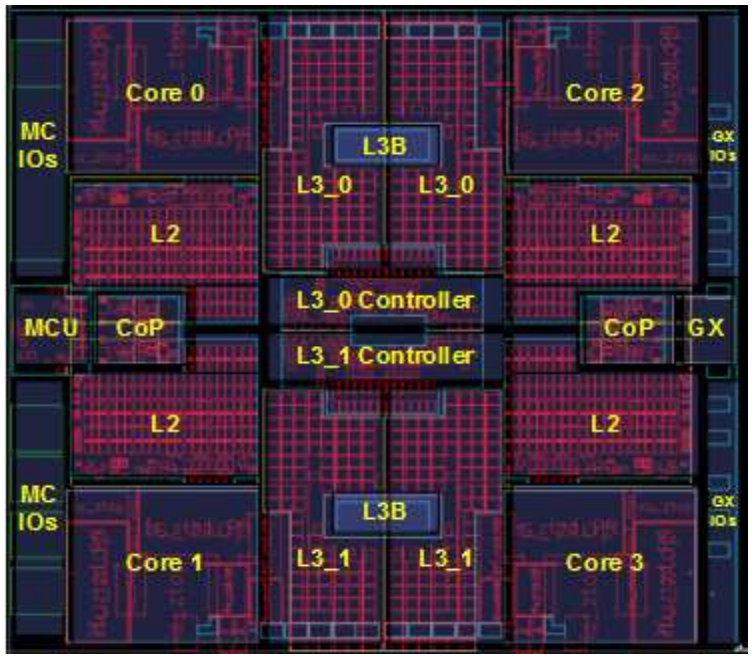
- z10/z9/z990 are in order superscalar dual pipelines, can dispatch up to 3 instrs/cycle
- z196 pipeline is OOO, permits 3 inst/group

Platform Tuned Optimizations

- idiom recognition, dynamic literal pool, etc

New Hardware Facility Exploitation

- z196: high-word, non-destructive, interlock, conditional load/store
- z10: traps, compare-and-branch, pre-fetch, Decimal Floating Point, etc
- z9: extended immediate support
- z990: long displacement support



IBM J9 2.6 and z196

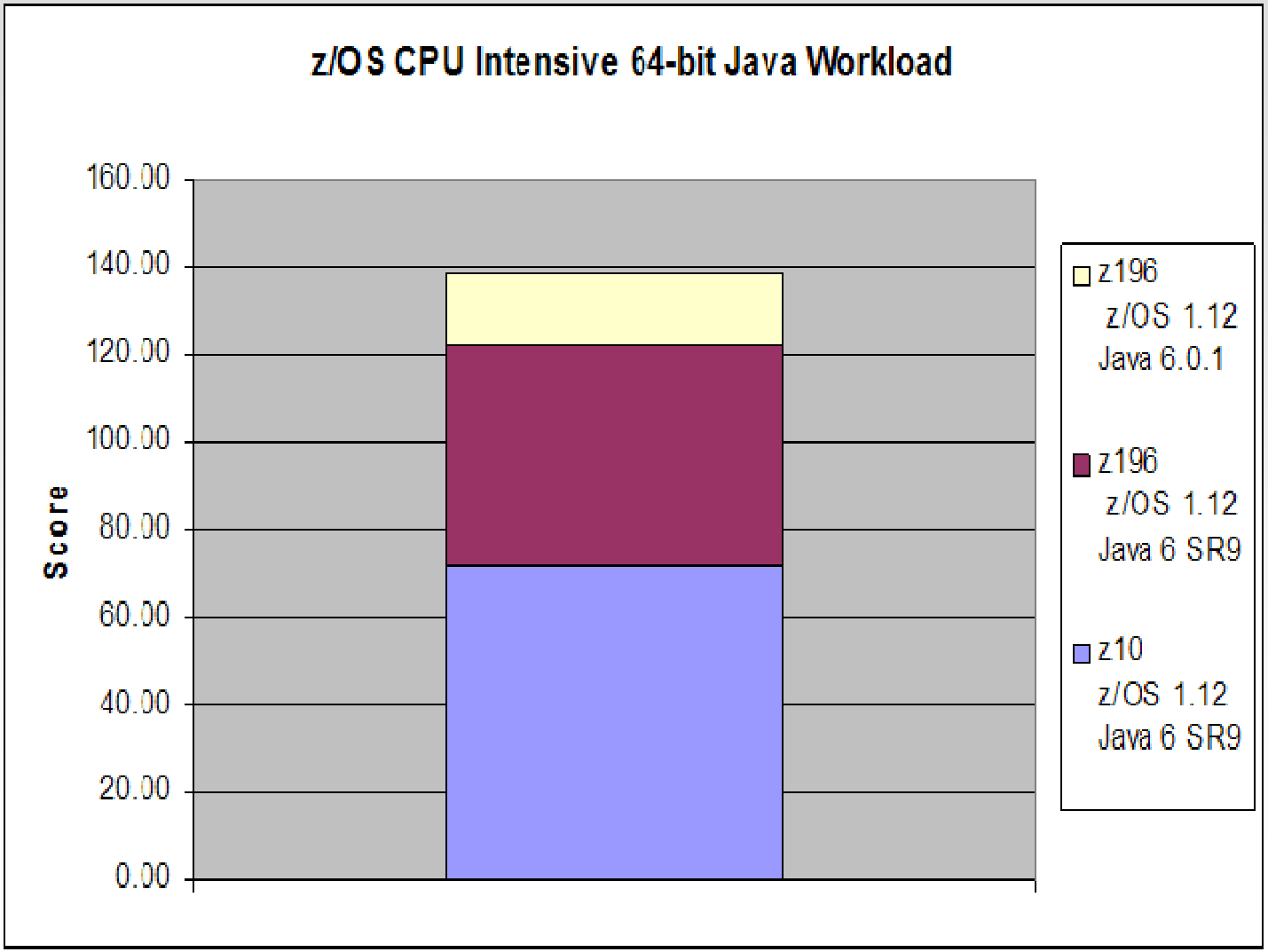
z196: Hardware for Java

- New Out-Of-Order pipeline design
- New larger cache structure
- Higher clock speed (~5.2GHz)

J9 R26: JRE for z196

- Reducing pressure on the data/instruction cache
  - Enables better exploitation of new compute bandwidth
  - Mitigates effects of cache latencies for leveraging core speed
- Concurrency improvements
  - Better scalability
- General optimizer and codegen improvements
  - Reduced path-length

Performance on z/OS: CPU-Intensive Benchmark

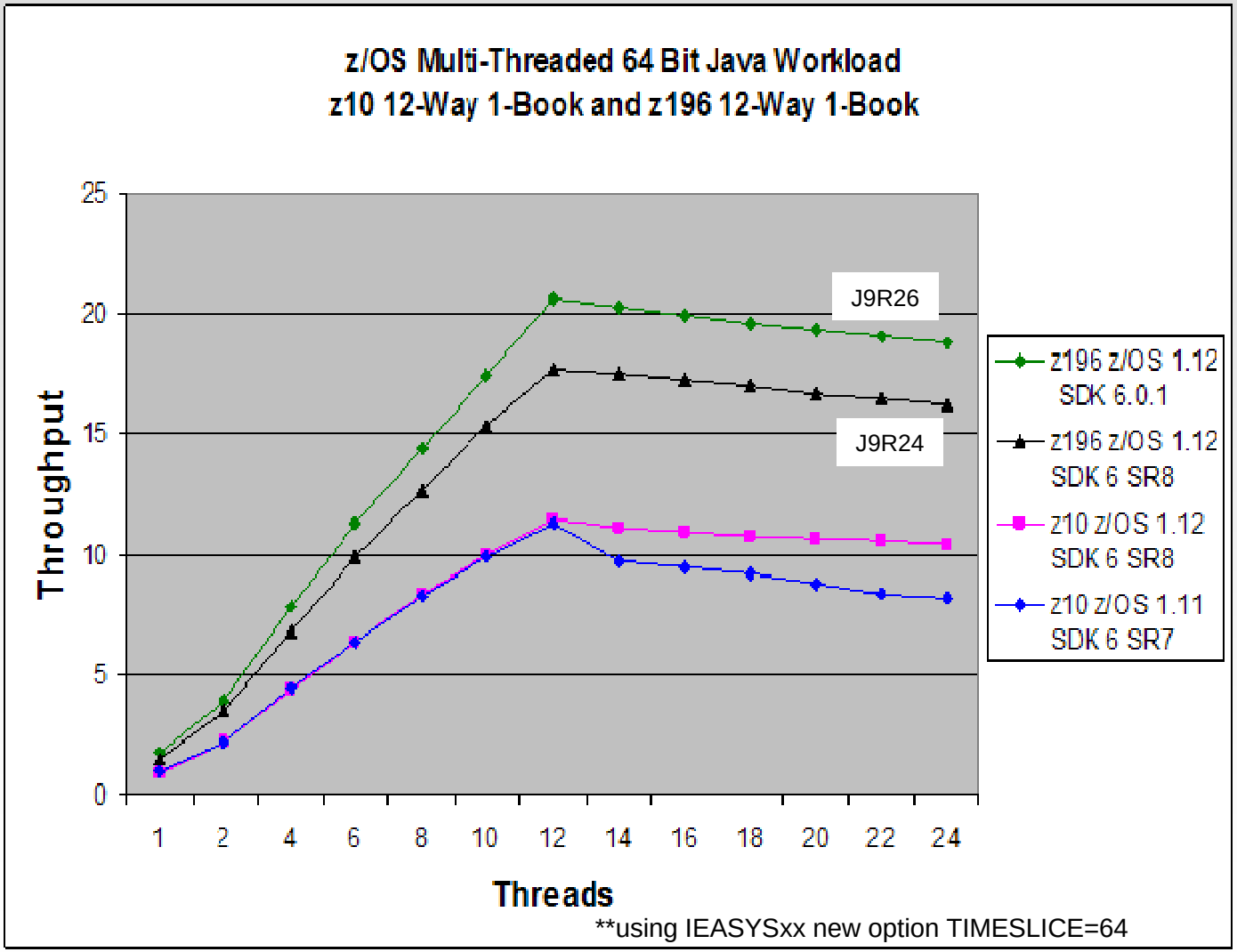


93% Aggregate improvement

- 14% Java 6.0.1 improvement
- 70% Hardware improvement

(Controlled measurement environment, results may vary)

Performance on z/OS: Multi-threaded Benchmark

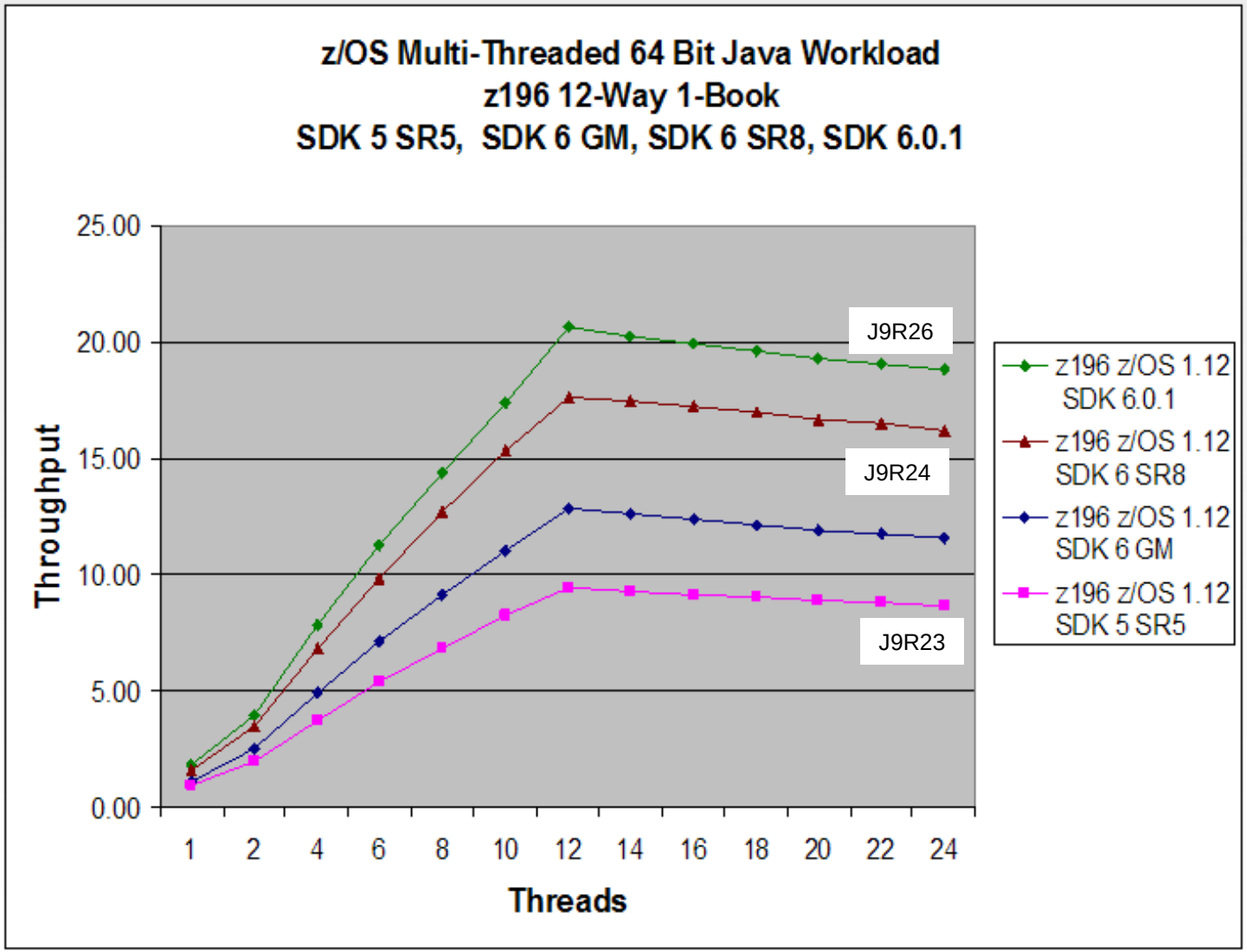


2.1x Aggregate improvement

- 16% Java 6.0.1 improvement
- 56% Hardware improvement
- 17% z/OS 1.12 improvement**

(Controlled measurement environment, results may vary)

z/OS Java SDK 6.0.1 Performance: 64 Bit Multi-threaded Benchmark



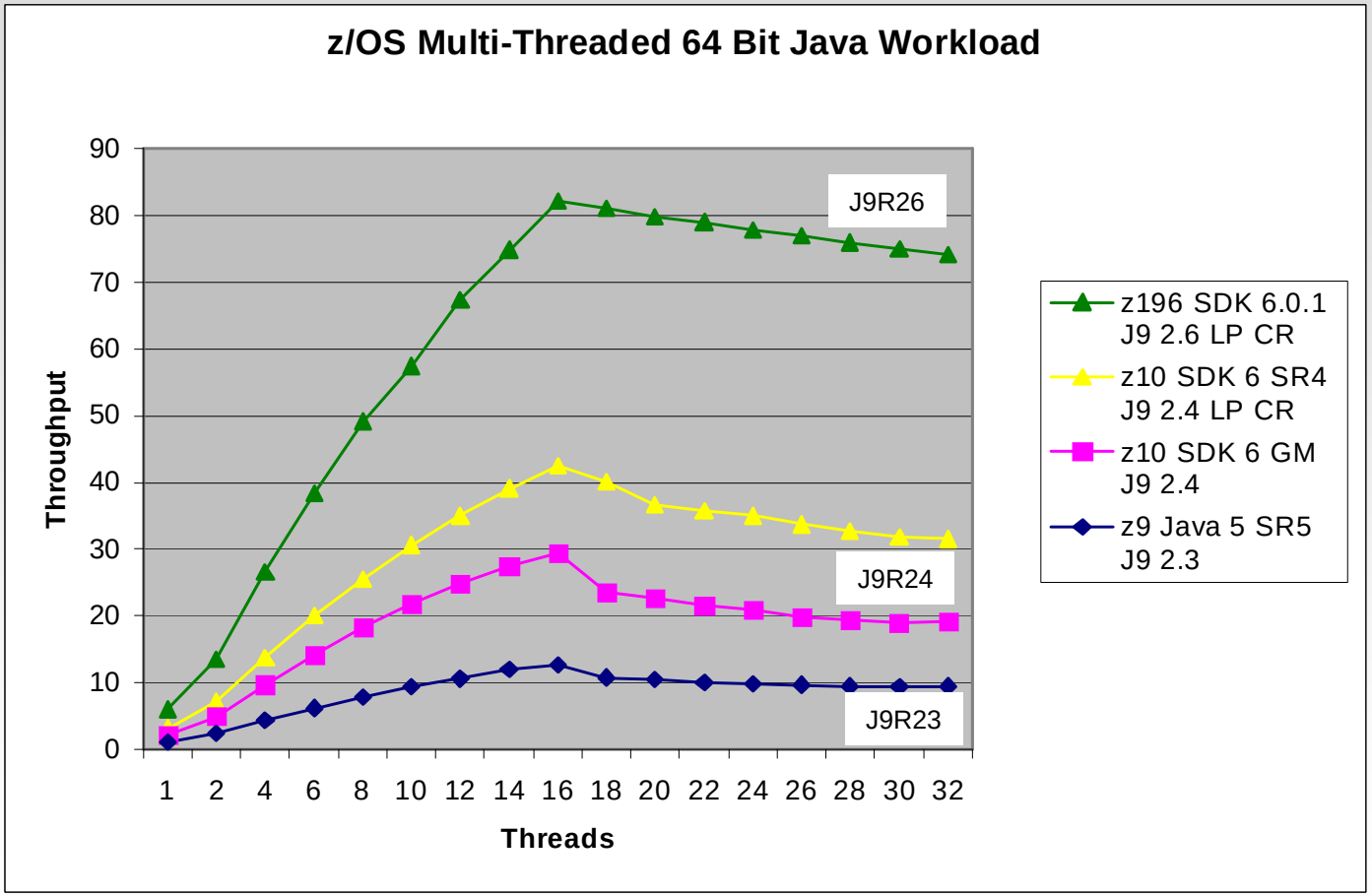
2.17x Aggregate Software improvement

- 16% Java 6.0.1 improvement
- 39% Java 6 SR8 versus Java 6 GM improvement
- 35% Java 6 GM versus Java 5 SR5

(Controlled measurement environment, results may vary)

z/OS Java SDK 6.0.1 Performance

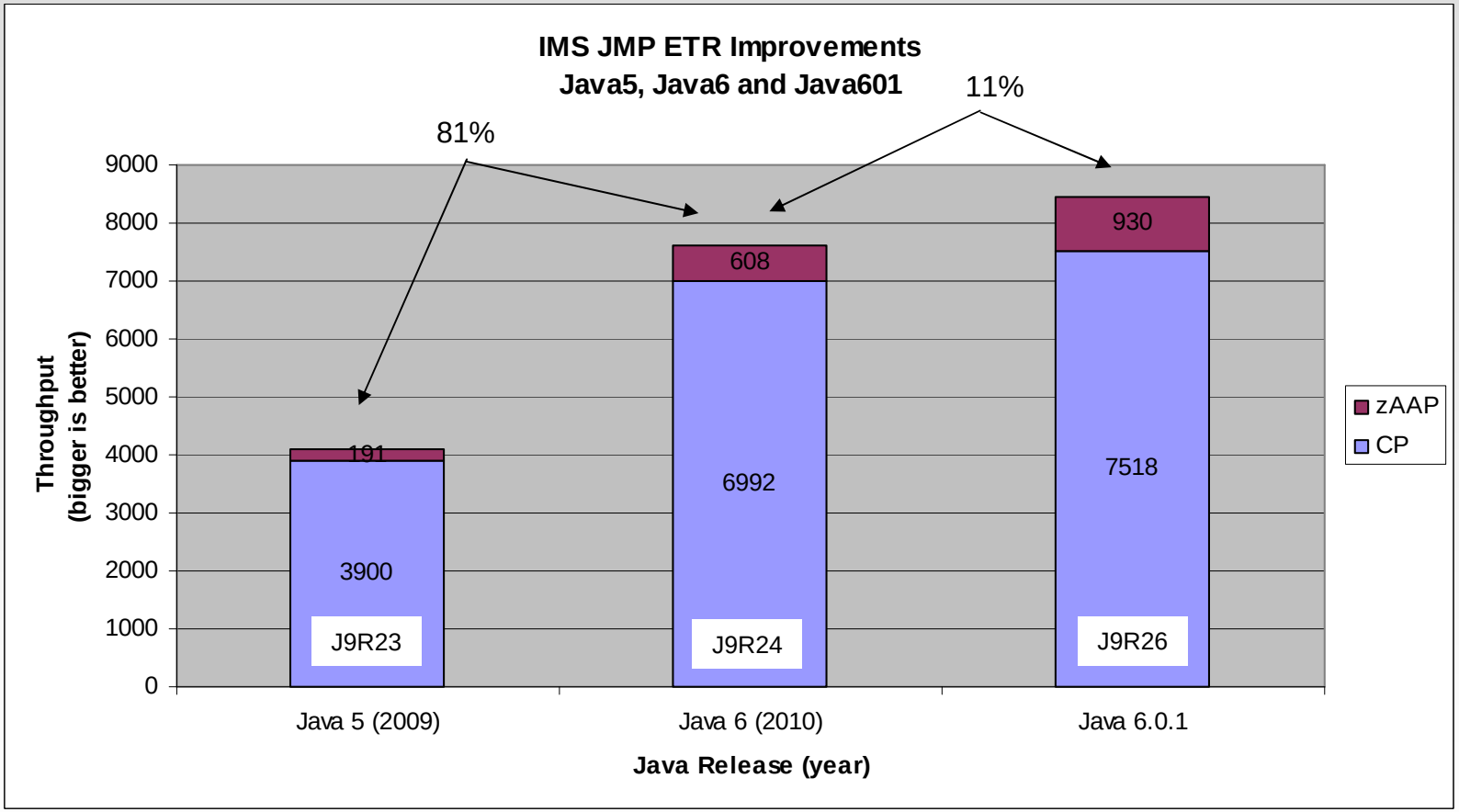
Aggregate HW and SDK Improvement z10, z196, Java6 to Java6.0.1



~7x Improvement from z10, z196, Java6 and Java6.0.1

(Controlled measurement environment, results may vary)

Performance – IMS JMP

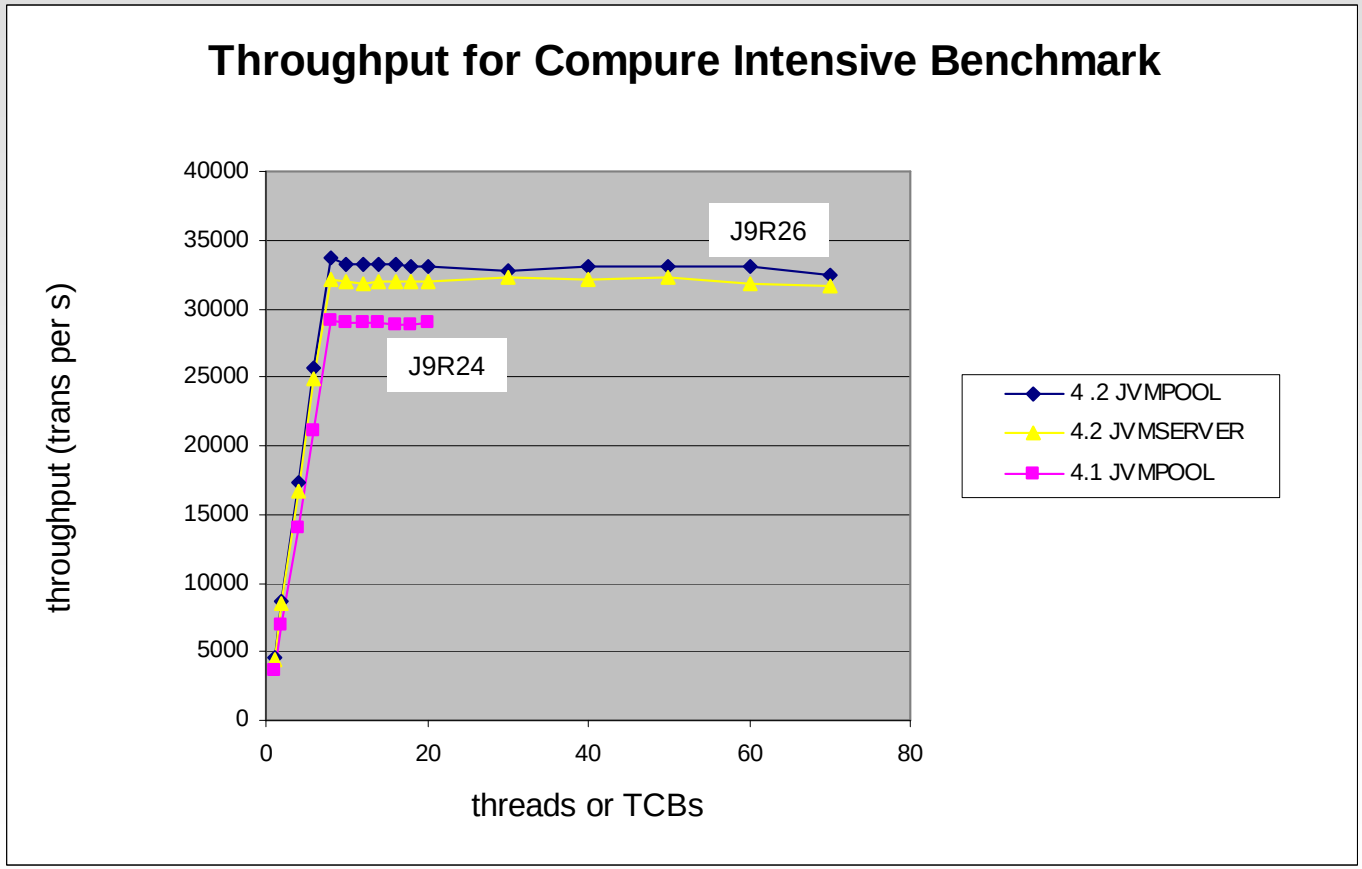


2 GCP + 2 zAAP

(Controlled measurement environment, results may vary)

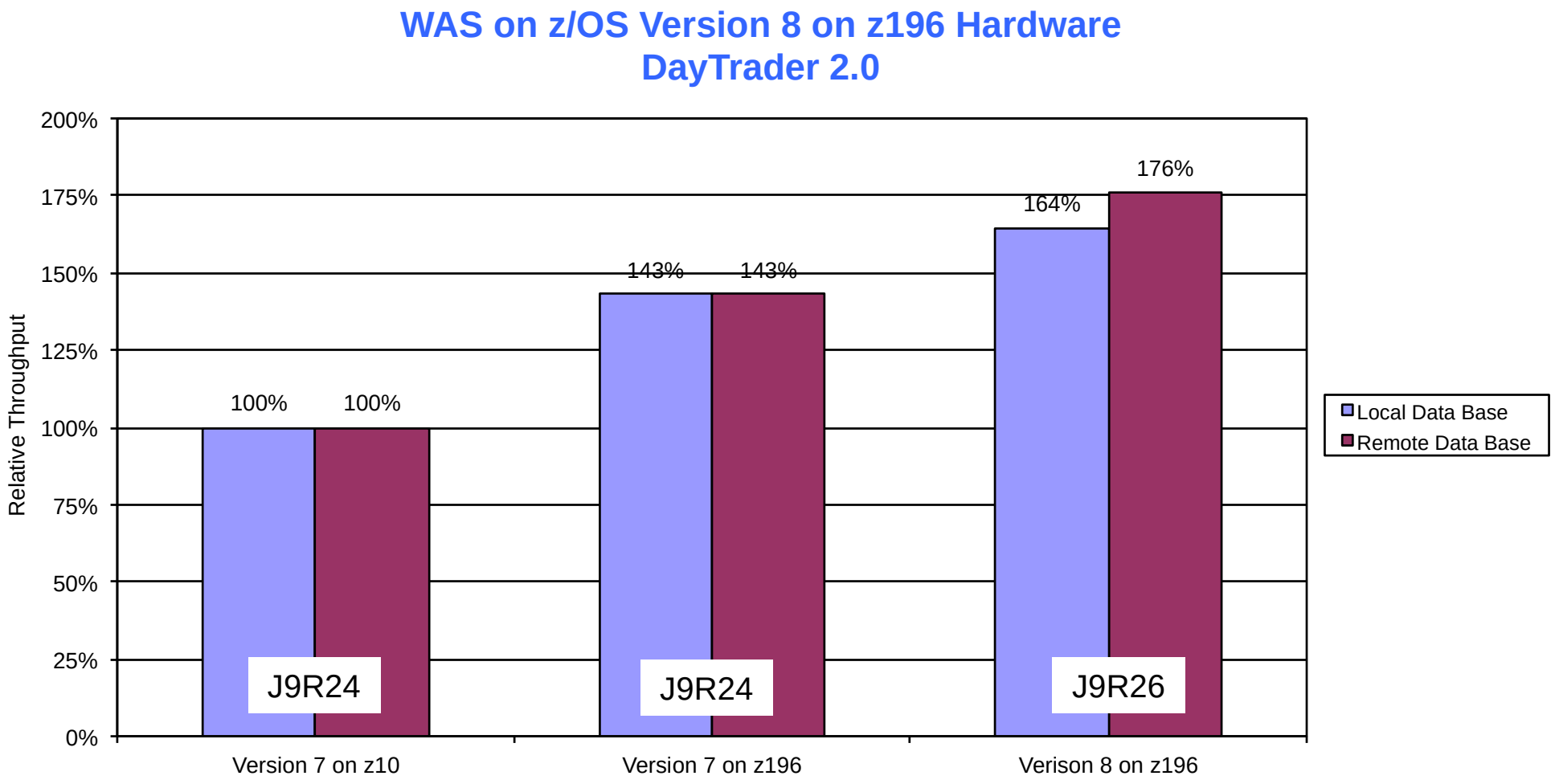
z196™ – z/OS V1.12

Comparing Java Throughput on CICS 4.2 with CICS 4.1



- **CICS 4.2 JVMPOOL and JVMSERVER use 64-bit Java 6.0.1 relieving 31-bit storage constraint**
- **~17% improvement to throughput with CICS 4.2/Java 6.0.1**
- **JVMSERVER slightly more expensive than JVMPOOL in CPU usage but requires less memory**
- **All configurations scale well**

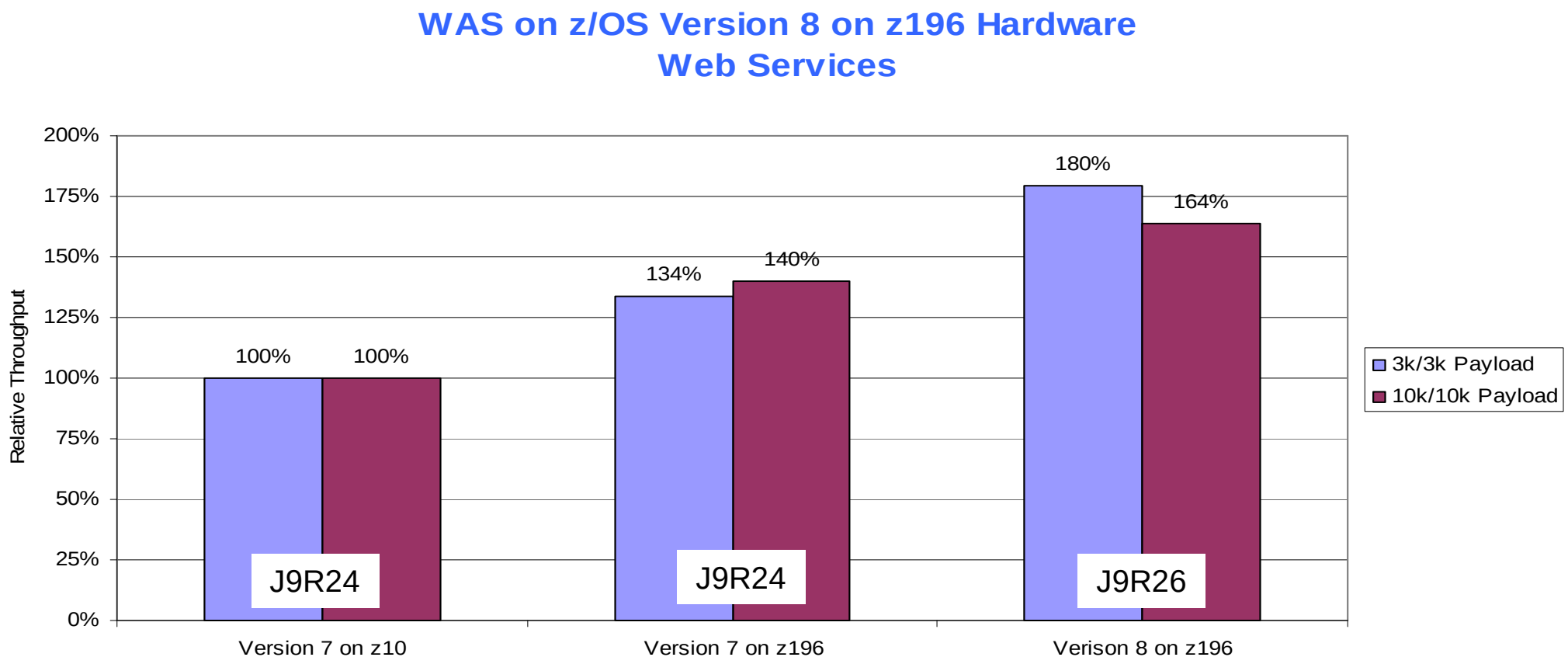
Performance on z/OS: WAS on z/OS



- z196 hardware measured 43% more throughput for local and remote database configurations
- Version 8 improved throughput of 2-tier configurations by another 15% for an aggregate benefit of 64%
- Version 8 improved throughput of 3-tier configurations by another 23% for an aggregate benefit of 76%

(Controlled measurement environment, results may vary)

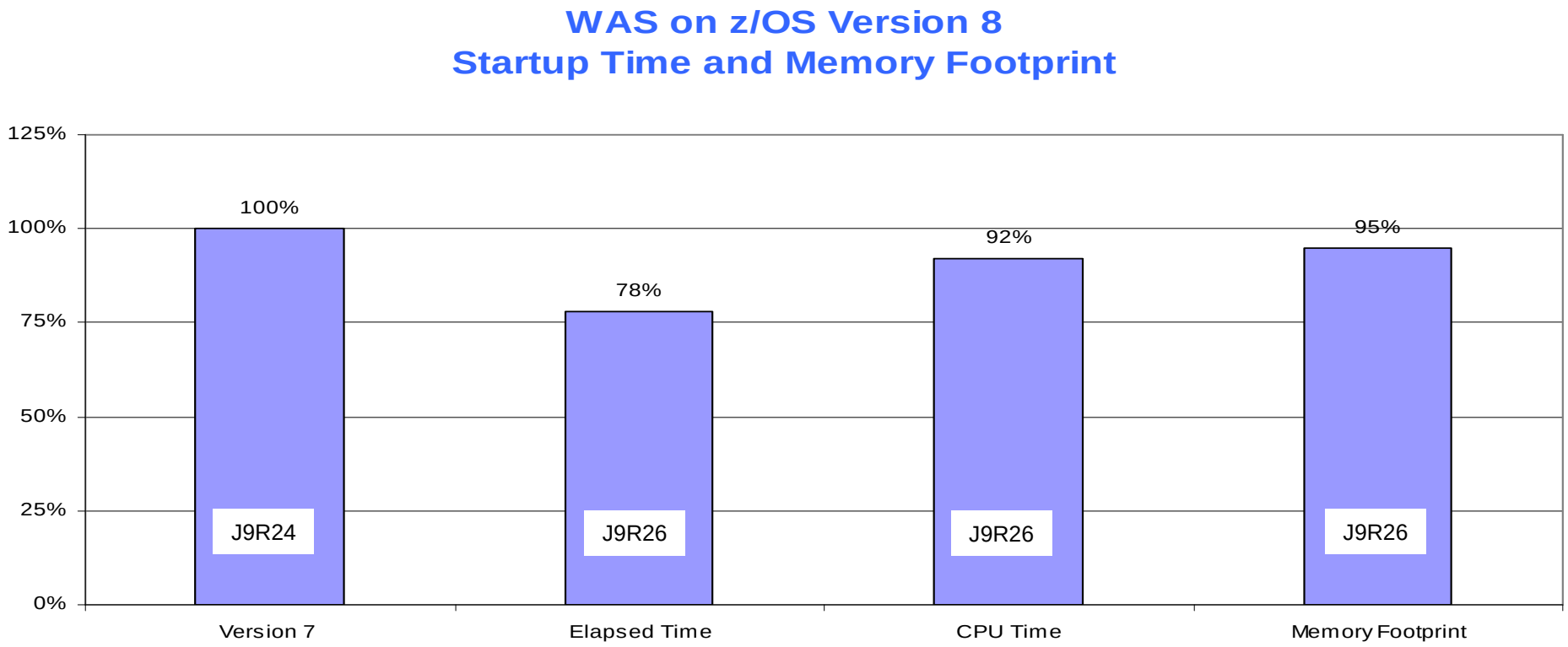
Performance on z/OS: WAS on z/OS



- z196 hardware measured 34% more throughput for small payload sizes (3kin/3kout) and 40% more for typical payload sizes (10kin/10kout) than z10 hardware
- Version 8 throughput improved over Version 7 by another 34% with the 3k/3k payload and 17% with the 10k/10k payload for and aggregate hardware and software benefit of +80% and +64% respectively
  - Improved JAXB parsing

(Controlled measurement environment, results may vary)

Performance on z/OS: WAS on z/OS

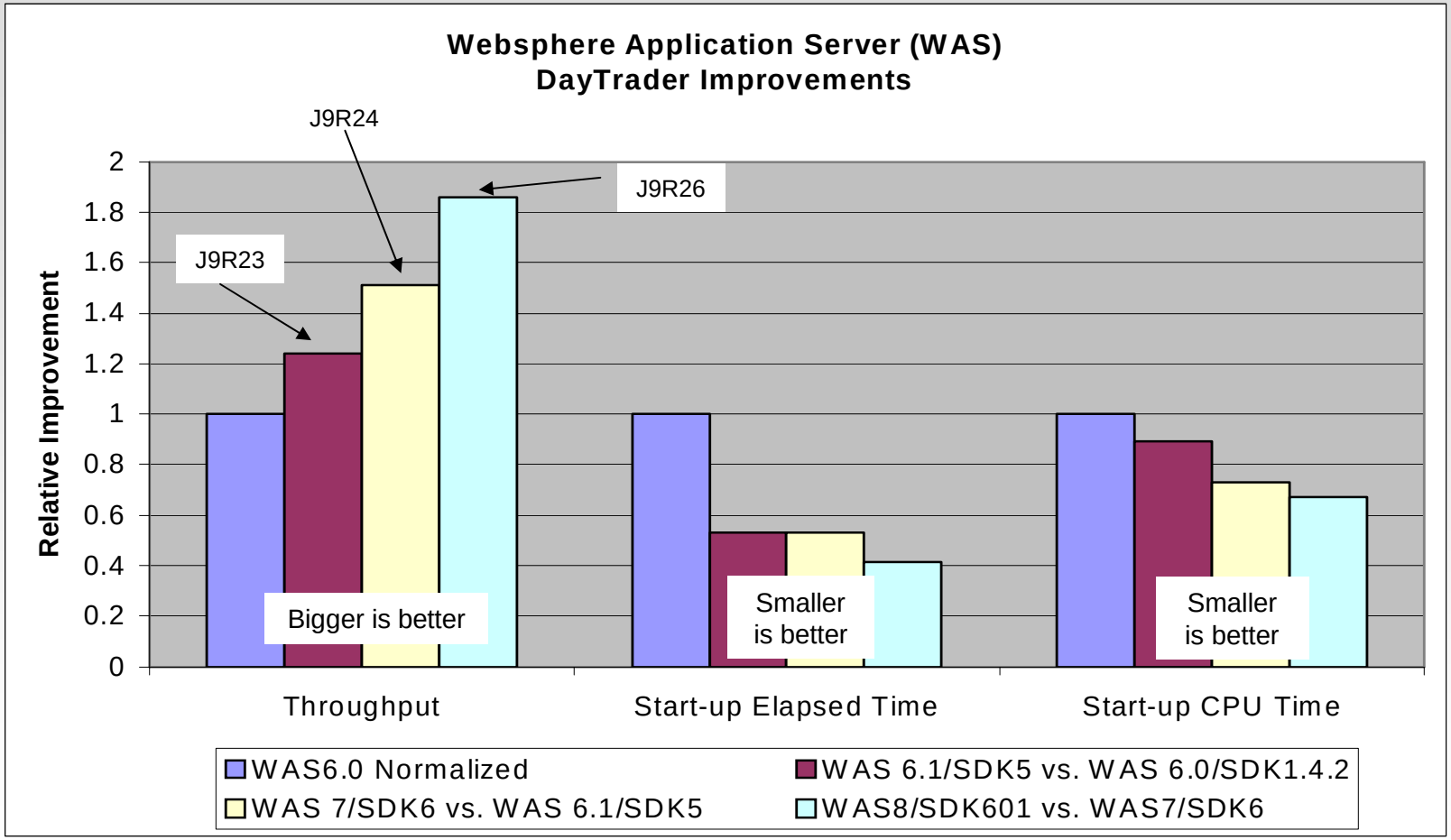


- Version 8 server startup time has been reduced by 22%, in elapsed time and 8% in CPU time compared to Version 7
  - Larger shared class cache reduced class load times
  - Optimized annotation scanning
  - Only delegate class loading to the JDK class loader instead of all class loaders for JDK classes
- Version 8 memory footprint has been reduced by 5%
  - Reductions in JVM native memory as well as class memory

(Controlled measurement environment, results may vary)

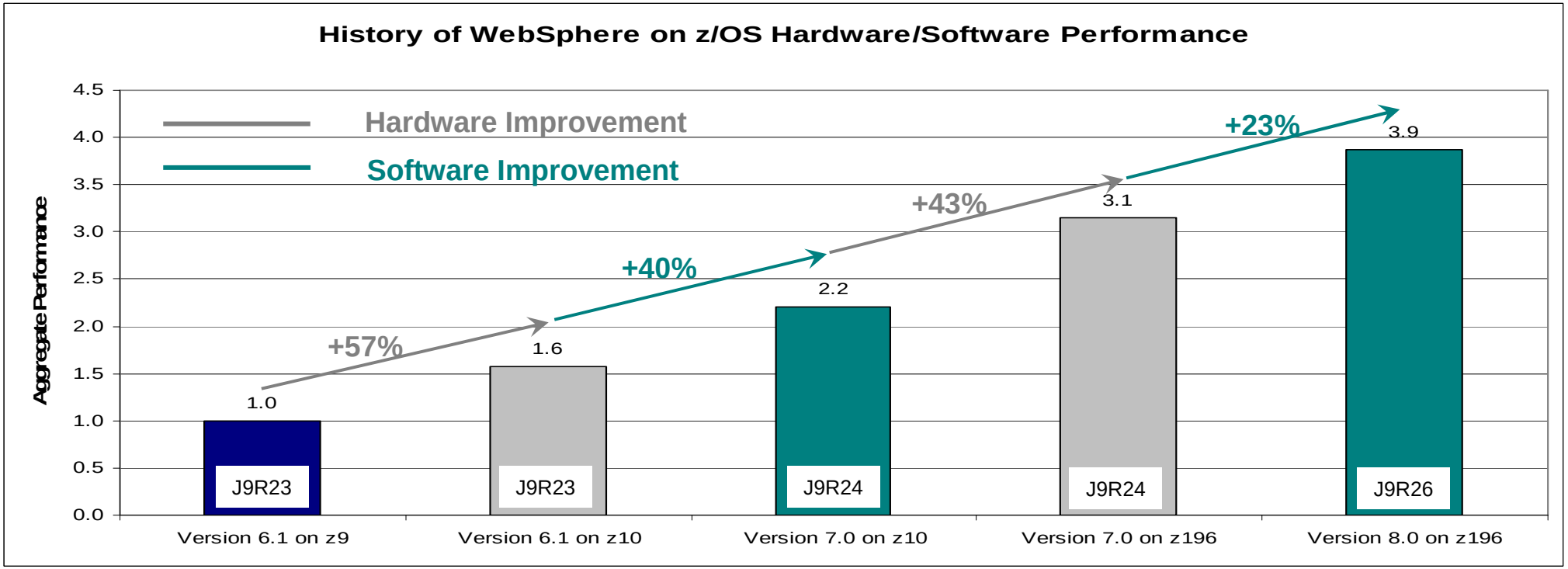
Performance – WAS on zOS

(WAS6.0 - WAS6.1 - WAS7.0 - WAS8.0 DayTrader on z/OS)



(Controlled measurement environment, results may vary)

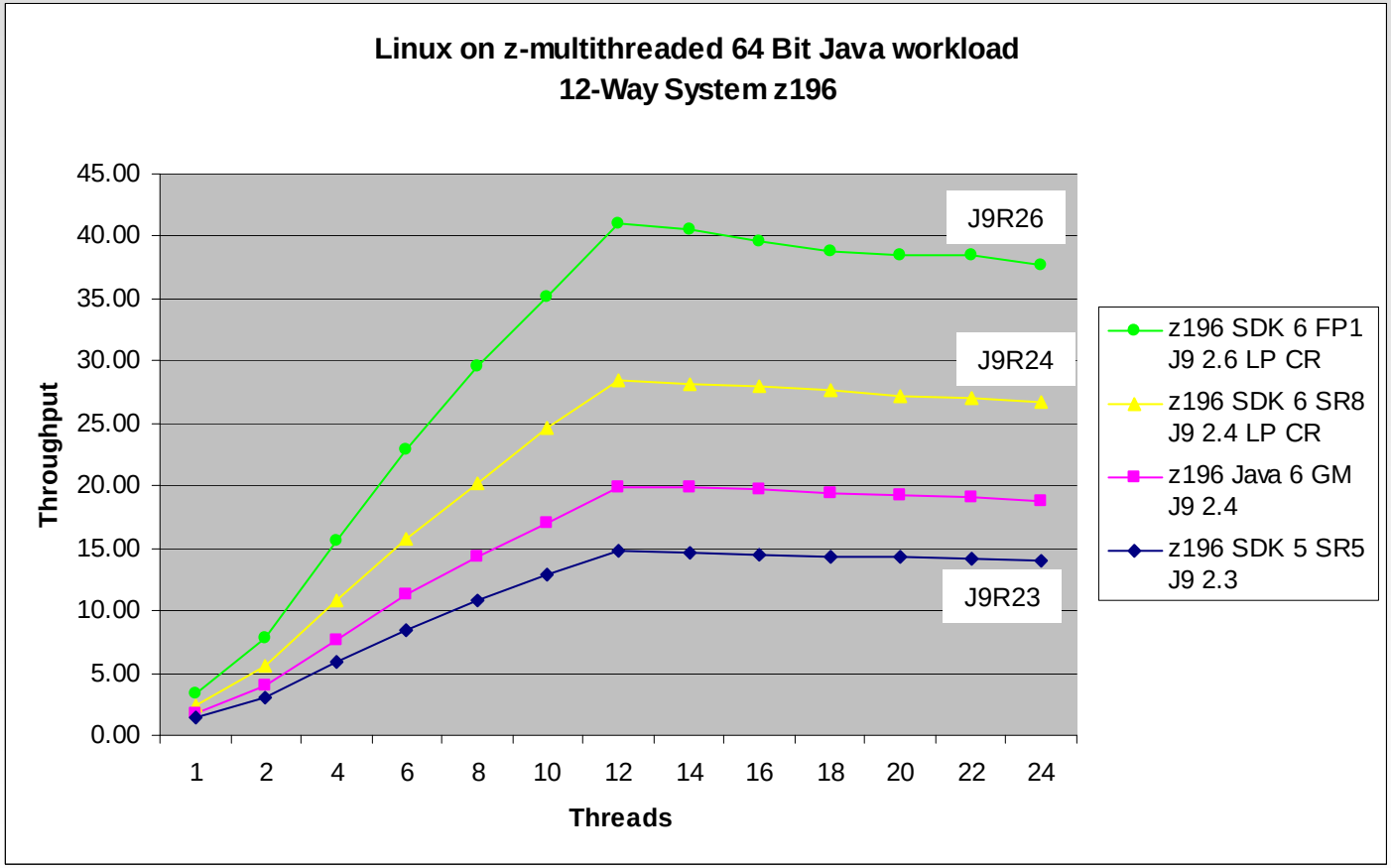
Performance on z/OS: WAS on z/OS



- This chart shows a history of improvements made from zSeries hardware (from z9 to z196) and software (from V6.1 to V8.0). The data is from measurements done using the DayTrader EJB workload.
  - The chart shows an aggregate performance improvement of almost 4x moving from WAS V6.1 on a z9 to WAS V8.0 on a z196.
  - The hardware component of this increase is about 2.25x (1.57 x 1.43)
  - The software component is about 1.72x (1.40 x .123)

(Controlled measurement environment, results may vary)

z/Linux Java6 R26 Performance: 64 Bit Multi-threaded Benchmark

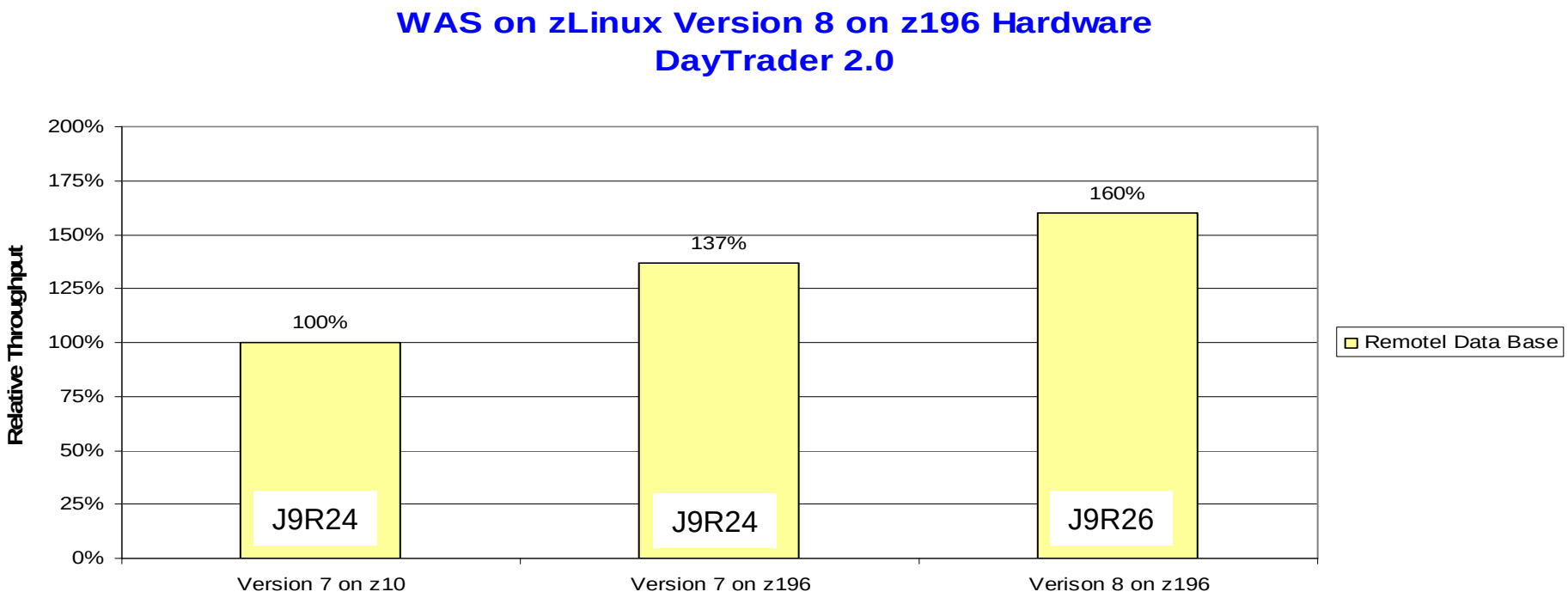


2.7x Aggregate Software improvement

- 42% Java6R26 vs Java6R24 improvement
- 42% Java6R24 SR9 vs Java6R24 GM
- 35% Java 6 GM versus Java 5 SR5

(Controlled measurement environment, results may vary)

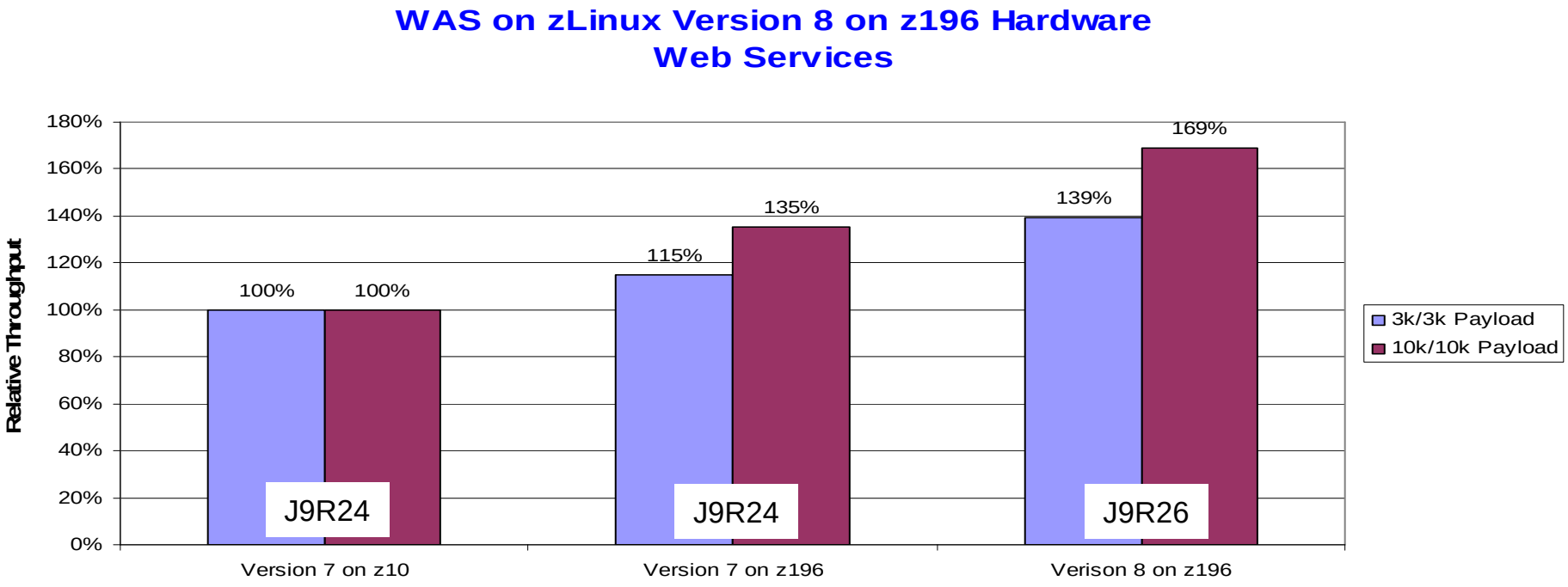
Performance – WAS8.0 on zLinux



- Upgrading from z10 to z196 improved throughput by 37% using our DayTrader 2.0 EJB benchmark.
- Additionally, upgrading to WAS V8.0 improved performance by another 17%. This increase is a result of improvements to the following areas:
  - JVM and JIT optimizations
  - OpenJPA code paths
- The combine hardware and software improvement is 60%.

(Controlled measurement environment, results may vary)

Performance – WAS8.0 on zLinux



- Upgrading from z10 to z196 improved throughput by as much as 35% using our SOABench webservices benchmark (15% for the 3k/3k payload and 35% for the 10k/10k payload).
- Additionally, upgrading to WAS V8.0 improved performance by another 21% for the 3k/3k payload and 25% for the 10k/10k payload. This increase is a result of improvements to the following areas:
  - JVM and JIT optimizations
  - JAXB fastpath optimizations
- The combine hardware and software improvement is 39% for the 3k/3k case and 69% in the 10k/10k case.

(Controlled measurement environment, results may vary)

IBM J9 Garbage Collector

- **IBM J9 VM garbage collector family**
  - Parallel global (mark, sweep, compact)
  - Generational collection
  - Partial concurrency at global level
  - Use of OS level features (Virtual Memory, large pages, etc)
  - Type accurate stacks, cooperative suspend
  - No pinned objects for JNI (less heap fragmentation)

  - Tunable garbage collection policies to best match application behaviour

Policy	Recommended usage	Notes
`optThroughput`	optimized for throughput	default in Java5 and Java6
`optAveragePause`	optimized to reduce pause times	
`gencon`	optimized for transactional workloads	default in Java601
`subPools`	optimized for large MP systems	deprecated in Java601
`balanced`	optimized for large heaps	added in Java601

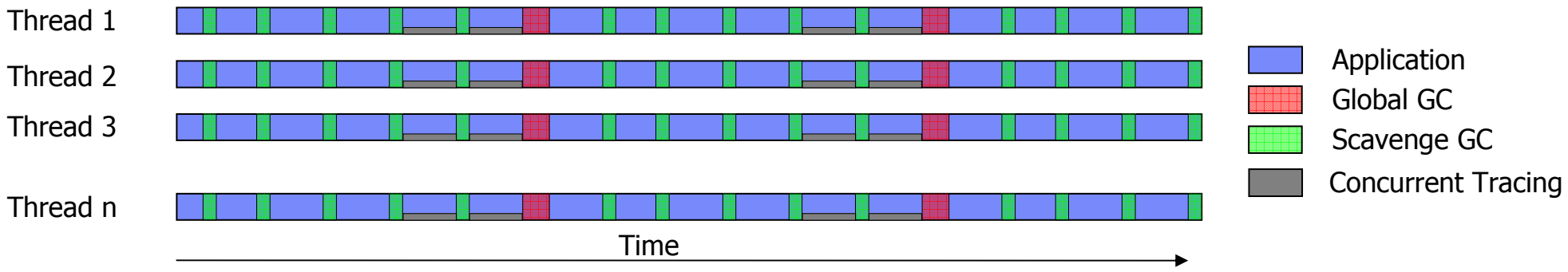
IBM J9 Garbage Collector: -Xgcpolicy:gencon

- **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 Java 6.0.1

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.



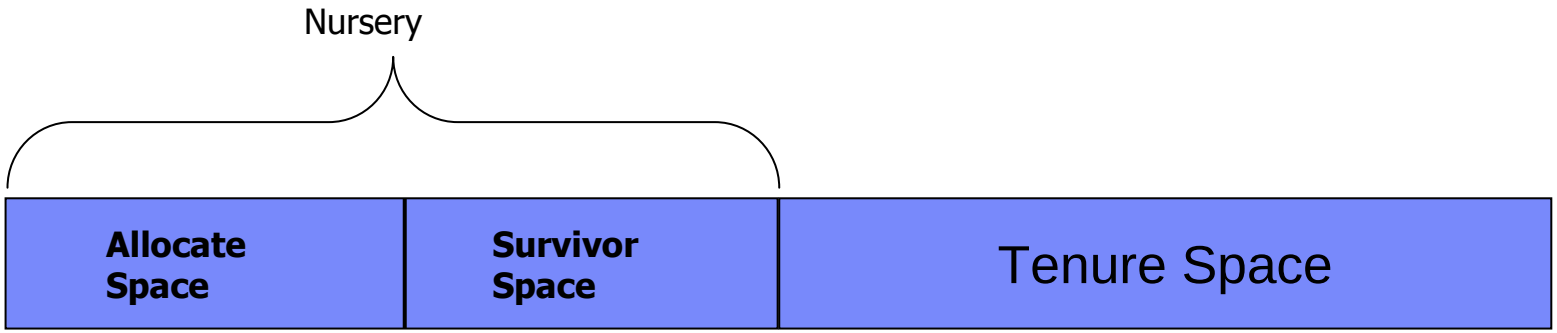
Picture is only illustrative and doesn't reflect any particular real-life application. The purpose is to show theoretical differences in pause times between GC policies.

IBM J9 Garbage Collector: A closer look inside 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

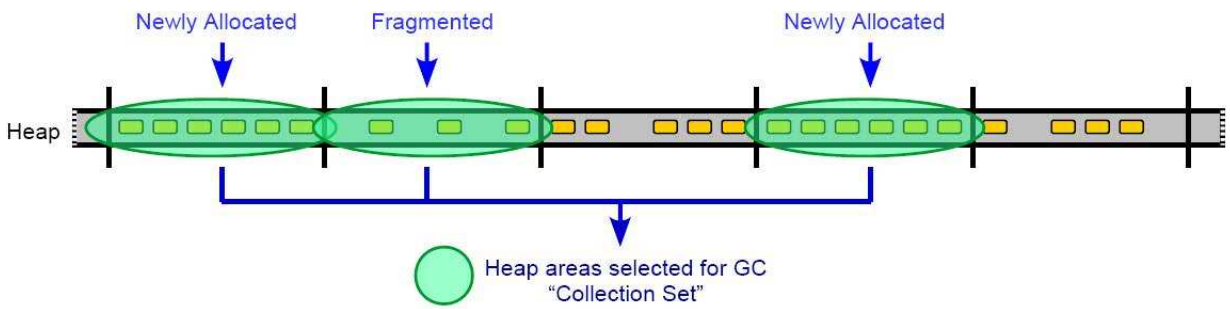
Next generation technology expands platform exploitation possibilities

- 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



Inter-language Communication: Signal Handling with -XCEEHDLR (31-bit z/OS only)

- New feature in Java 6.0.1
  - Looking for feedback
- Switches JVM from POSIX to LE Signal Handling for
  - SIGBUS
  - SIGFPE
  - SIGILL
  - SIGSEGV
  - SIGTRAP
- A condition triggered while executing a JNI component causes the JVM to convert the Language Environment condition into a Java ConditionException
  - Allows Java application to see/catch LE conditions
  - com.ibm.le.conditionhandling.ConditionException exception is thrown

Inter-language Communication - JNI Best Practices

See http://www.ibm.com/developerworks/java/library/j-jni/

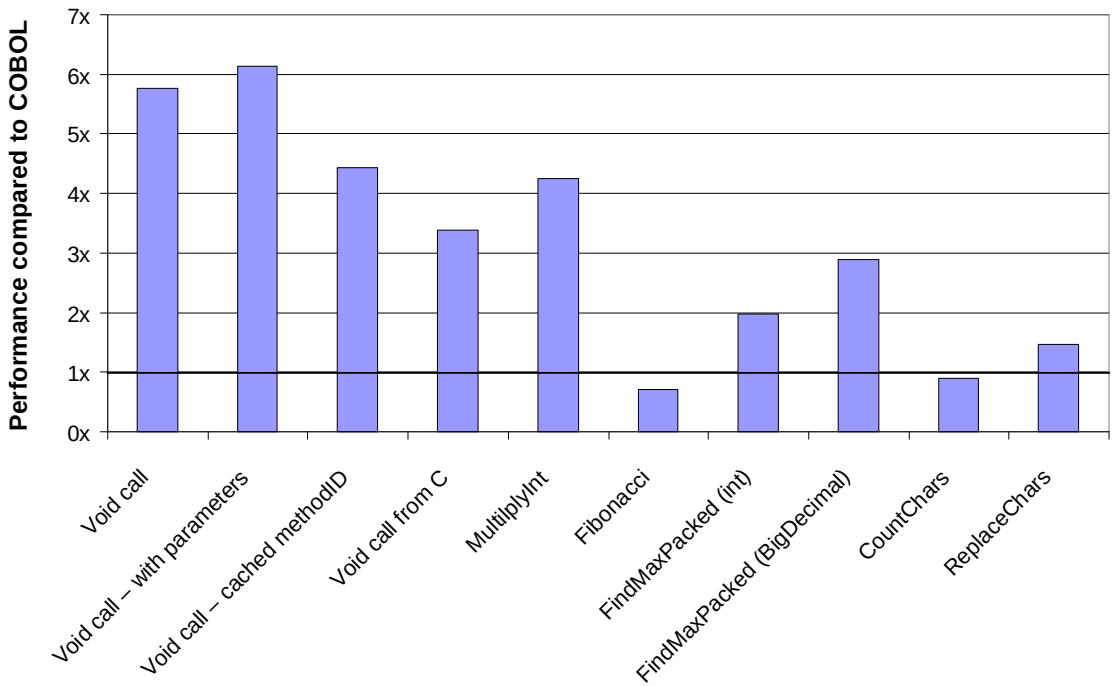
- Common performance pitfalls
  1. Not caching method IDs, field IDs, and classes
     - Avoid redundent calls to `FindClass(), GetFieldID(), GetMethodId(), and GetStaticMethodID()`
  2. Triggering array copies
     - Assume arrays are buffered, hence be precise about which elements you really need to avoid needless copying
  3. Reaching back instead of passing parameters
     - When possible, flatten object fields into parameters of call. Avoid using JNI services to get to object fields
  4. Choosing the wrong boundary between native and Java code
     - Assume Native ⇔ Java call overhead 10x slower than Native ⇔ Native or Java ⇔ Java
  5. Using many local references without informing the JVM

Inter-language Communication
Java/COBOL Inter-Operability Performance

- COBOL → Java compared to COBOL → COBOL
  - Java void() method shows 6x more overhead
  - Caching methodID reduces overhead to 4.4x
  - Max operation on a set of packed decimals:
    - 2x slowdown when Java transformed decimals into ints
    - 2.9x slowdown when Java transformed decimals into BigDecimals
  - Fibonacci (42 adds in a loop) shows Java performs 40% better than COBOL

- +96% of the program is eligible for zAAP offload

- Best practices:
  - Do as much work in Java as possible
  - Have as few COBOL ↔ Java transitions as possible



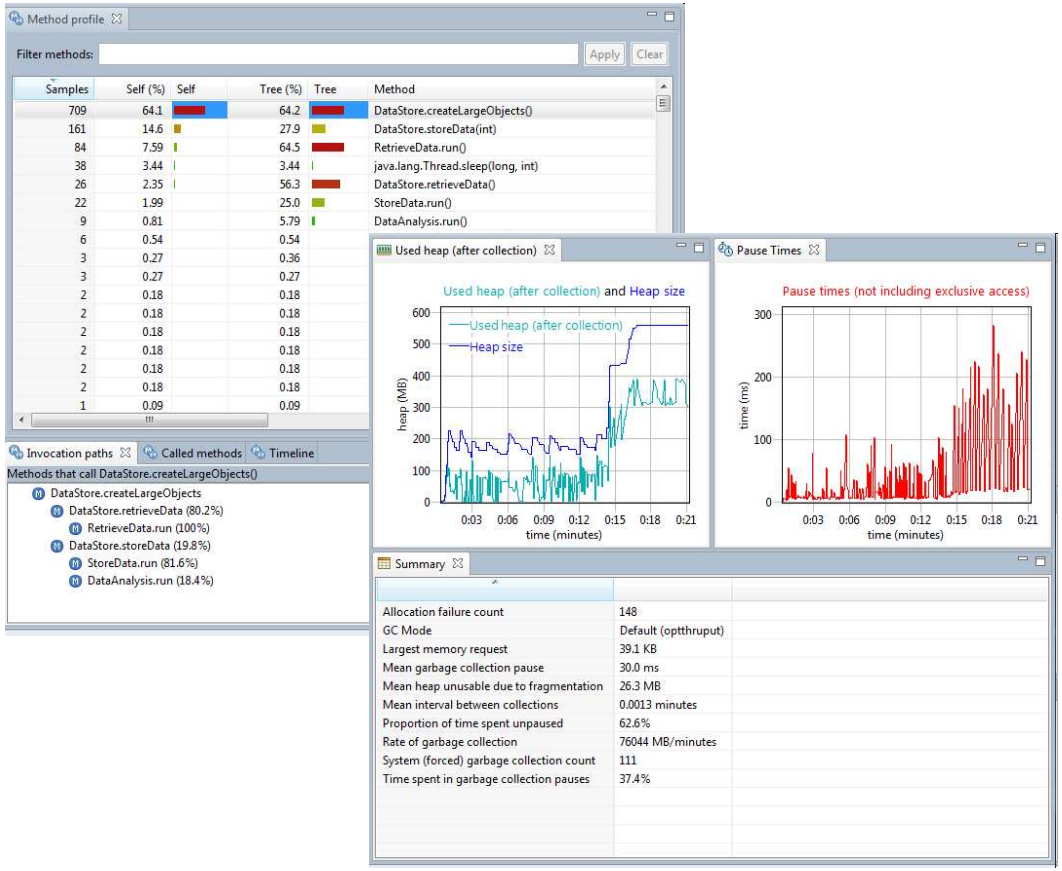
Appendix

A. IBM Java Consumability

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
- **Not a lot of experience using ISA under IMS**… however it should just work

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

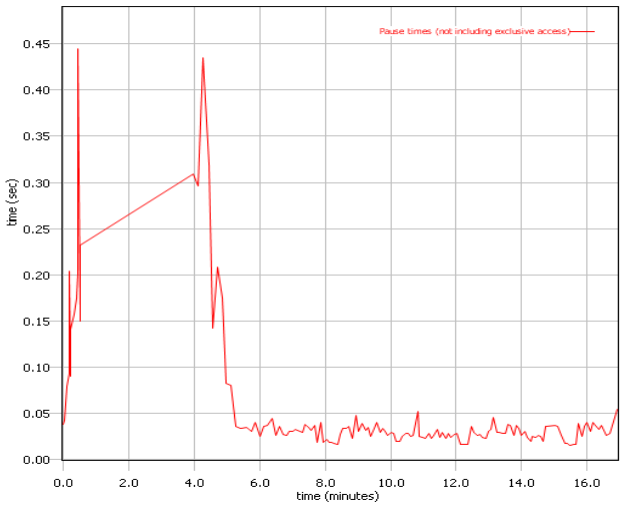
IBM Monitoring and Diagnostic Tools for Java - GCMV

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 (forced) GCs. 5 out of 145 collections (3.448%) 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 algorithms 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.

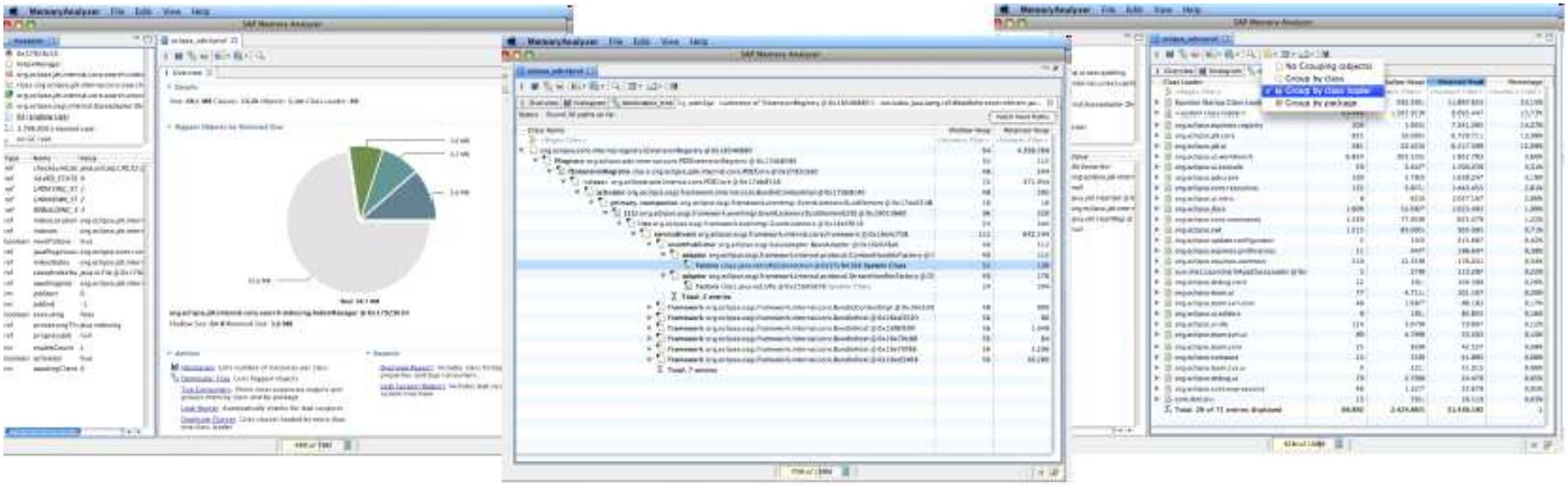
Summary

Allocation failure count	140
Concurrent collection count	0
Forced collection count	5
GC Mode	gencon
Global collections - Mean garbage collection pause (ms)	185
Global collections - Mean interval between collections (minutes)	0.13
Global collections - Number of collections	5
Global collections - Total amount tenured (MB)	93.1
Largest memory request (bytes)	127784
Minor collections - Mean garbage collection pause (ms)	48.2
Minor collections - Mean interval between collections (ms)	7193
Minor collections - Number of collections	140

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)

B. zAAPs

The System z Application Assist Processor (zAAP)

- Helps enable integration of Java-based applications into z/OS environment
- Can help simplify and reduce server infrastructure and improve operational efficiencies over distributed multi-tier solutions.
- Executes Java with the reliability, availability, and security you expect from the mainframe…with no changes to applications
- Reduces up front and maintenance costs for workloads with Java cycles e.g.: WebSphere, DB2
- Provides additional capacity at low cost of acquisition to process Java without affecting MSU rating or machine model designation…No additional IBM software costs

Objective: Enable integration of new Java based Web applications with core z/OS backend database environment for high performance, reliability, availability, security, and lower total cost of ownership

zAAP: An Example…

WebSphere Java-based Application that is transactional in nature and requires 1000 MIPS on System z.

In this example, with zAAP, we can reduce the standard CP capacity requirement for the Application to 500 MIPS or a 50% reduction. ** For illustrative purposes only*

C. IBM Testarossa JIT

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 – 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:

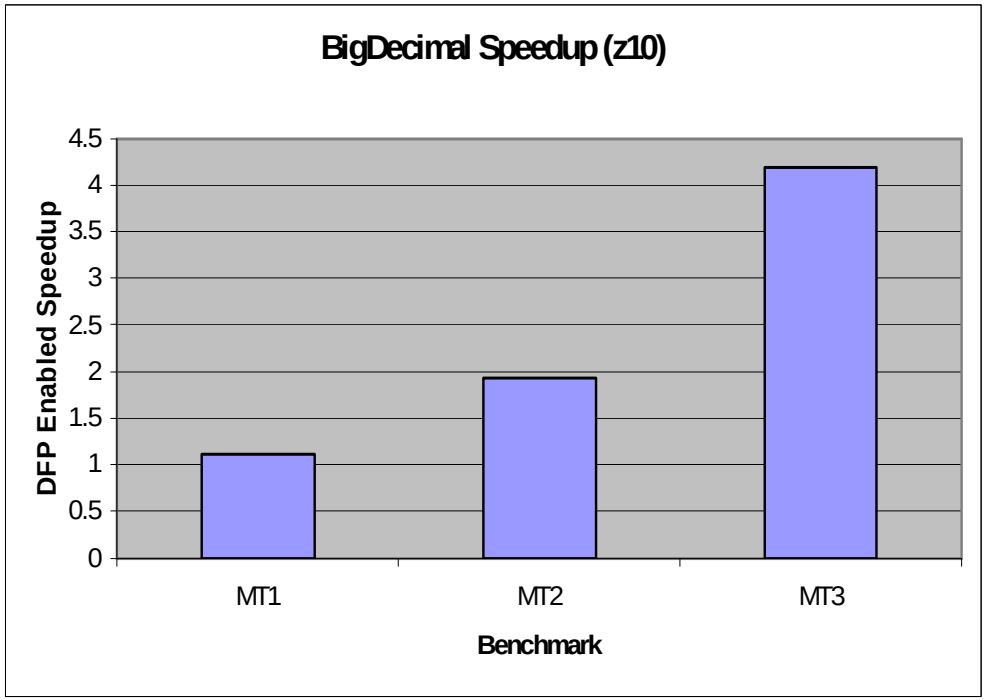
```
while (i < end) {
   value = table[arrB[i+offsetB]];
   if (value == termChar) break;
   arrA[i+offsetA] = value;
   ++i;
}
```

```
LB:  DS 0H
     TRxx     // xx depends on arrA/B types
     BRC LB   // re-drive long xlate
```

** M. Kawahito, *et al.*, A new idiom recognition framework for exploiting hardware-assist instructions, ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, 2006

IBM Testarossa JIT -- Decimal Floating Point

- 31/64/128-bit DFP in hardware
  - specialized accelerator technology
- Java 6 exploits DFP via Java BigDecimal**
- The class observes uses of BigDecimal and makes a decision about exploiting DFP
- MathContext64 must be used to ensure optimal performance



10% to 4x improvement in BigDecimal benchmark when exploiting DFP

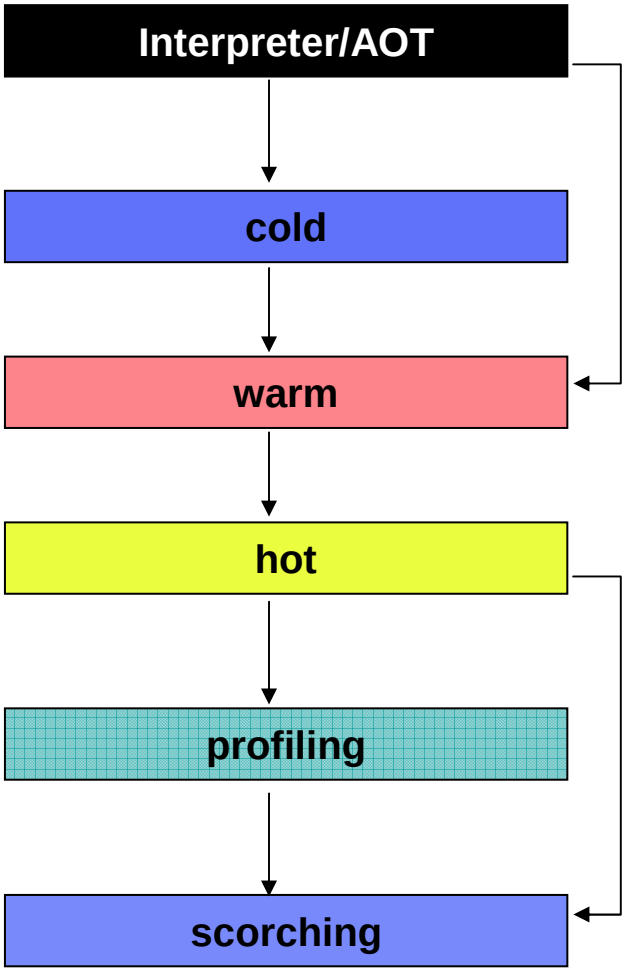
Controlled measurement environment, results may vary.

z10™ – z/OS V1.9 - Java 6 SR1

** *Mitran, Sham, Stepanian,* ***Decimal floating-point in Java 6: Best Practices****, Partner World, 2008 –*
https://www-304.ibm.com/jct09002c/partnerworld/wps/servlet/ContentHandler/whitepaper/power/java6_sdk/best_practice

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



Shared Classes & Ahead-Of-Time (AOT) Compilation

- Shared Classes
  - Store classes into a cache that can be shared by multiple JVMs
  - Read-only portions of the class
  - Memory footprint reduction
  - Startup time improvements (class initialization)
  - Cache memory page protection (read-only caches)
  - Class compression (64-bit class compression)
  - Persistent cache (between reboots)

- AOT Compilation
  - Compiled code generated “ahead-of-time” to be used by a subsequent execution
    - Performance of AOT code is poor
      - Cannot be specialized due multi-instance use and dynamic class loading
      - Dynamic class loading imposes overhead of assumption management
    - Rely on recompilation to make code that matters better
  - Persisted into the same shared cache
  - Startup time improvements
  - CPU utilization reduction

D. IBM J9 Garbage Collector

IBM J9 Garbage Collector

- IBM J9 VM garbage collector family
  - Parallel global (mark, sweep, compact)
  - Generational collection
  - Partial concurrency at global level
  - Use of OS level features (Virtual Memory, large pages, etc)
  - Type accurate stacks, cooperative suspend
  - No pinned objects for JNI (less heap fragmentation)

  - Tunable garbage collection policies to best match application behaviour

Policy	Recommended usage	Notes
`optThroughput`	optimized for throughput	default in Java5 and Java6
`optAveragePause`	optimized to reduce pause times	
`gencon`	optimized for transactional workloads	default in Java601
`subPools`	optimized for large MP systems	deprecated in Java601
`balanced`	optimized for large heaps	added in Java601

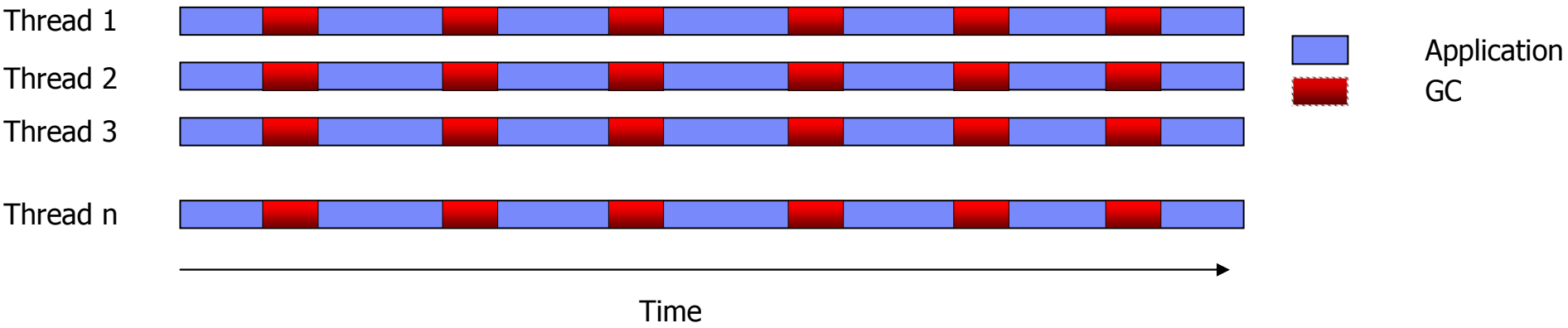
IBM J9 Garbage Collector

- Selectable policies under which it will recycle objects
- 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: -Xgcpolicy:optthruput

The default policy in Java5 and Java6.

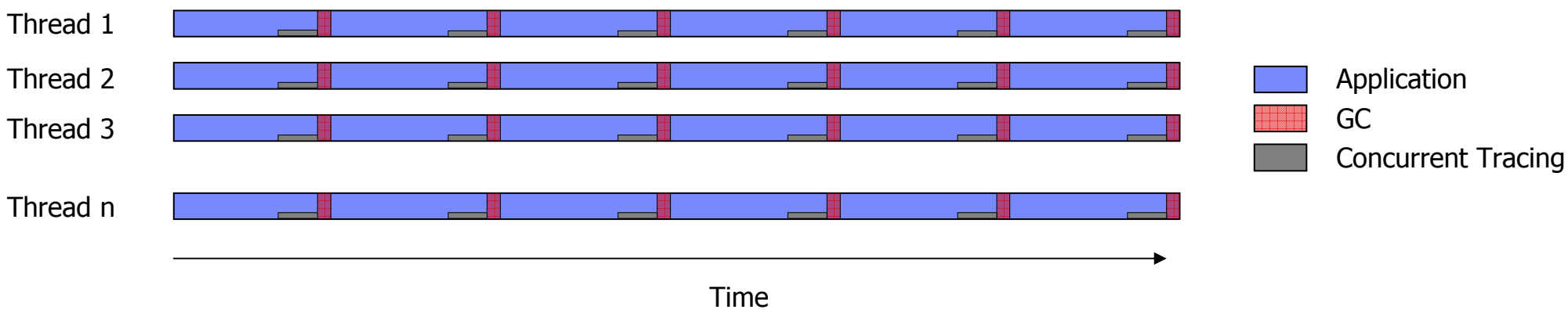
Used for applications where raw throughput is more important than short GC pauses. The application is stopped each time that garbage is collected.



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.

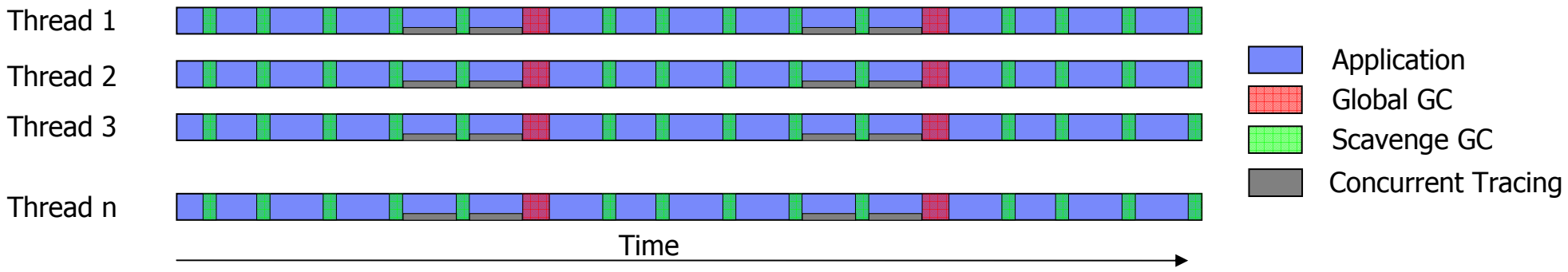
IBM J9 Garbage Collector: -Xgcpolicy:gencon

- 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 Java 6.0.1

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.



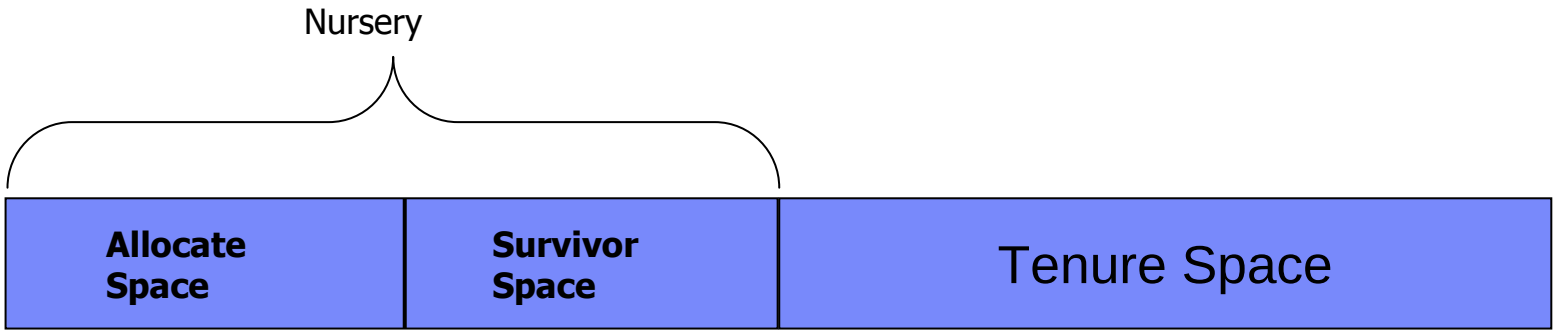
Picture is only illustrative and doesn't reflect any particular real-life application. The purpose is to show theoretical differences in pause times between GC policies.

IBM J9 Garbage Collector: A closer look inside 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

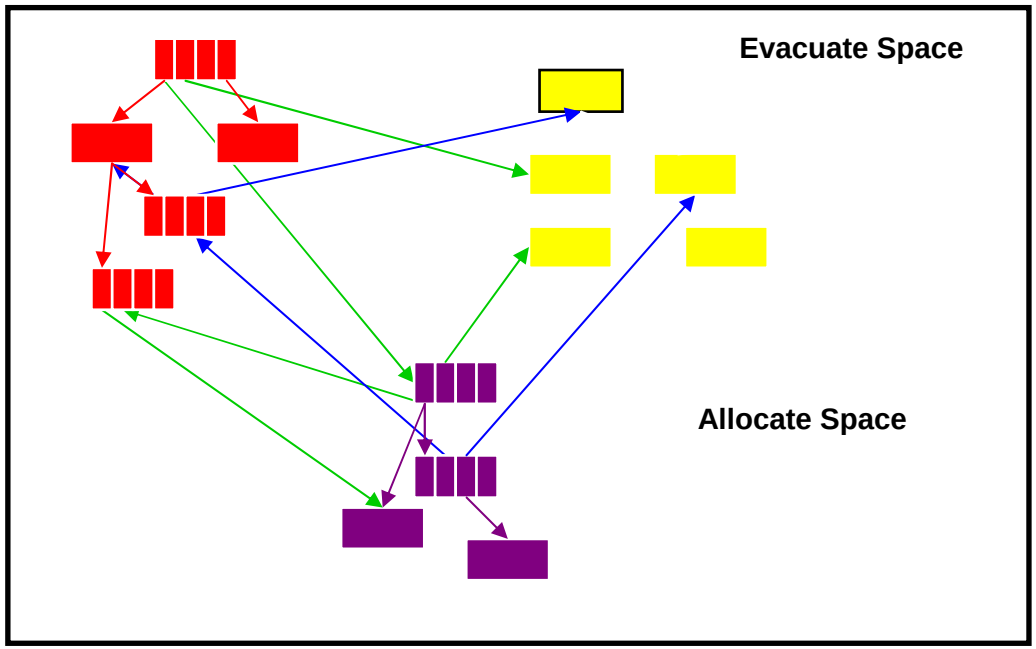
- If an object survive X number of scavenges it gets promoted to the 'tenure' space

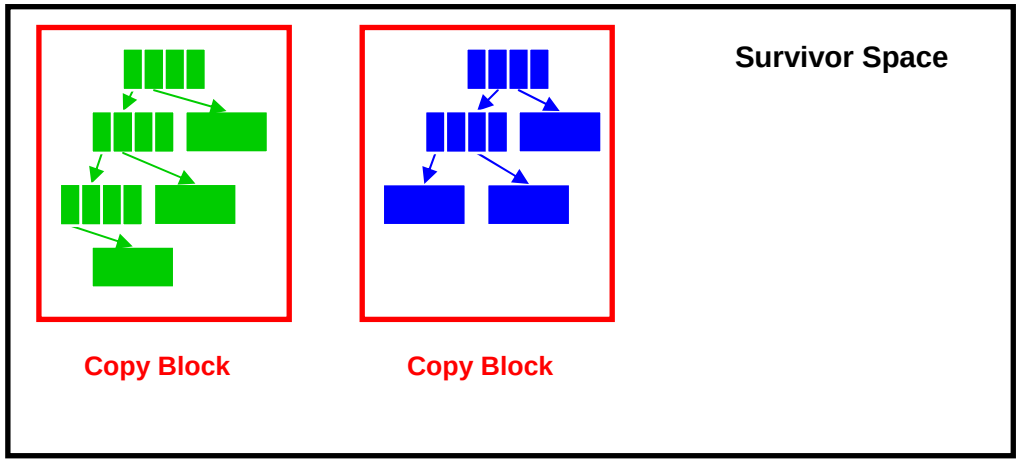


The division between allocate and survivor space is dynamic.

It will be adjusted depending on the survival rate.

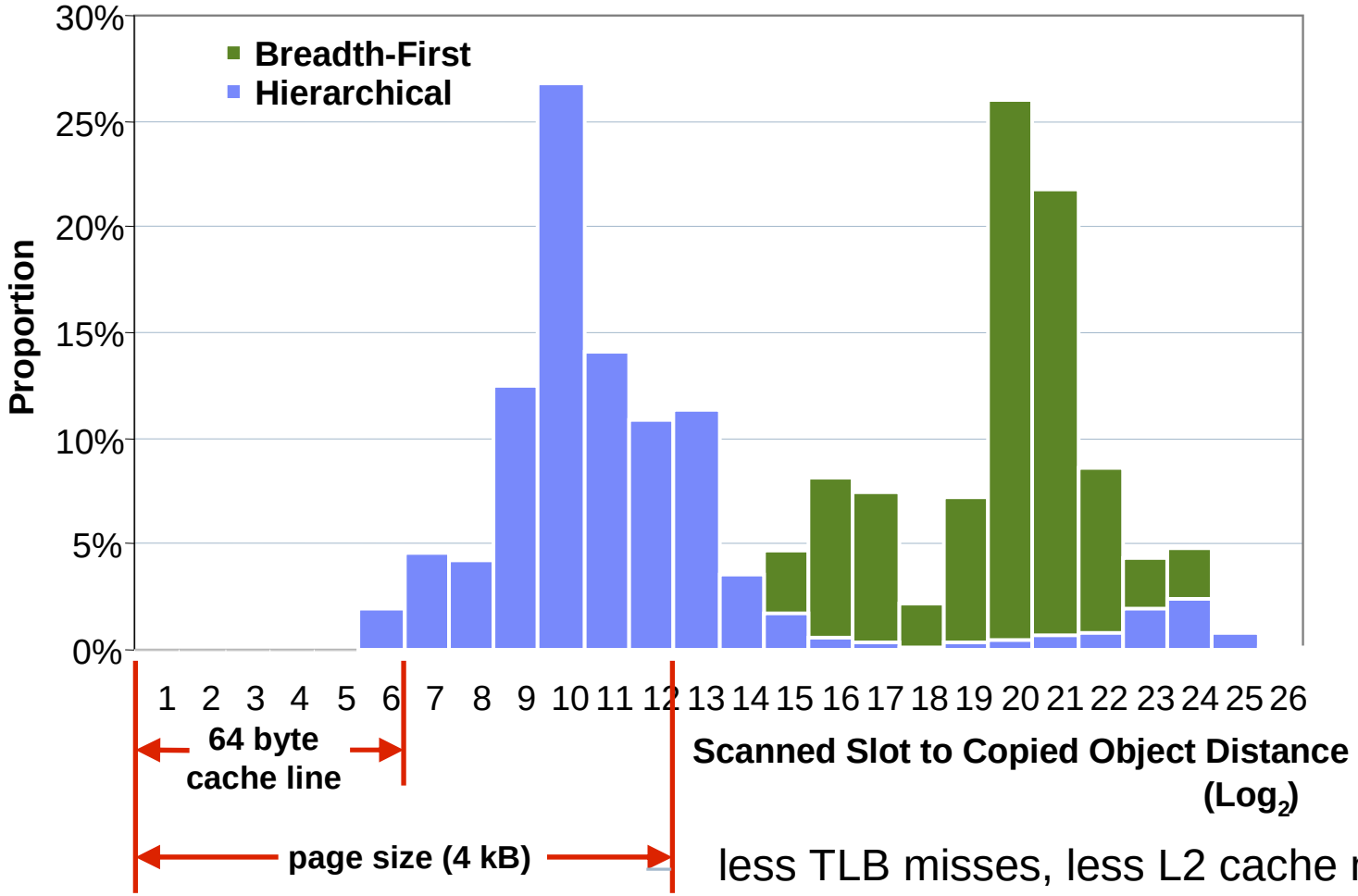
IBM J9 Garbage Collector: Hierarchical Scan





- Initially objects are allocated close to relatives.
- Connectivity changes and objects die; locality worsens.
- At copy collection GC, live objects are copied from the evacuate space to the survivor space.
- GC Threads copy objects into thread local blocks
- Hierarchical GC copies related objects (child, parents, siblings, cousins) into same copy block
- … improving locality between related objects

IBM J9 Garbage Collector: Hierarchical Scan



less TLB misses, less L2 cache misses

Controlled measurement environment, results may vary.

IBM J9 Garbage Collector: 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

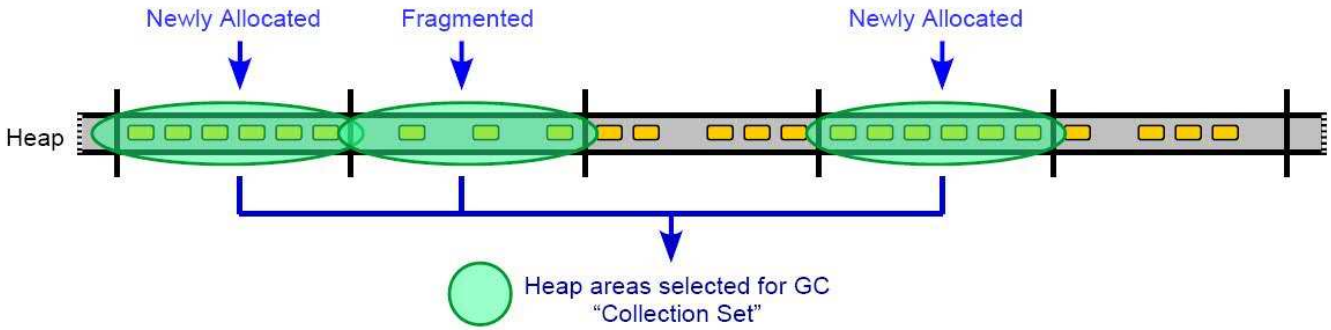
Next generation technology expands platform exploitation possibilities

- 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

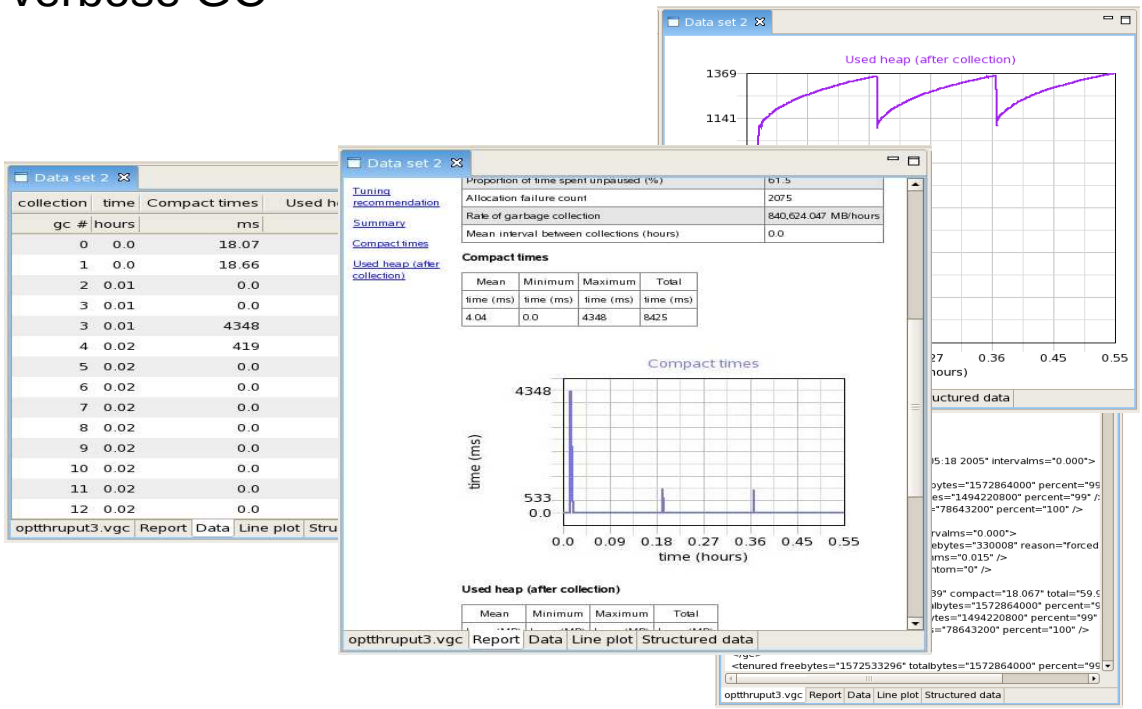
- Large (>4GB) heaps
- Frequent global garbage collections
- Excessive time spent in global compaction
- Relatively frequent allocation of large (>1MB) arrays

Input welcome: Help set directions by telling us your needs



IBM J9 Garbage Collector: Tuning

- GC Tuning documentation
  - http://www.ibm.com/developerworks/views/java/libraryview.jsp?search_by=java+technology+ibm+style:
  - http://www-01.ibm.com/support/docview.wss?uid=swg27013824&aid=1
  - http://proceedings.share.org/client_files/SHARE_in_San_Jose/S1448KI161816.pdf
  - http://www.redbooks.ibm.com/redpapers/pdfs/redp3950.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



E. 64-bit Performance

64-bit Performance with Java6

- 31-bit address space
  - Theoretically: 2GB of addressable virtual memory
  - Realistically: less than 2GB (~800M to 1.3GB in typical WAS)
    - Java heap needs to be contiguous
    - Native application code, J9/TR runtimes and data, OS modules
  - Customers reaching limits (OOM Exceptions)

- Move to 64-bit <u>pointers</u> is not free
  - Objects on average ~60% bigger
    - ~ 60% increase in Java heap footprint (smaller heap occupancy ratio)
  - Increased Cache/TLB pressure
    - Addressability increased, hardware remained constant (throughput effects)

- Concerted investment in Java6 JRE
  - Large Pages Technology
  - Compressed References Technology

Compressed References

- Option to enable compression in 64-bit Java 6 SR4, WAS 7 (SPack3)
  - use –Xcompressedrefs
- Java objects are 8-byte aligned
  - Low 3 bits of object address = 000
- Address range restriction: Java heap allocated in 2^{31} – 2^{35} range (2GB – 32GB virtual)
  - High 29 bits of object address = 000 … 000
  - 32 out 64 bits are 0!
- Main idea is to store 32-bit shifted offset in objects
  - Shift values of 0 through 3 are used
- Maximum allowable heap is 30GB

Compressed References: Technical Details

- **32-bit Object (24 bytes – 100%)**

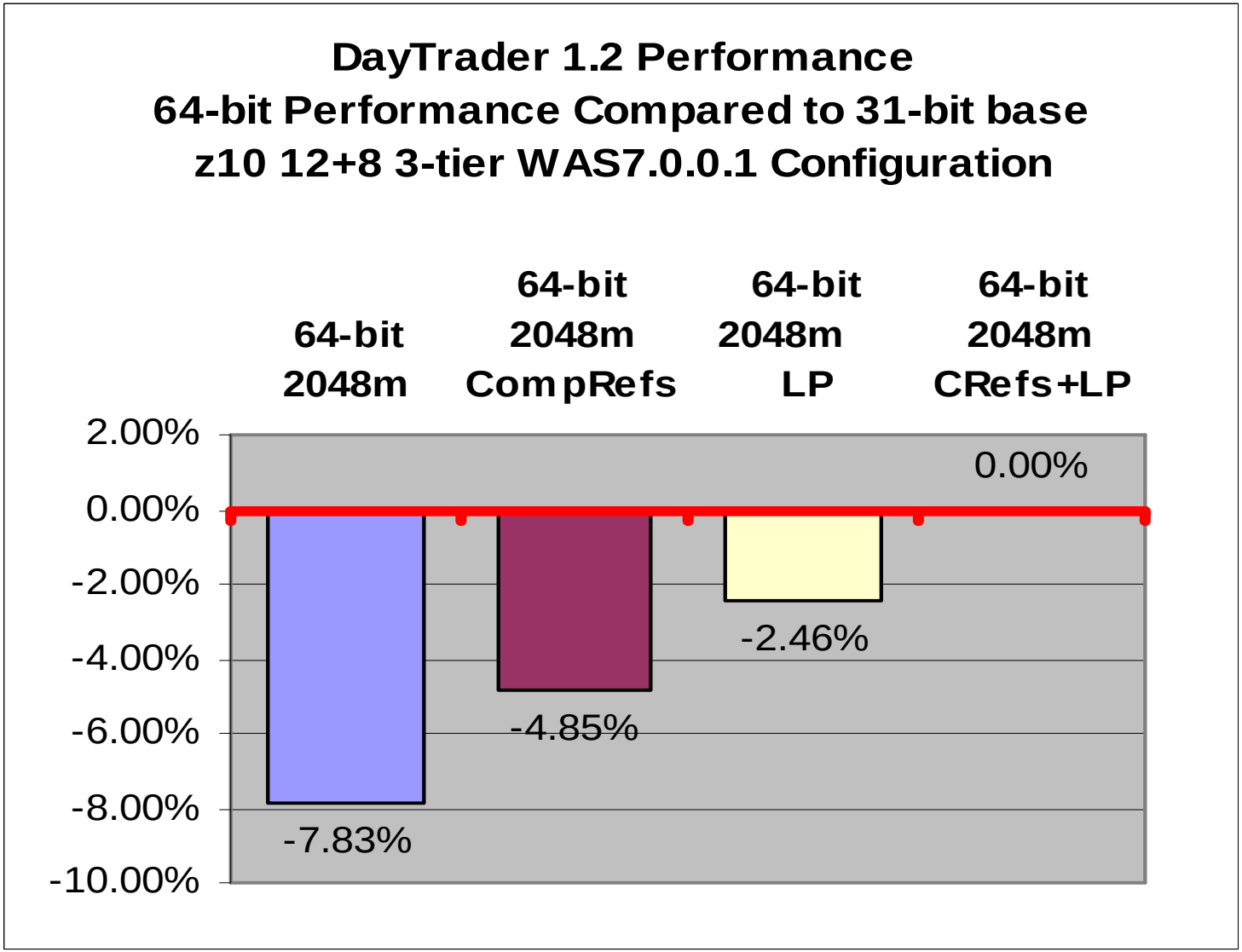
- **64-bit Object (48 bytes – 50%)**

- **64-bit Compressed References (24 bytes – 100%)**

Use 32-bit values (offsets) to represent object fields

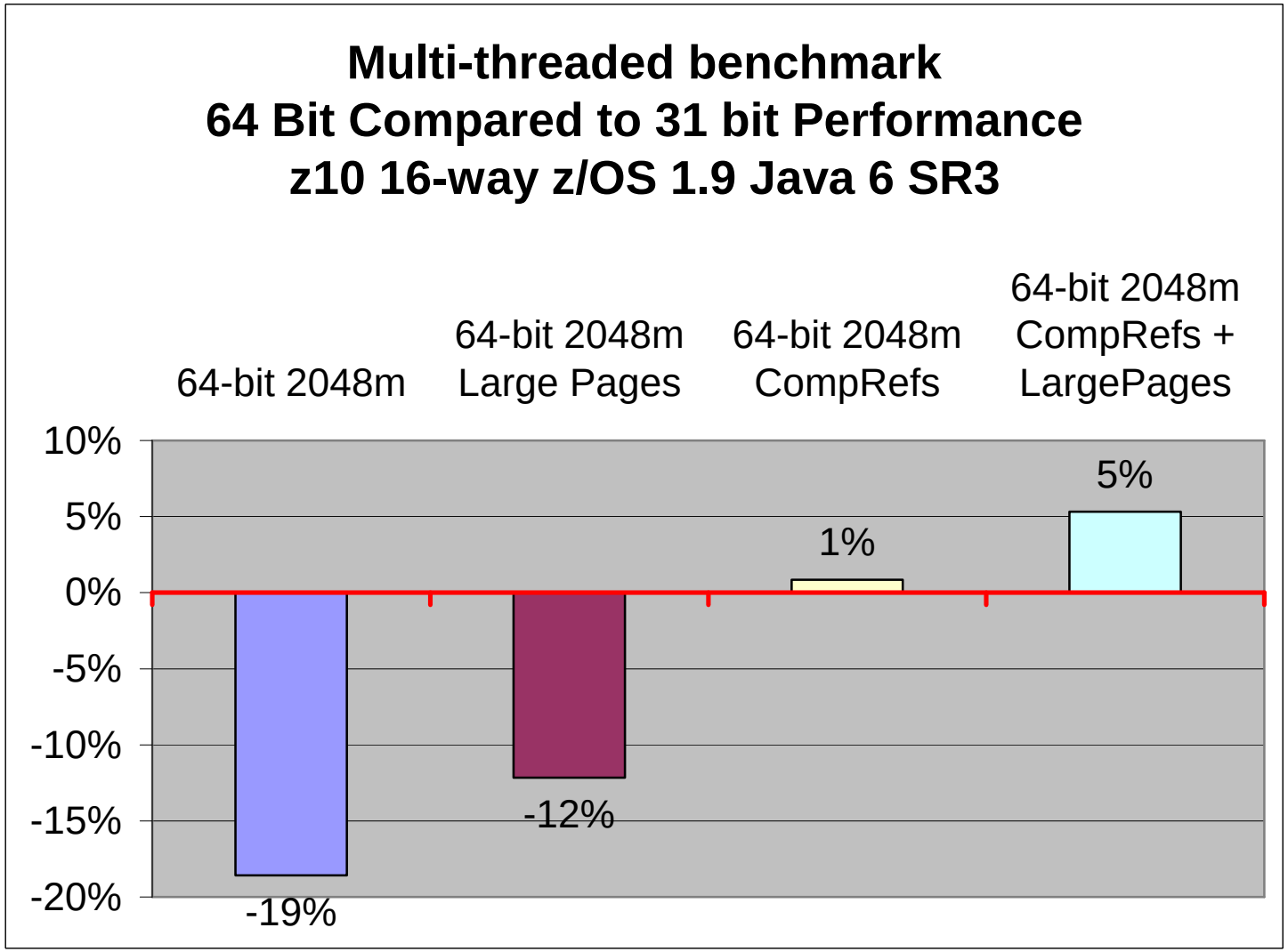
With scaling, between 4 GB and 32 GB can be addressed

WebSphere 64- vs. 31-bit Performance



Controlled measurement environment, results may vary.

Java6 64- vs. 31-bit Performance

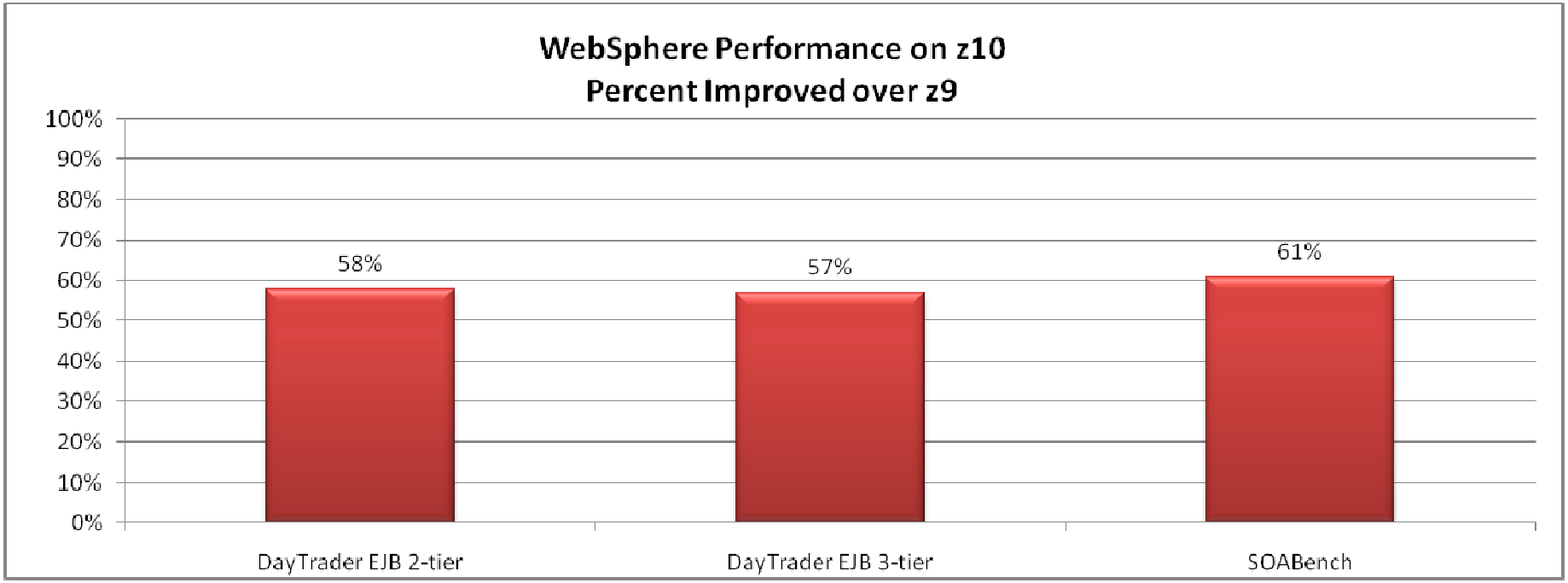


(Controlled measurement environment, results may vary)

F. IBM System z Hardware and Java

System z10™ vs. z9™ Performance

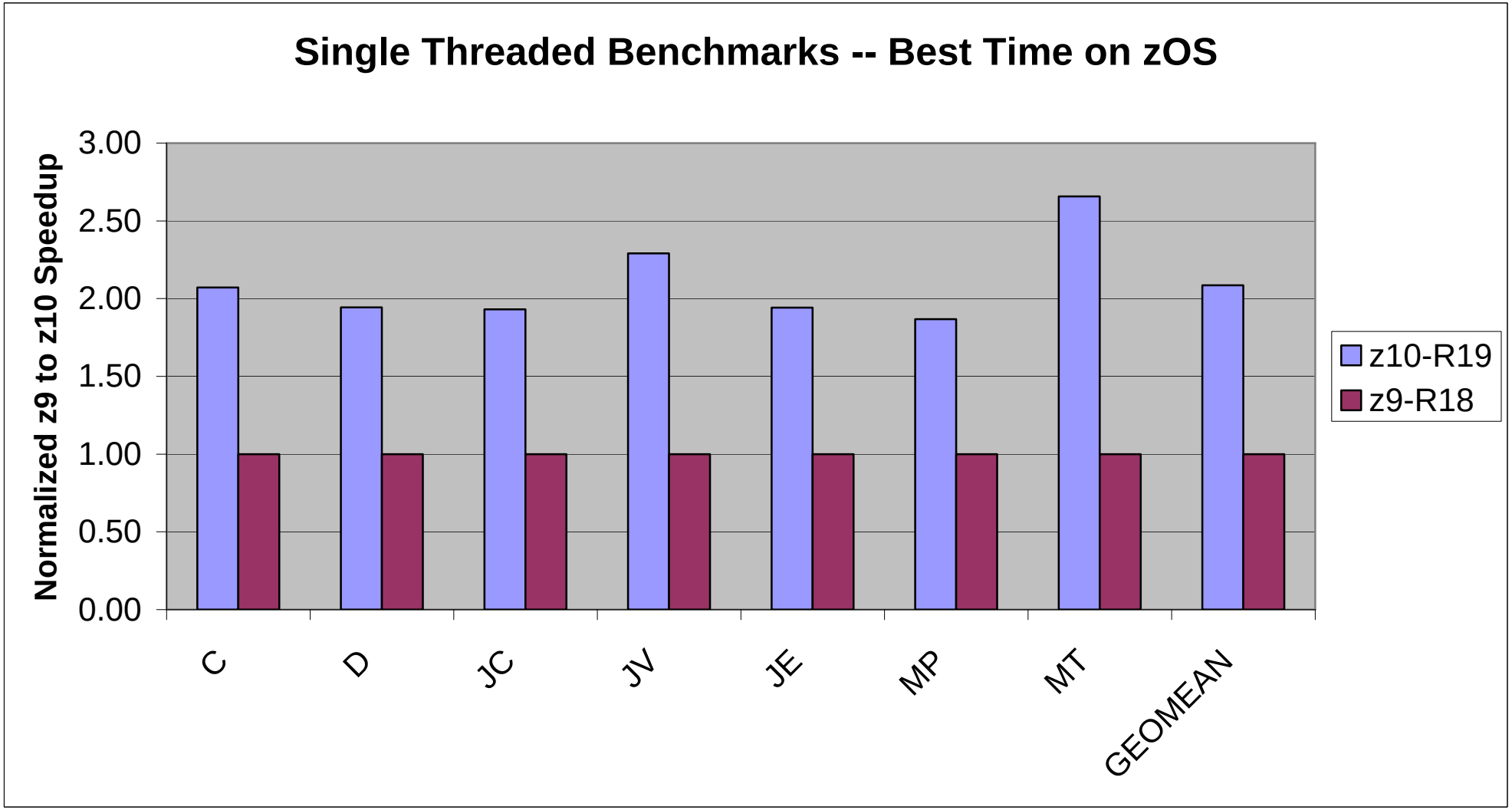
WebSphere V7 Performance on z10™ Hardware



- **WebSphere V7 Workload Performance on z10™**
  - Clear performance improvement by moving to z10™ hardware from z9™ for JEE applications
    - zOS 2-tier configuration (WAS/DB2 co-located) improved 58%
      - zOS 2-tier configuration is, by far the more prevalent configuration for zOS customers.
    - zOS 3-tier configuration (WAS and DB2 on separate zOS images) improved by 57%
  - SOABench (Web Services) improved by 61%

System Configuration
Workload: DayTrader EJB, SOABench (WS)
SUT: IBM z9™ Processor (model 2094 – 715) 4 x 1.7 GHz, 10 GB Real
SUT: IBM z10™ Processor (model 2097 – 720) 4 x 4.4 GHz, 32 GB Real
Controlled measurement environment, results may vary.

System z10 to z9 Performance – Compute Intensive



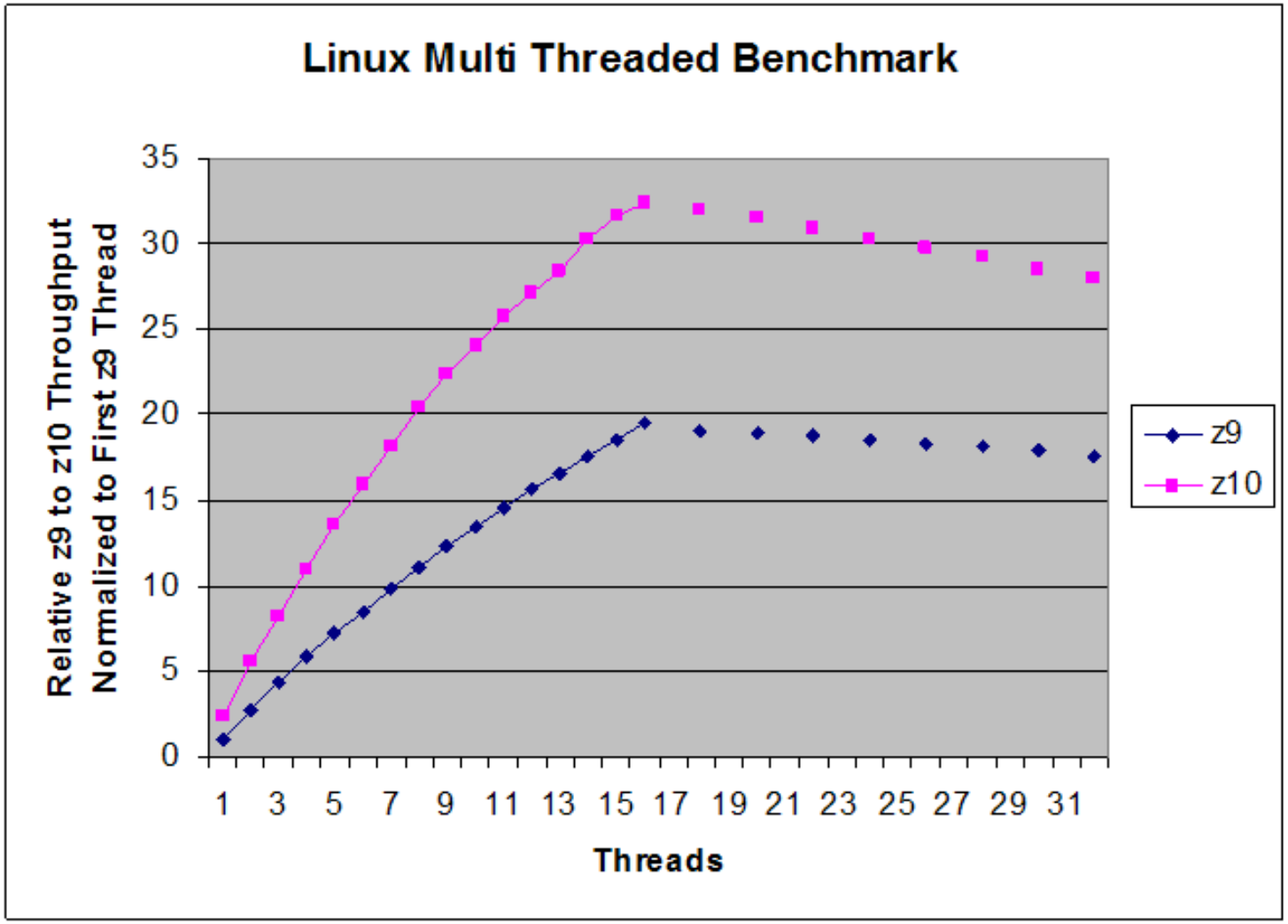
z10 sees 2x improvement over z9

Bigger is better

Java6 31bit - z9 – zOS V1.8 vs z10 – zOS V1.9

 Controlled measurement environment, results may vary.

System z10 to z9 Performance – Multi-Threaded Benchmark



z10 see ~60% improvement over z9

Bigger is better

Java6 31bit- z9 – SLES7 vs z10 – SLES9

Controlled measurement environment, results may vary.

G. Links and Documentation

Summary of Links

- Documentation
  - http://www.ibm.com/developerworks/java/jdk/docs.html
- zOS SDK
  - http://www.ibm.com/servers/eserver/zseries/software/java
- System z Linux SDK
  - http://www.ibm.com/developerworks/java/jdk/linux/download.html
- GC Tuning documentation
  - http://www.ibm.com/developerworks/views/java/libraryview.jsp?search_by=java+technology+ibm+style:
- IBM Support Assistant
  - http://www.ibm.com/software/support/isa/

Read More: Java Technology, IBM Style

http://www.ibm.com/developerworks/views/java/libraryview.jsp?search_by=java+technology+ibm+style:

Java technology, IBM Style: Monitoring and problem determination
The **IBM** implementation of **Java technology**, Version 5.0 contains a number of useful tools that help you diagnose and solve **Java** development problems. In this final installment in our **Java technology**, **IBM** style series, you learn about the information available from the **IBM** trace and dump engines. You also get an introduction to the Diagnostic Toolkit and Framework for **Java** (DTFJ) API, with which you can write code that queries and analyzes diagnostic data.

Java technology, IBM style: Class sharing
The new Shared Classes feature in the **IBM** implementation of version 5.0 of the **Java** platform offers a completely transparent and dynamic means of sharing all loaded classes that places no restrictions on the JVMs that are sharing the class data. This feature offers a straightforward and flexible solution for reducing virtual memory footprint and improving startup time, and there are few applications that will not benefit from it. This article explores how the feature works, how to use it, and when to use it, along with some of the features it provides.

Java technology, IBM style: Garbage collection policies, Part 2
The previous installment in this series introduced the different garbage collection (GC) policies available in the **IBM** implementation of version 5.0 of the **Java** runtime and discussed their general characteristics. In this article, series contributor Mattias Persson is joined by Holly Cummins to present a quantitative approach to choosing a policy with some examples. They describe what you should consider in making a choice, how to get guidance on a choice from the verbose GC logs, and present two case studies.

Java technology, IBM style: Garbage collection policies, Part 1
One of the great benefits of the **Java** platform is that it takes care of much of the work of garbage collection for you, but there are occasions when you still want to tweak the way garbage collection takes place. With the latest **Java technology** implementation from **IBM**, you can choose among several garbage collection policies to help you get the most out of your application. In this second article in the series **Java technology**, **IBM** style, **Java** developer Mattias Persson explores the available options and details the situations in which each might be appropriate.

Java technology, IBM style: Introduction to the IBM Developer Kit
With the advent of **Java** 5.0 comes advances from **IBM** in its **Java technology** implementation. This article opens a six-part series on changes in the latest version of the **IBM** Developer Kit. Author Chris Bailey begins with a brief description of the improvements to the **Java** 5.0 platform and moves on to a whistle-stop tour of the changes in SDKs from **IBM**.

SDK V6 Reference Materials

- Prerequisites
  - http://www-03.ibm.com/servers/eserver/zseries/software/java/j6prereq64.html
- Download SDKs
  - http://www.ibm.com/java (z/OS)
  - http://www.ibm.com/developerworks/java/jdk/linux/download.html (zLinux)
- SDK V6 APIs
  - http://java.sun.com/javase/6/docs/api/
- Deprecated APIs
  - http://java.sun.com/javase/6/docs/api/deprecated-list.html
- Incompatibilities, visit Sun site:
  - http://java.sun.com/javase/6/webnotes/compatibility.html#incompatibilities
- Restrictions and Other Considerations
  - http://www-03.ibm.com/servers/eserver/zseries/software/java/j6restrict31.html
- SDK V6 key features
  - http://java.sun.com/javase/6/features.jsp
- **IBM SDK V6 Guide**
  - http://www-03.ibm.com/servers/eserver/zseries/software/java/pdf/java6/sdkguide.zos.pdf
- **IBM SDK V6.0.1**
  - http://public.dhe.ibm.com/common/ssi/ecm/en/zsl03118usen/ZSL03118USEN.PDF

Reference Summary

- Garbage Collection in the IBM SDK
  - http://www-128.ibm.com/developerworks/java/library/j-ibmjava2/index.html
- IBM SDK Diagnostic Guide
  - http://www.ibm.com/developerworks/java/jdk/diagnosis/
- z/OS 64-bit C/C++ and Java Programming Environment
  - http://www.redbooks.ibm.com/redpapers/pdfs/redp9110.pdf
- z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode
  - http://publibz.boulder.ibm.com/epubs/pdf/ceeam110.pdf
- z/OS SDK 5 and Runtime Environment User Guide
  - http://www-03.ibm.com/servers/eserver/zseries/software/java/pdf/sdkguide.zos.pdf
- Tuning and Diagnostic
  - http://www.ibm.com/developerworks/java/jdk/tools/index.html
- zOS R13 Announcement
  - http://www-01.ibm.com/common/ssi/ShowDoc.jsp?docURL=/common/ssi/rep_ca/2/897/ENUS211-252/index.html&breadCrum=DET001PT022&url=buttonpressed=DET002PT005&specific_index=DET001PEF502&DET015PGL002=DET001PEF011&submit.x=7&submit.y=8&lang=en_US
- Decimal floating-point in Java 6: Best Practices, Partner World
  - https://www-304.ibm.com/jct09002c/partnerworld/wps/servlet/ContentHandler/whitepaper/power/java6_sdk/best_practice
- Match 31-bit WebSphere Application Server performance with new features in 64-bit Java on System z
  - https://www-304.ibm.com/partnerworld/wps/servlet/ContentHandler/whitepaper/systemz/java_websphere/performance

Consumability Tools - Links

- Email: javatool@uk.ibm.com
- Tools Landing Page

http://www.ibm.com/developerworks/java/jdk/tools/

- IBM Support Assistant:

http://www.ibm.com/software/support/isa

https://wst.austin.ibm.com/isablue/2136.html

- ECuRep/Archive Explorer:

http://www.ibm.com/de/support/ecurep/

- Add the JDK documentation
  - Select the “Updater” tab
  - Add the “IBM Developer Kit for Java 5.0” (or 6.0) set

Important references

- z/OS Java web site
  - http://www.ibm.com/systems/z/os/zos/tools/java/
- IBM SDK Java Technology Edition Version 6 Supplement
  - http://public.dhe.ibm.com/common/ssi/ecm/en/zsl03118usen/ZSL03118USEN.PDF
- JZOS Batch Launcher and Toolkit Installation and User's Guide (SA23-2245-03)
  - http://publibfi.boulder.ibm.com/epubs/pdf/ajvc0103.pdf

New and existing supported Java products – z/OS

- IBM 31-bit SDK for z/OS, Java Technology Edition, Version 6.0.1
  - Web available on March 15, 2011 at Java SE 6 level
  - Product 5655-R31, supported on z/OS V1.10 and above
- IBM 64-bit SDK for z/OS, Java Technology Edition, Version 6.0.1
  - Web available on March 15, 2011 at Java SE 6 level
  - Product 5655-R32, supported on z/OS V1.10 and above
- Earlier Deliveries
  - IBM SDK for z/OS, Java 2 Technology Edition, Version 1.4
    - EOS September, 2011
  - IBM 31-bit SDK for z/OS, Java 2 Technology Edition, Version 5.0
    - Web available on November 30, 2005
    - Product 5655-N98
  - IBM 64-bit SDK for z/OS, Java 2 Technology Edition, Version 5.0
    - Web available on November 30, 2005
    - Product 5655-N99
  - IBM 31-bit SDK for z/OS, Java Technology Edition, Version 6.0.0
    - Web available on December 14, 2007 at Java SE 6 level
    - Product 5655-R31
  - IBM 64-bit SDK for z/OS, Java Technology Edition, Version 6.0.0
    - Web available on December 14, 2007 at Java SE 6 level
    - Product 5655-R32
- All products are delivered via the z/OS Java website in non-SMP/E format and via ShopIBM in SMP/E format
- All products are independently orderable and serviceable and follow the z/OS RFA rules for Withdrawal from Marketing and End of Service

Trademarks and Copyrights

- Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both
- Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries
- Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Performance - Disclaimer

- Based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput improvements equivalent to the performance ratios stated here.