The Next Generation of Java Virtual Machines

John Duimovich

IBM Distinguished Engineer

Java CTO

About me

- IBM Java CTO, responsible for IBM Java, JavaScript and “other” runtimes
- 25 years experience developing virtual machines, runtimes, tools
- Developer of J9, IBM's high performance production JVM
- One of many creators of Eclipse, currently Tools PMC lead
- Smalltalk still the love of my life

- john_duimovich@ca.ibm.com
- http://duimovich.blogspot.com

Abstract

“The next generation of ~~Java~~ Virtual Machines"

Abstract: A virtual machine is like your mom. Keeps you safe, protects you from the complex world of operating systems, sharp objects like pointers, teaches you right from wrong, prevents you from doing dangerous activities and cleans up after you. These are all good things, but eventually, there is a time to do things that are not quite mom approved.

Sorry mom, I grew up to be a VM implementer

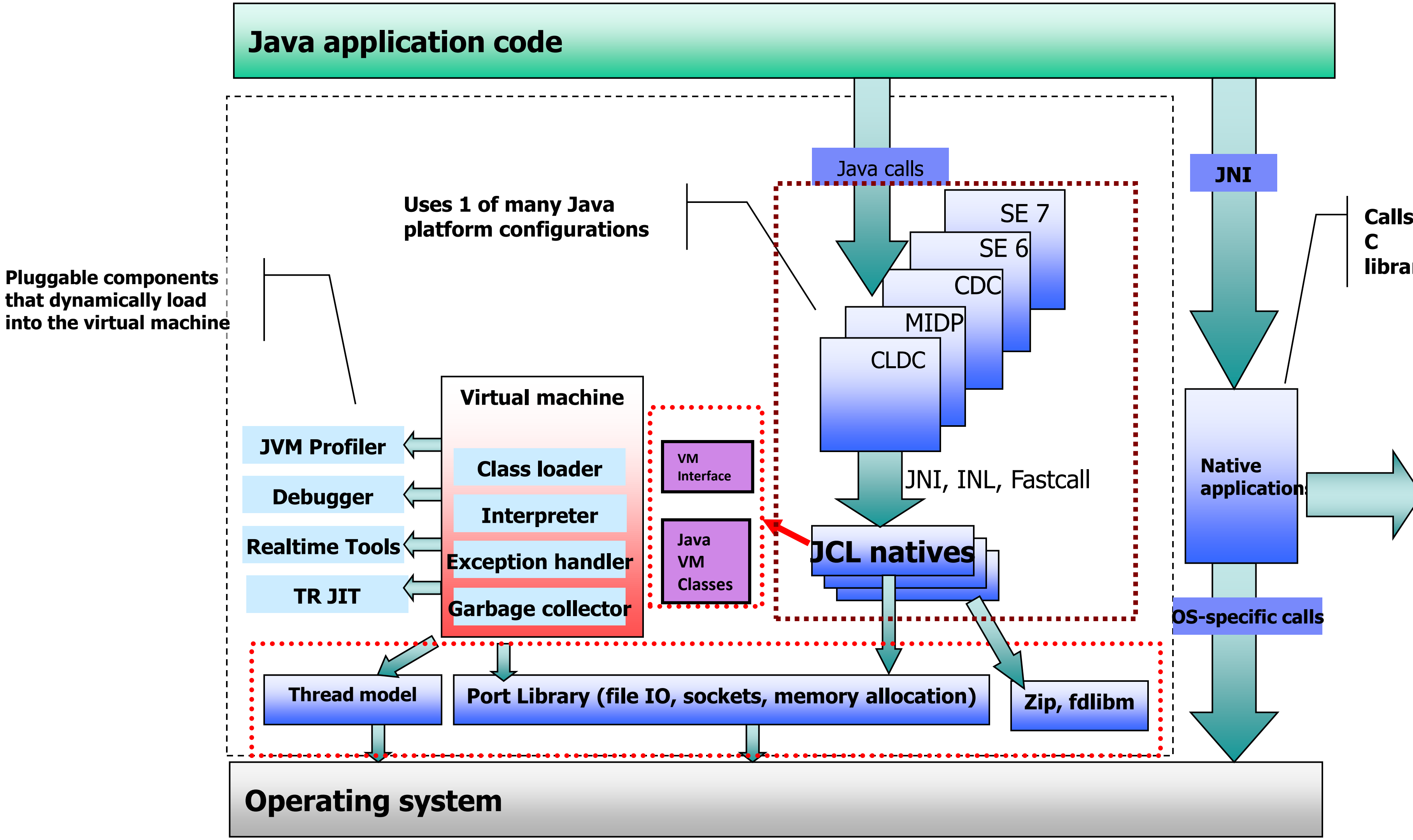
Agenda

- The IBM Java Virtual Machine
  - Technical overview
- Technical Challenges
  - HW evolution
    - Packed Objects and friends
  - Language Evolution
    - Multi-tenancy, Compatability
  - Polyglot ?
- Questions
  - or directions to the Primary Bar Location

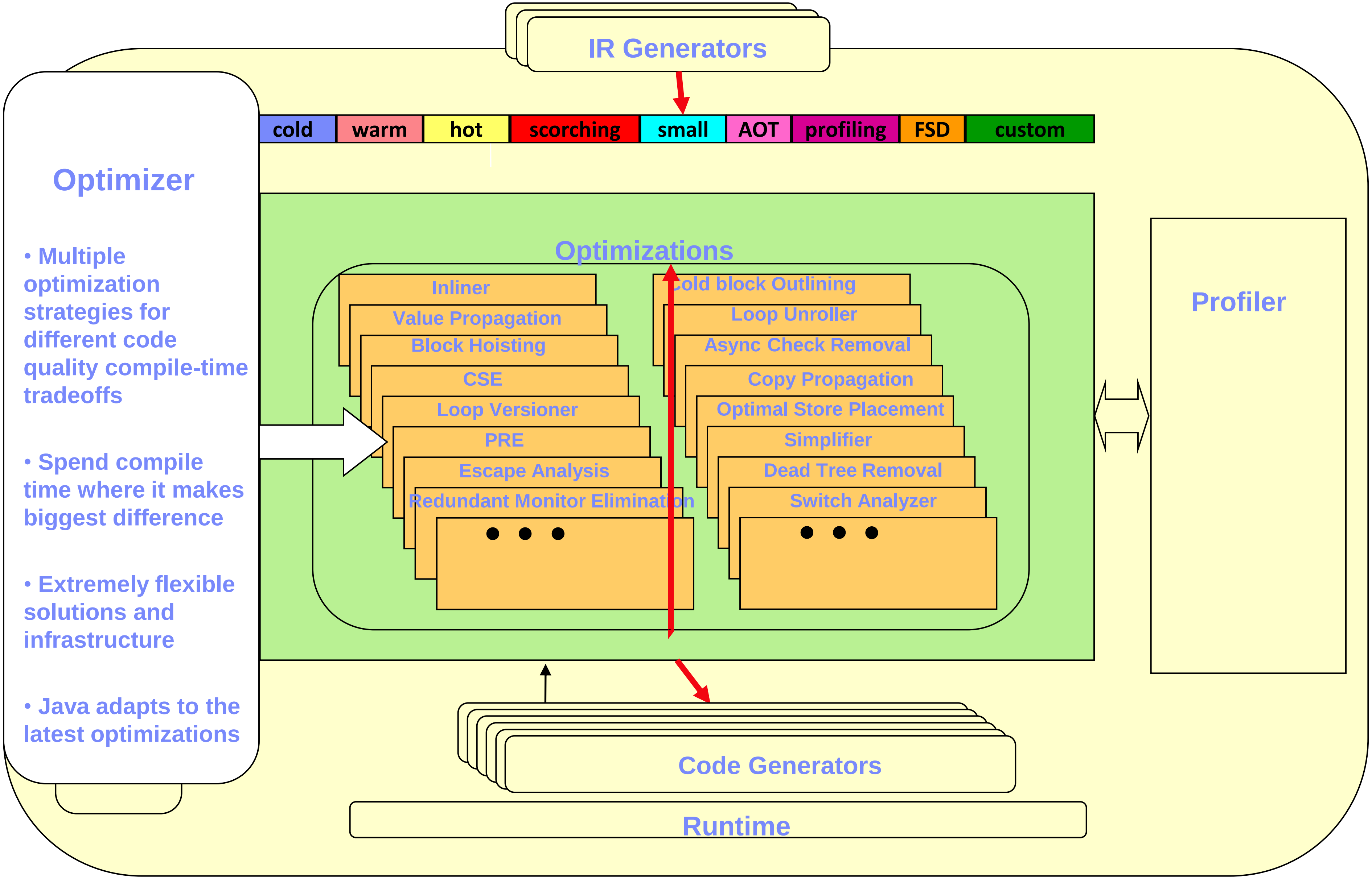
The J9 Virtual Machine

- IBM's strategic virtual machine
  - Designed from the ground up by IBM
  - Focused on high performance , high reliability and serviceability
  - Scales from embedded and handheld devices to large SMPs to mainframes
  - Highly configurable with pluggable interfaces for alternative implementations of GC, JIT
- Composed of several key components
  - Reconfigurable, portable virtual machine framework and interpreter called "J9"
  - Type accurate garbage collection frameworks (Modron, Tarok, Metronome)
  - Highly optimizing just-in-time (JIT) compiler "Testarossa"
  - Integrated RAS features to enhance problem determination
  - Unique Features - SharedClasses, Dynamic AOT

J9 Architecture

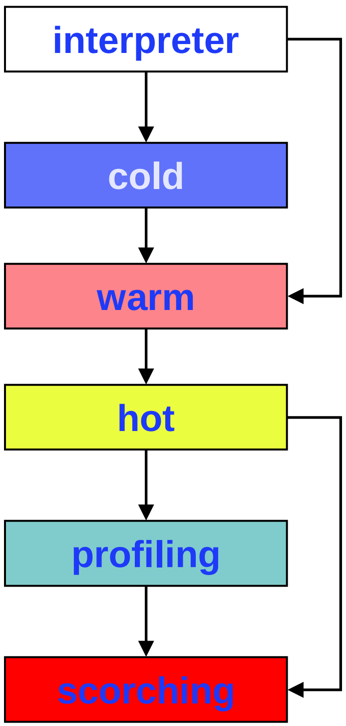


Testarossa: Code optimizations are common across dynamic/static compilers



Java Adaptive Compilation : trade off effort and benefit

Java's bytecodes are compiled as required, optimized based on runtime profiling
- **Dynamic compilation determines the target machine capabilities and app demands**
- **Multiple phases, enable adaptive response to changing environment**



- Methods start out running bytecode form directly
- After many invocations (or via sampling) code get compiled at ‘cold’ or ‘warm’ level
- Low overhead sampling thread is used to identify hot methods
- Methods may get recompiled at ‘hot’ or ‘scorching’ levels (for more optimizations)
- Transition to ‘scorching’ goes through a temporary profiling step

Results can be stored for future runs and shared across invocations

Example

```
public static int total = 55;

public static int compute(int j, int N, int[] a) {
   int k = 0;
   for (int i = 0; i < N; i++) {
      k = k + j + a[i] + (total + foo());
   }
   return k;
   }

public static int foo() {
   return 75;
   }
```

Optimization level	Code Size (bytes)	Compilation Time (ms)	Wall clock runtime (ms)
Cold	139	2.2	31,685
Warm	265	4	10,078
Hot	436	8.9	7,765
Profiling	1,322	9	n/a
Scorching	578	11	6,187

J9 Garbage Collection

- J9 contains a set of scalable garbage collection policies (5)
  - "Modron", "Tarok" - GC frameworks that enables policies to be configured at runtimes
  - Fully type accurate
  - Parallel global (mark, sweep, compact) and generational collection
  - Partial concurrency at global level
  - Exploitation of OS level features (Virtual Memory, Large pages)
  - Common code base across J2SE and J2ME implementations
  - Highly configurable from command or invocation API

Standard Collection Policies

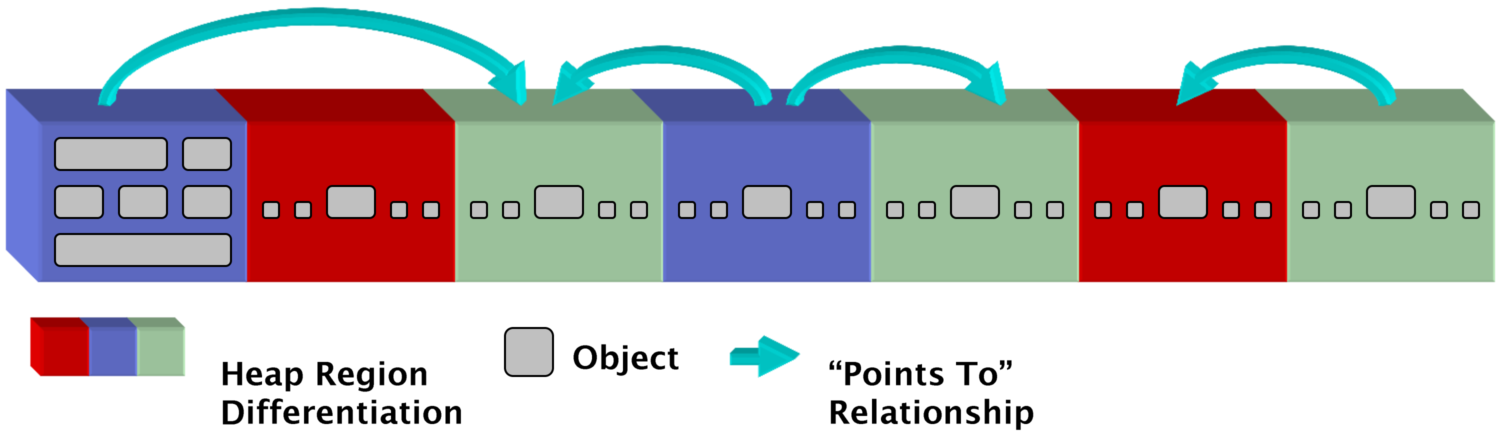
- Modes of operation (-Xgcpolicy:)
  - Throughput (**Optthruput**)
    - Stop the world mark/sweep/compact (MSC) collector with all stages being parallel
  - Average Pause (**Optavgpause**)
    - MSC with concurrent mark and sweep phases to reduce average pause times
  - Generational Collector (**Gencon**)
    - Partial concurrent-mark for old-space and “semi-space” collected new area

      Parallel Copy, Tilted New Spaces, Dynamic New Space resizing
  - Large Heap Multicore (**Subpool**)
    - Throughput with high performance allocator for large heaps/CPU’s
    - Specialty use for large MP systems, deprecated in favour of Balanced.
  - Multi-region Large Heap (**Balanced**)
    - Region based collection supporting partial gc, high mobility, differentiated memory, goal based collections, with ROI heuristics
    - Reduces maximum pause times in very large heaps
    - Native memory aware reduces non-object heap consumption

- Metronome
  - Realtime GC with hard realtime configurability, max cpu utilization, max pause, max memory
  - Hard Real-time version requires RT OS, Soft-RT on vanilla OS

Balanced: Segregate Memory by Differentiators

Heap



- Gather objects with common properties
  - Locality - sibling, child
  - Usage frequency
  - Lifetime, birthplace, resting place
  - Levels of "Read-only"ness
    - none, R/O, mostly R/O, HW enforced

✓**Placement improves cache performance**
✓**Allocation efficiency (Throughput)**
✓**Reduced size working set (Reduce paging, optimize paging)**

- Optimize based on memory characteristics
  - Sharing status of mapped pages
  - Regions grouped by memory speed
    - NUMA, Tiered memory
  - Flash, SSD, GPU exploitation
    - Swapping, compression, LRU, custom formats

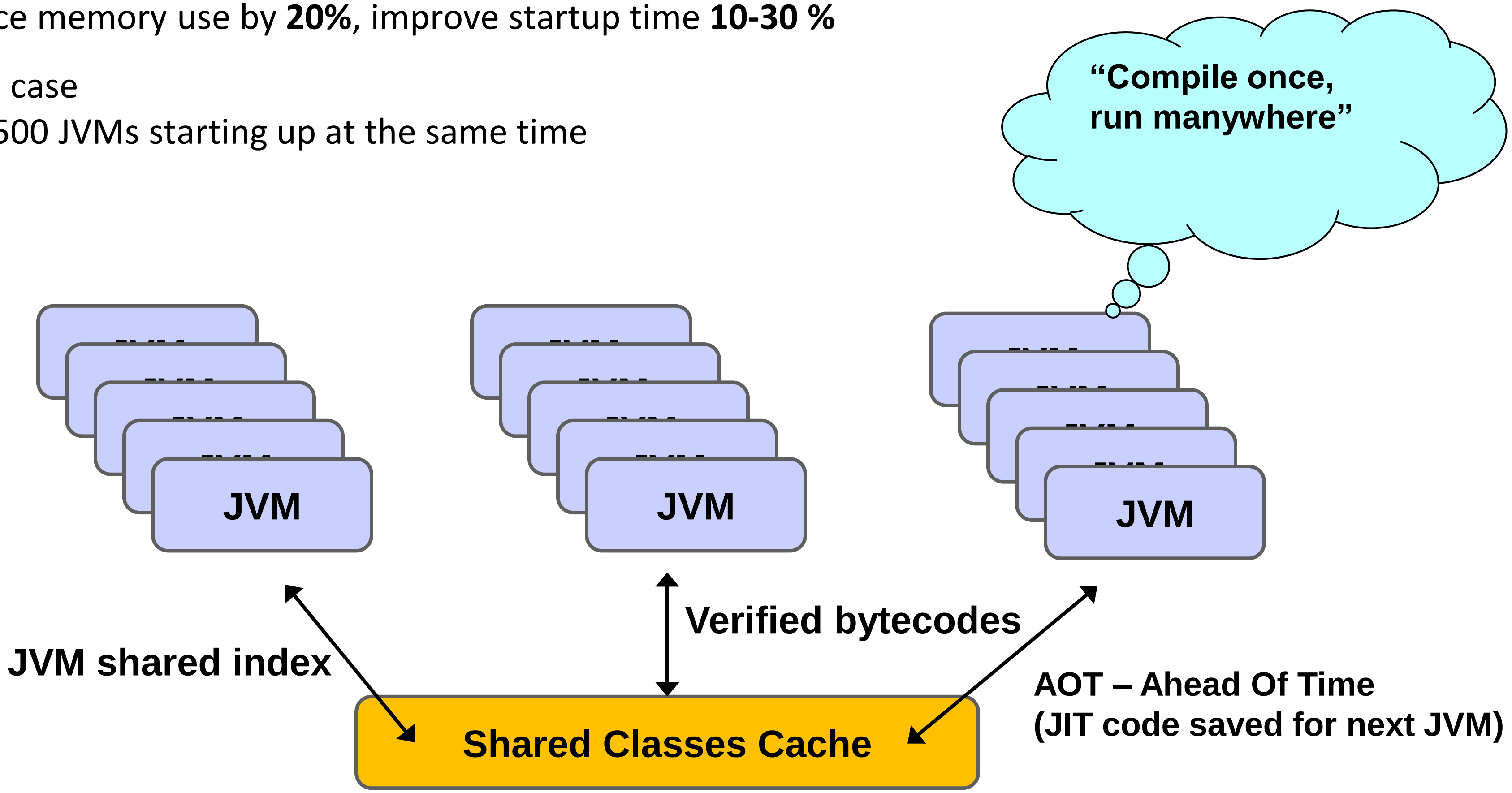
✓**IaaS optimization and sharing (consolidation)**
✓**SSD Exploitation (memory efficiency)**
✓**Multicore scalability (scale up)**

- "Results based" incremental operations
  - productive GC every cycle
  - localized garbage collect

✓**Very Large Heap GC with low pause times**
✓**Soft Real-Time**

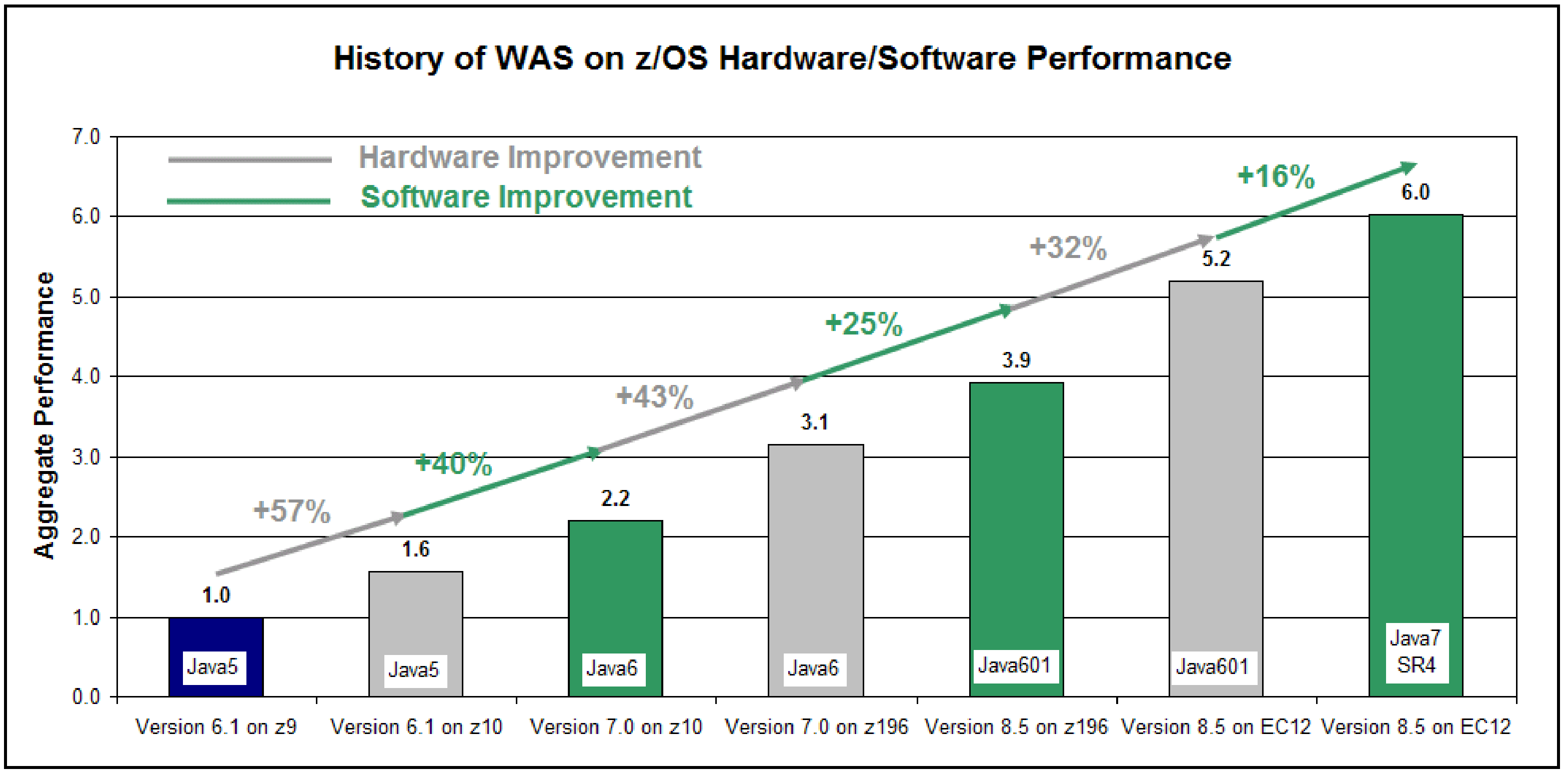
Shared Classes Cache

- J9 JVMs use sharing to reduce memory and startup costs
  - Ability to securely common Java class code across multiple JVM instances
  - Reduces footprint due to sharing of read-only components (Java code)
  - Reduces startup time by caching “ready to run” previously JITed code (Dynamic AOT)
  - Dynamic AOT - reuse JIT code from multiple JVMs
  - Reduce memory use by **20%**, improve startup time **10-30 %**
- Cloud use case
  - 100-500 JVMs starting up at the same time



Performance

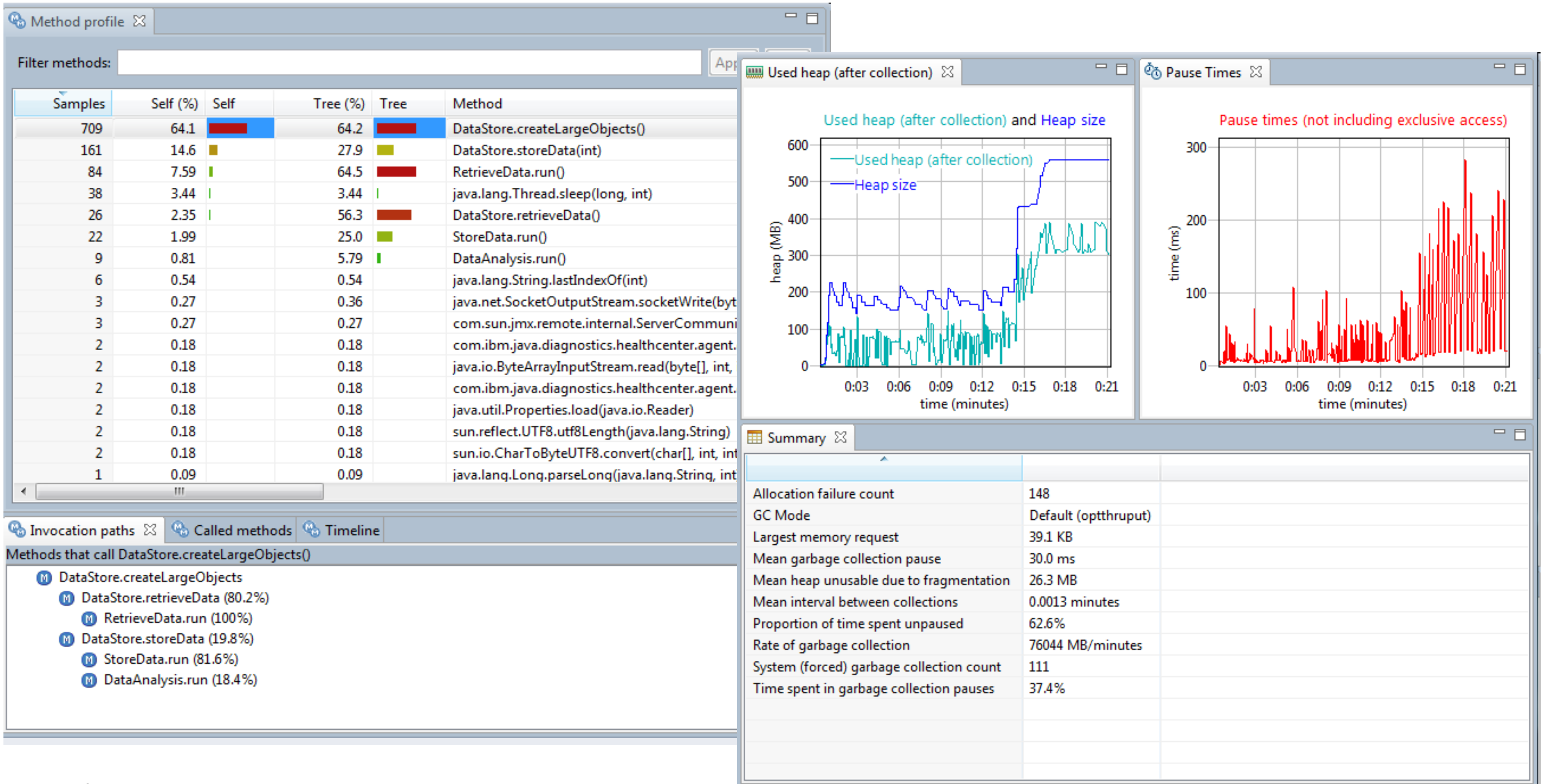
HW, Java and WAS Improvements: WAS 6.1 (IBM Java 5) on z9 to WAS 8.5 (IBM Java 7) on zEC12



6x aggregate hardware and software improvement comparing WAS 6.1 IBM Java5 on z9 to WAS 8.5 IBM Java7 on zEC12

(Controlled measurement environment, results may vary)

IBM Monitoring and Diagnostic Tools for Java - Health Center

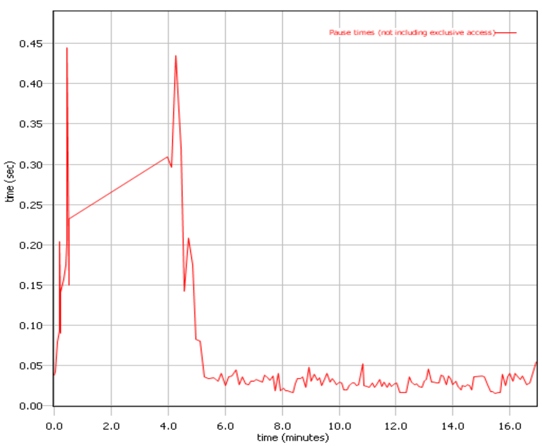


Overview

- Lightweight live monitoring tool with very low overhead (<2%)
- 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

IBM Monitoring and Diagnostic Tools for Java - GCMV

- Recommendations guide 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.

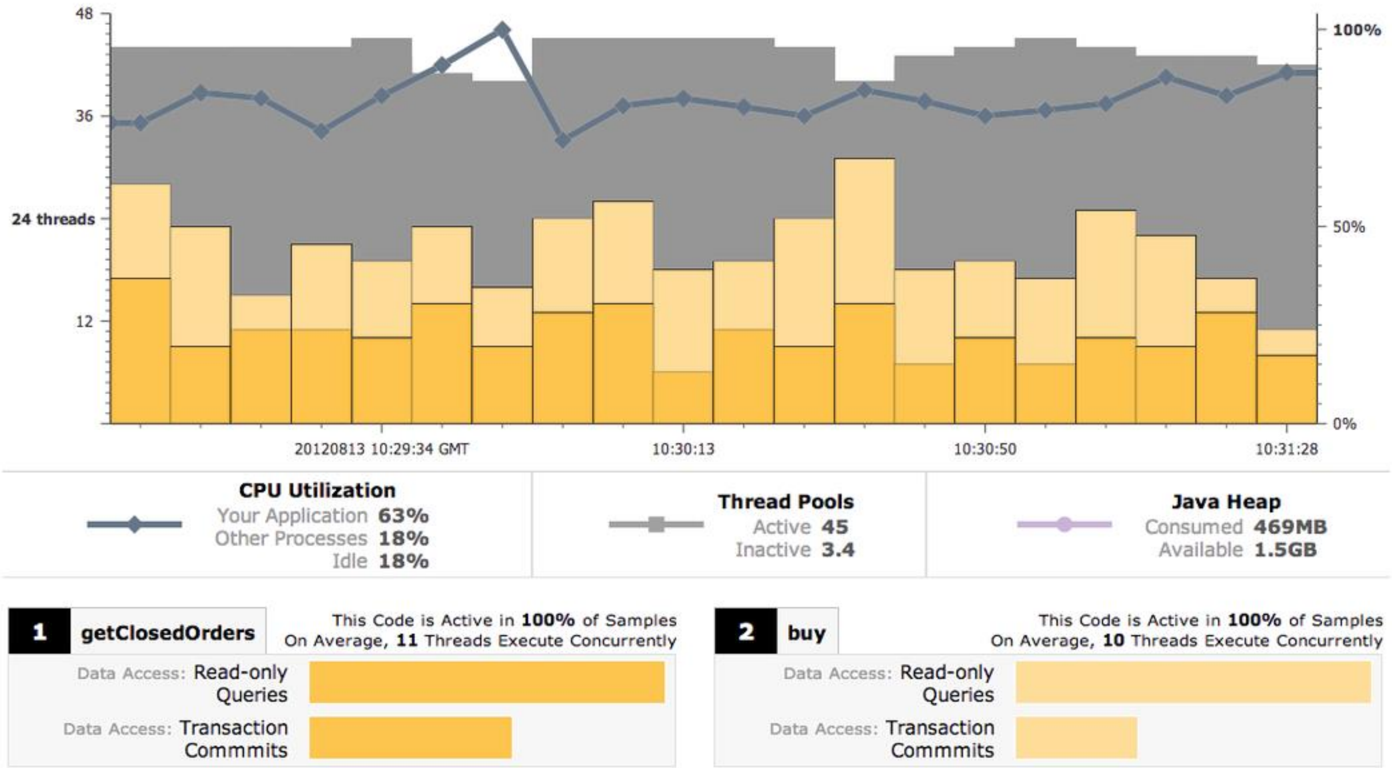
i The mean occupancy in the tenured area is 14%. This is low, so you have some room to shrink the heap if required.

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

Cloud Based Monitoring

- https://wait.ibm.com/



VMs are perfect !

so

We're Done Right ?

Challenges

- "Cloud"
  - virtualization, footprint/density, runtime dynamism, cost
- "Big*.*"
  - more data, more threads, more(and less) memory, scale out, scale up
- Compatibility
  - protect existing investment in code and tools while delivering innovation
- HW Evolution
  - HW is evolving faster than SW can keep up (in some cases)
  - Existing HW may also not have great Java language support (PackedDecimal)
- Security
  - innovation required to drive simplified security
  - VM assist to deliver multiple lines of defense
- Plus … many more … development efficiency, simplicity, software lifecycle

So, how can the virtual machine help solve these challenges ?

The wish list …

- Solve Cloud, Big*.*, Security and retain Compatibility

- And when you have a chance, please add
  - Runtime resource control {memory, sockets, files} scoped by {context, module}
  - Runtime capabilities use {locks, threads, finalization} scoped by {context, modules}
  - Language extensions for new primitive types (value types) and packed data formats
  - High performance memory model primitives for improved parallelism (no unsafe)
  - High performance Foreign Function Interface(FFI) and “Structs”, a better Java Native Interface (JNI)
  - Reified generics and true lambdas
  - Large arrays, restartable exceptions, read-only objects, unsigned ints
  - VM support for compatibility and evolution
  - … and 200 other things

Cloud

Density - Java

- Density improvements span multi-jvm and mult-tenancy (single JVM) delivery models
- Cloud with <u>metered consumption pricing</u> driving accurate and flexible options in the JVM

JRE Isolates + Heap Throttle
☑ Single-digit MB / tenant
☒ Process isolation
☑ No code changes

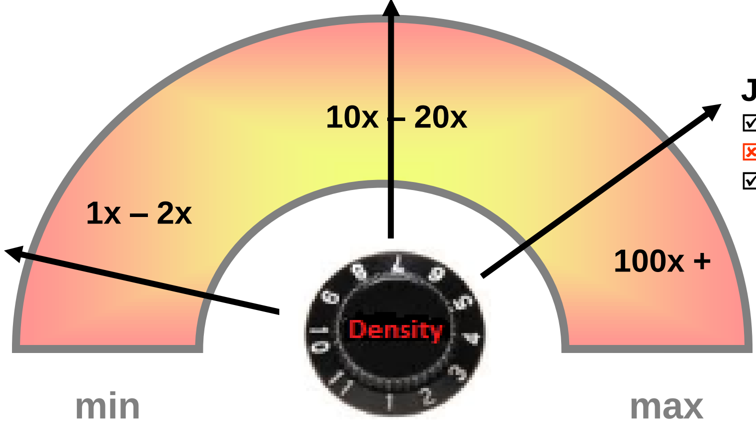
IaaS Optimization
☑ Hypervisor integration AMD/AME
☑ App migration
☑ Workload balancer integration

Snapshots
☑ Single-digit MB / tenant
☑ Process isolation
☒ Code changes needed

PaaS/SaaS Models
☒ Programming model changes
☑ Highest density, lowest cost per tenant
☑ App light, right-sized deployments

JRE Isolates
☑ 10's KB / tenant
☒ Process isolation
☑ No code changes

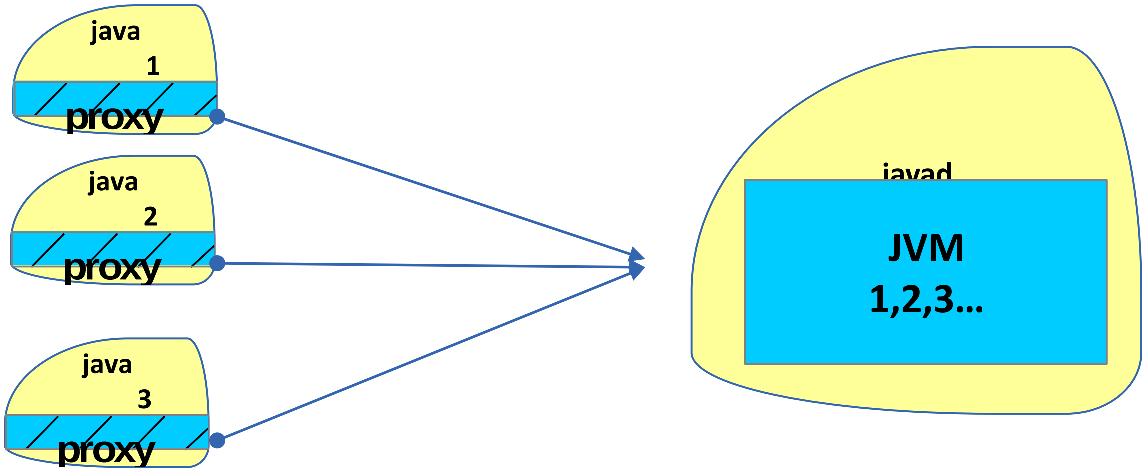
Shared Classes
☑ 20% saving
☑ Java 5+
☑ Process isolation
☒ Double digit MB footprint



JRE with Mark-up
☑ 1's KB / tenant
☒ Process isolation
☒ Code changes needed

Multitenancy

- Application density: a measure of how many applications we can pack onto a piece of hardware
- For many Java applications memory is the density-limiting factor, because:
  - Java heaps are big and aren't shared between JVMs
  - JIT compiled code and metadata are big and aren't shared between JVMs
  - Most JVM heuristics are tuned for maximum performance not footprint reduction
  - Shared services (GC, JIT) don't co-operate between JVMs even when on same physical server
- Multitenancy support 'virtualizes' the JVM
  - Sharing a runtime enables maxiumum artifact sharing (classes, code) between applications
  - **Isolation** of the statics while sharing classes
  - **Isolation** of resources to ensure friendly neighbour behaviour



Data Isolation in shared classes

- **The MT JVM provides isolation contexts and each context has a separate set static variables**
- **Using the `@TenantScope` annotation.**

LocaleSettings.java

```java
package javaone;

import java.util.Locale;
import com.ibm.tenant.TenantScope;

public final class LocaleSettings {

    public static final Locale CANADA = new Locale("en", "CA");
    public static final Locale UK = new Locale("en", "GB");
    public static final Locale USA = new Locale("en", "US");

    private static @TenantScope Locale defaultLocale = CANADA;

    public static void setDefaultLocale(Locale defaultLocale) {

    public static Locale getDefaultLocale() {
```

Tenant1

```java
...
LocaleSettings.setDefaultLocale(
LocaleSettings.UK );
...
```

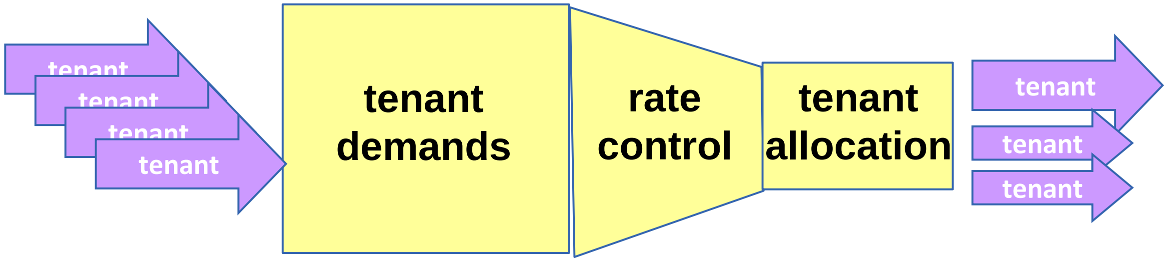
Tenant2

```java
...
LocaleSettings.setDefaultLocale(
LocaleSettings.USA );
...
```

- **`@TenantScope` Semantics: Static variable values are stored per-tenant**
- **Each tenant has their own `LocaleSettings.defaultLocale`**
- **Now many tenants can share a single `LocaleSettings class`**

Multitenancy : Resource Management

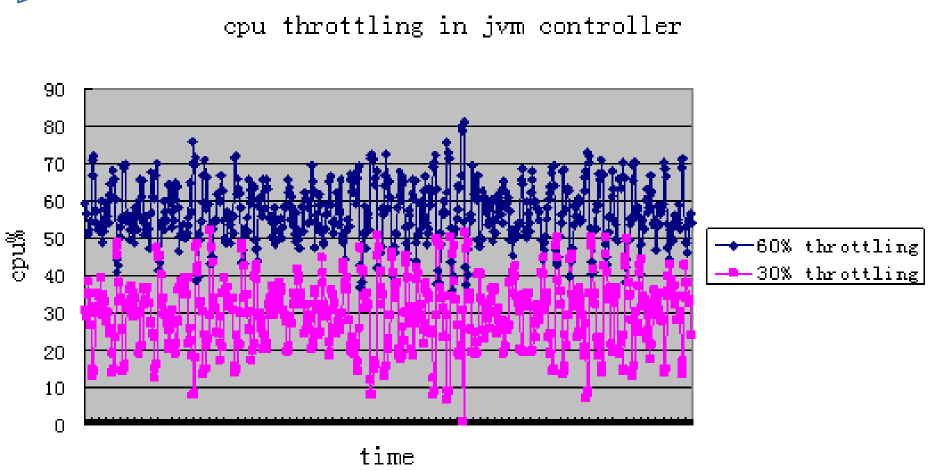
- Tenants scoped resource consumption rate by policy
  - Control of CPU percentage, number of threads, heap memory, disk and Network I/O
  - Usage controlled per tenant per second
- Controlled using : -Xlimit:<resource_name>=<min_limit>-<max_limit>
  - <min_limit>: minimum amount of the resource required to start.
  - <max_limit>: maximum amount of the resource allowed to use.



CPU-intensive apps each doing the same Fibonacci calculation, but with different CPU quota: 60% and 30%

java -Xmt -Xlimit:cpu=60 -jar fibonacci

java -Xmt -Xlimit:cpu=30 -jar fibonacci

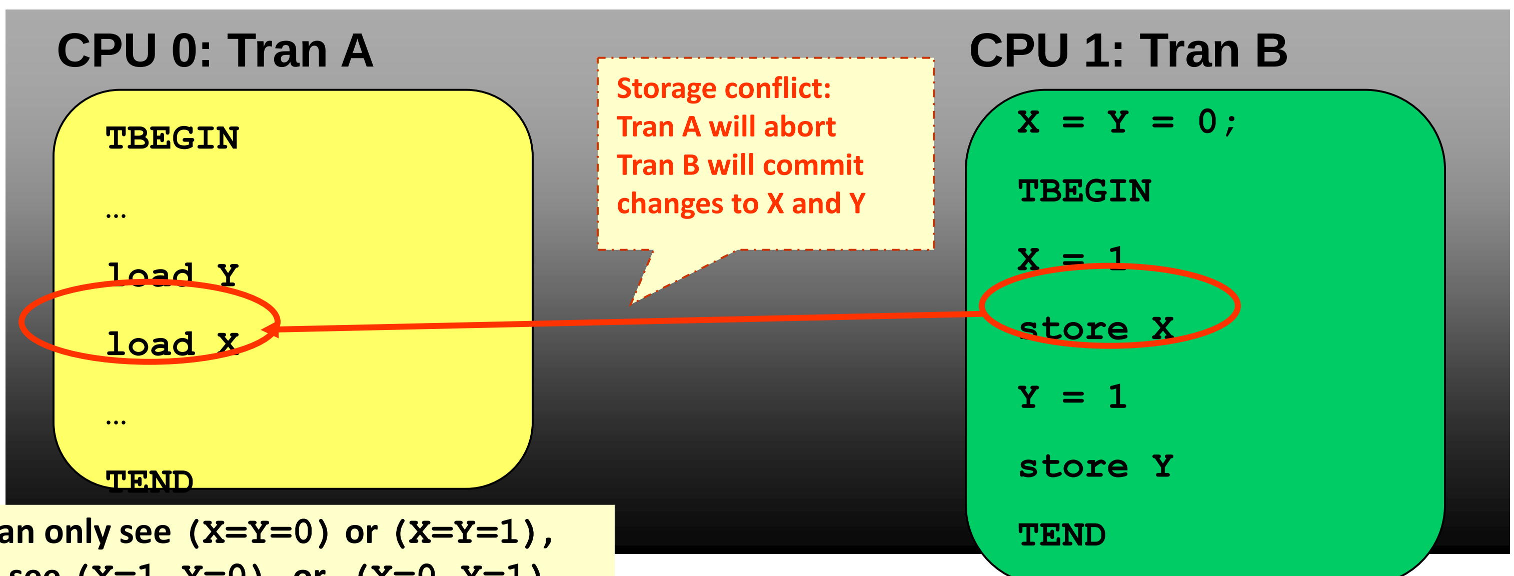


“Big*.*”

More
threads, memory
More
Problems

Hardware Transactional Memory (HTM)

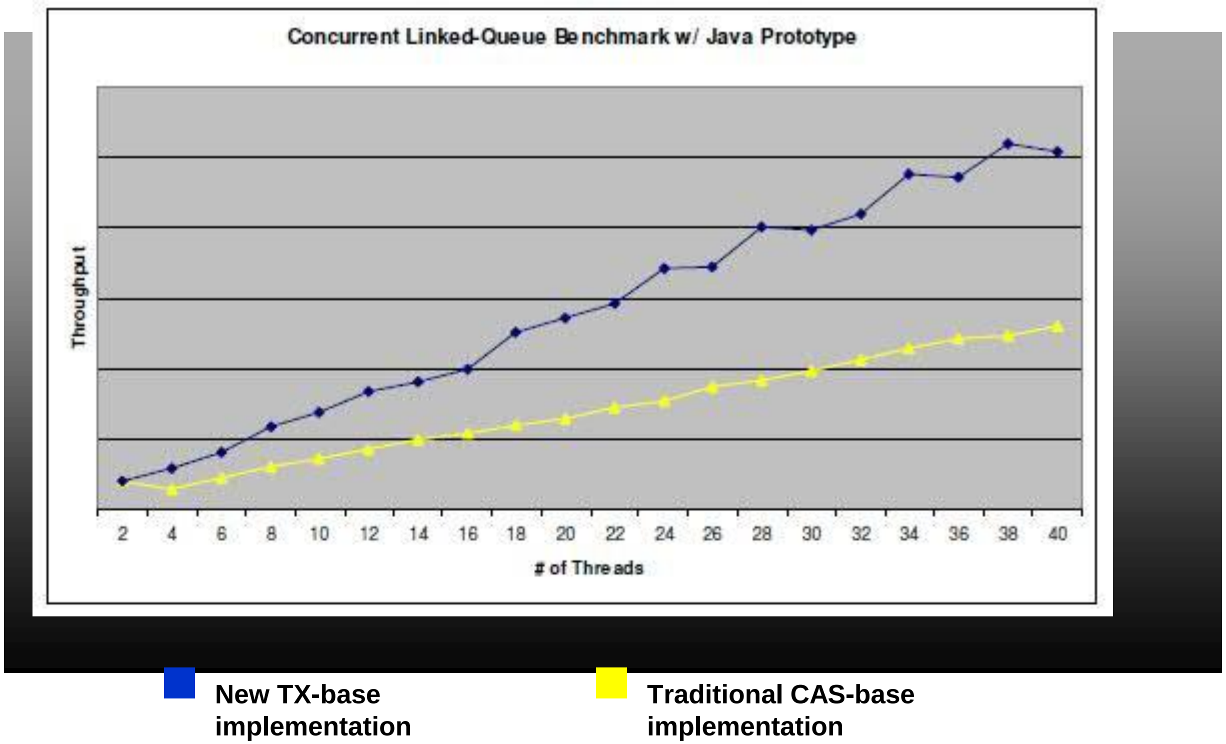
- Allow lockless interlocked execution of a block of code called a ‘transaction’
  - Transaction: Segment of code that appears to execute ‘atomically’ to other CPUs
    - Other processors in the system will either see all-or-none of the storage up-dates of transaction
- How it works:
  - TBEGIN instruction 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

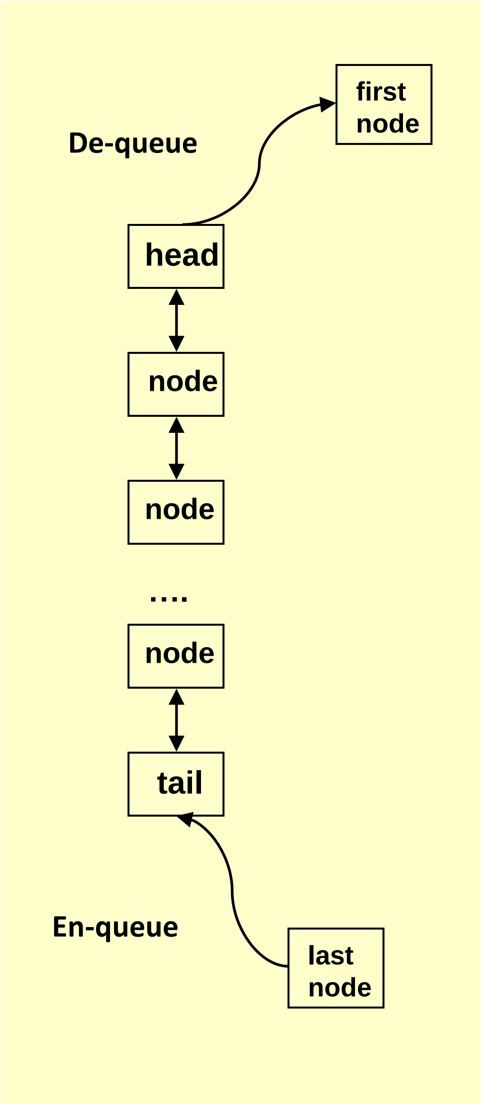


CPU 0 can only see (X=Y=0) or (X=Y=1), cannot see (X=1,Y=0) or (X=0,Y=1)

Transactional Execution: Concurrent Linked Queue

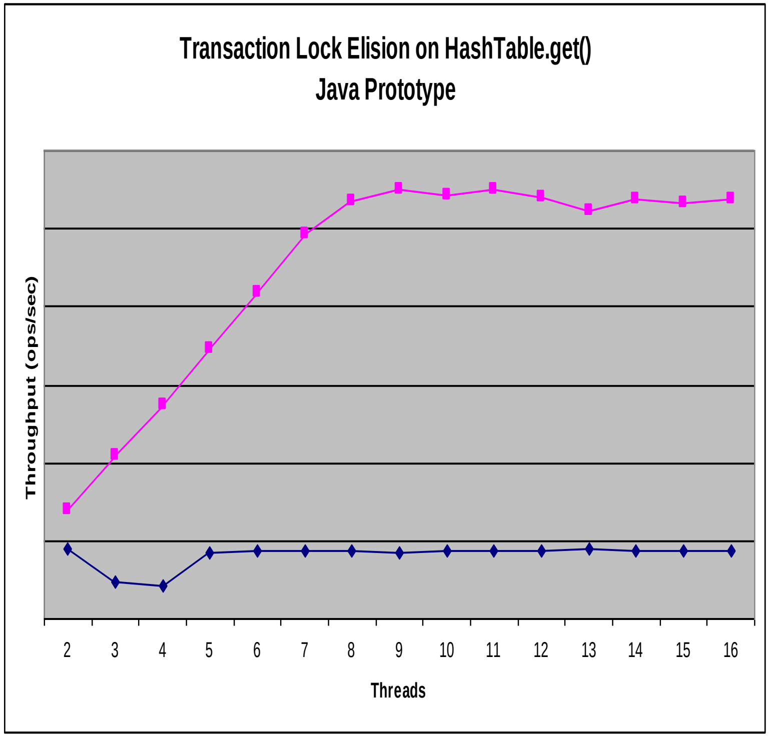
- ~2x improved scalability of juc.ConcurrentLinkedQueue
- Unbound Thread-Safe LinkedQueue
  - First-in-first-out (FIFO)
    - Insert elements into tail (en-queue)
    - Poll elements from head (de-queue)
  - No explicit locking required
- Example: a multi-threaded work queue
  - Tasks are inserted into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





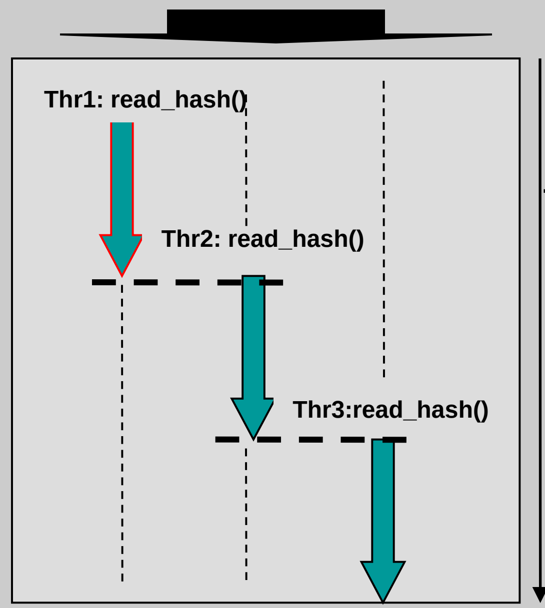
(Controlled measurement environment, results may vary)

HTM Example: Transactional Lock Elision (TLE)



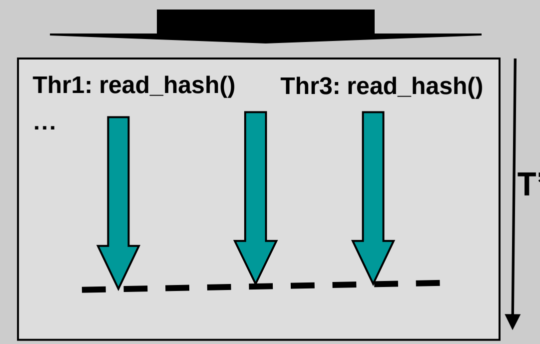
Threads must serialize despite only reading… just in-case a writer updates the hash

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



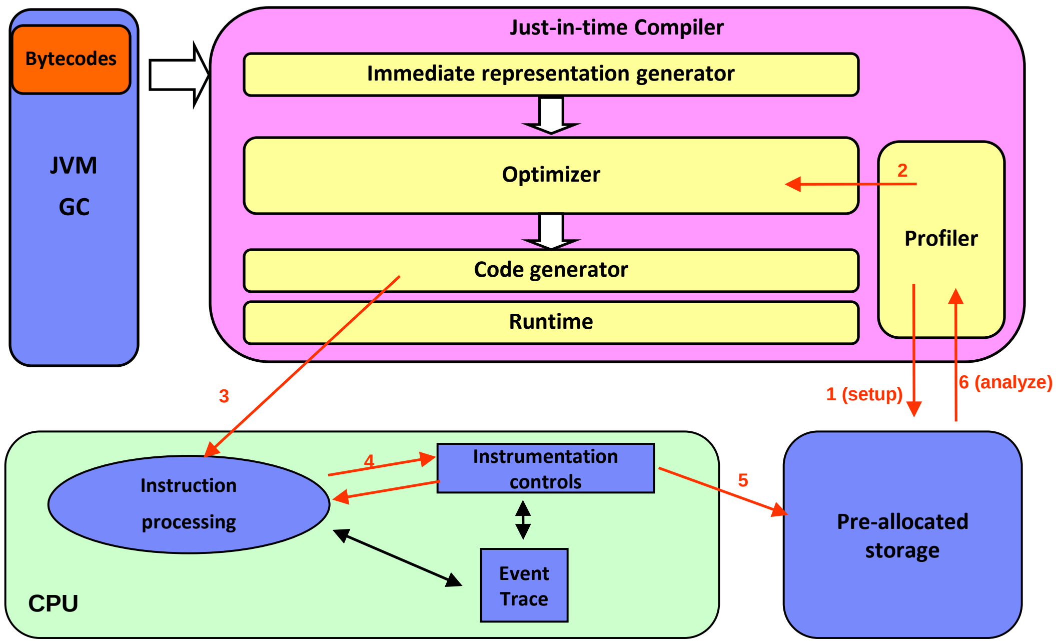
Lock elision allows readers to execute in parallel, and safely back-out should a writer update hash

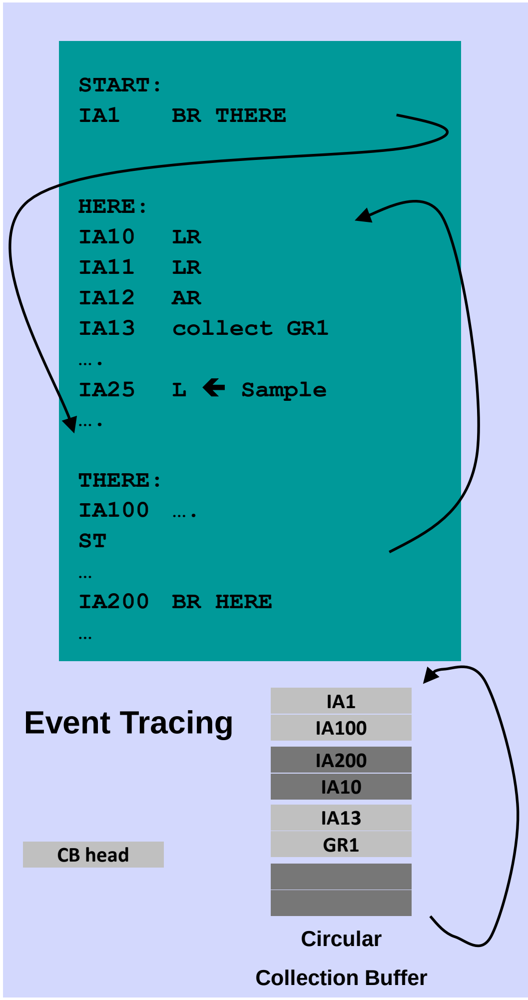
```
read_hash(key)
  TRANSACTION_BEGIN
  read hash.lock;
  BRNE serialize_on_hash_lock
  read (hash, key);
  TRANSACTION_END
```



Runtime Instrumentation in HW is coming

- *Low overhead profiling with hardware support*
  - Instruction samples by time, count or explicit marking
- *Sample reports include hard-to-get information:*
  - Event traces, e.g. taken branch trace
  - "costly" events of interest, e.g. cache miss information
  - GR value profiling
- *Enables better "self-tuning" opportunities*





Compatibility

Compatibility

- Binary Compatibility has been a key strength of Java
  – Protect customer investment by “never” breaking code
  – Clean API specification ensures evolution of API within specification and TCK
  – *Ridiculously* old code will run unmodified
- Supporting “eternal” binary compatibility is difficult
  – JVM complexity – old code drives special use cases in JVM, including security issues
  – Code bloat – deprecated is “just a suggestion” and code is loaded whether it’s used or not
- JVM and Platform support for evolution of Java language and API ?
  – It’s beginning !

Cudos ! to Extension (default) Methods in Java 8

Default methods

- Lambda and stream operations are useful on existing collection types
  - Need a way to extend well established type structures while retaining **compatibility**
- Option 1: Creating parallel hierarchy of similar structures
  - Bulky class library with constant need to juggle types
- Option 2: Adding a new method to an existing interface
  - Binary compatible, but disenfranchises implementers
- Option 3: Enhance language to provide default implementations in interfaces
  - Interface declarations contain code, or references to code, to run if classes do not pr ntation
  - COMPATIBILITY COMPATIBILITY COMPATIBILITY COMPATIBILITY COMPATIBILITY !

```
public interface Set<T> extends Collection<T> {
default Collections.<T>setForEach { … };
public boolean add(E e);
public void clear();
...
public void forEach(Block<T> blk)}
```

Compatibility evolution enables innovation

Can I have some more please …

VM Support for "breaking changes"

- API evolution as core jvm and platform ?
  - JVM supported field and method rename / rewrite on load ?
  - Selective visibility of methods based on **compile time versions** ?
  - JVM assisted refactoring for complex type cases
  - Source refactoring in IDEs automatically from metadata in binaries
- Serialization robustness
  - Class versions aware of prior versions and shapes and can provide conversions routines
  - Automatic "schema" migration via tools
  - Death to the serialVersionUID
- Optimized deployment
  - Java instances support "strict version" mode to ensure only latest version code loads
  - Optimized instances that offer smaller, faster, efficient deployment
  - Unused components not loaded, corner cases removed, evolve better security
    - Thread.stop, Thread.suspend, Thread.resume

Technology to help address compatible class evolution ?

- public class Example V1.0 {
  - public void a ();
  - public void b ();
  - public int badField;
  - }

Oops, don't use badField

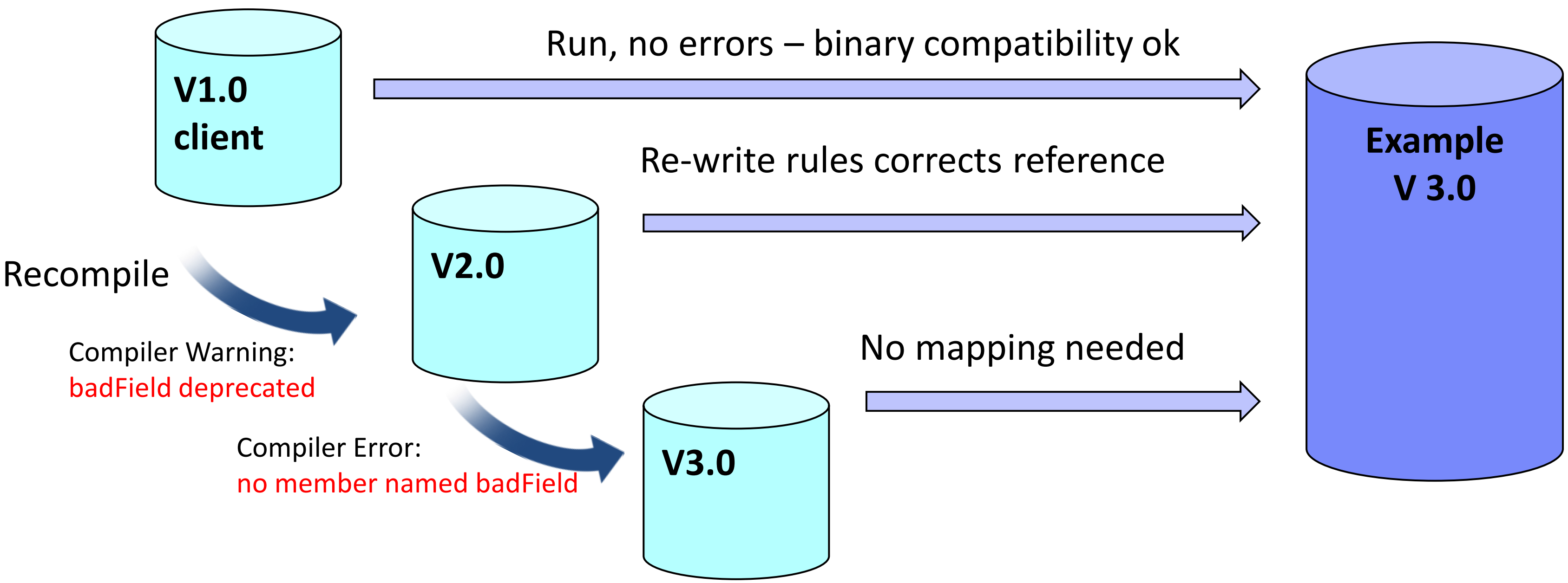
public class Example V2.0 {
public void a ();
public void b ();
deprecated V1.0 public int badField;
public int getField()
public void setField(int);
}

public class Example V3.0 {
public void a ();
public void b ();
public int getField()
public void setField(int);
deprecated V1.0-2.0 public int badField map(getField, setField);
}

Note: Imaginary Syntax

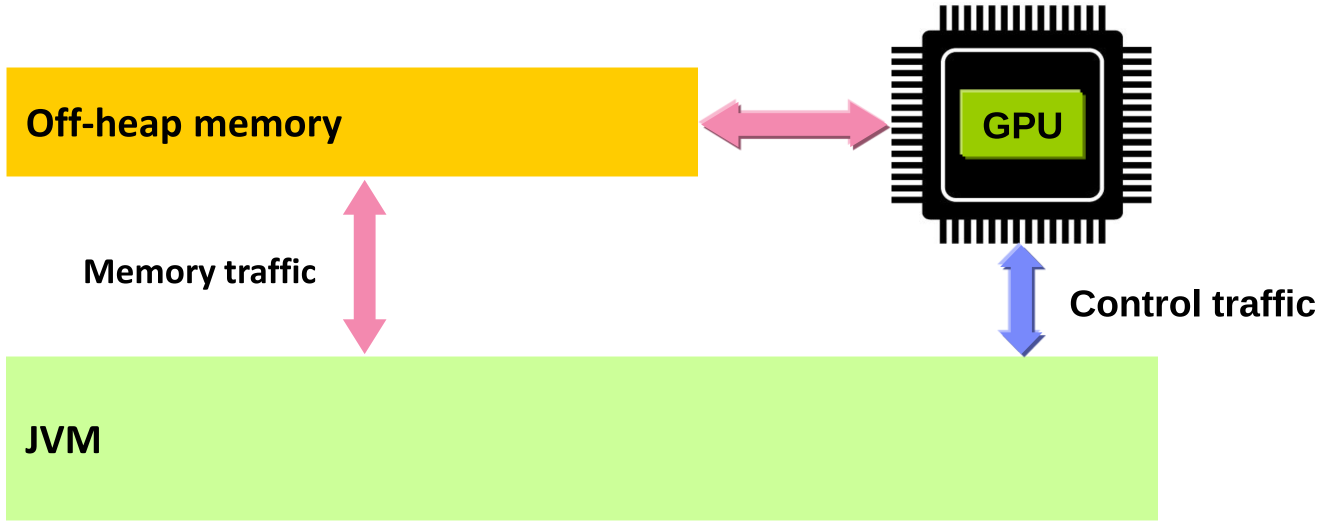
Versions to evolve class shapes

```
public class  Example  V3.0 {
      public void a ();
      public void b ();
      public int getField()
      public void setField(int);
      deprecated V1.0-2.0 public int  badField map(getField, setField);
      }
```



Hardware Evolution
or
Java is in the way

Use case: GPU offloading



- GPU technology growing in importance
  - Fast, low-power, data-parallel operations
  - Analytics, data mining, data conversions
- For Java, data transfer costs between CPU and GPU key inhibitor
  - Data marshaling costs are high, scatter / gather complexity, objects wrong shape
- Java platform needs good support to declare intent for best performance
  - Optimal layout and control of memory needed with freedom for VM to optimize

Packed objects

"Packed Objects" is an experimental language feature available on IBM Java 7 R1 technical preview form.

Packed objects allow developers greater control over the memory layout of their Java objects

Why?

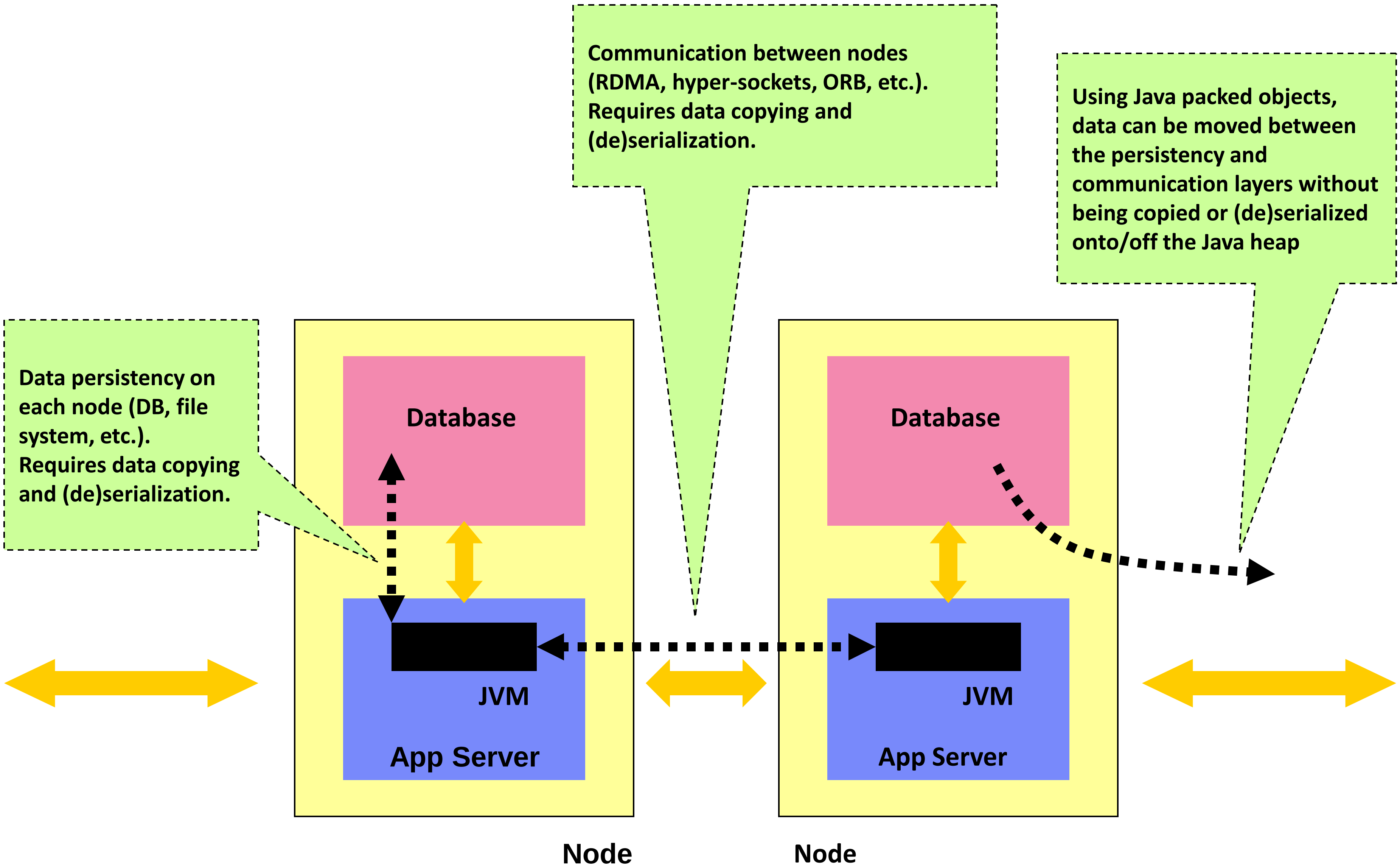
- ✔ Improve serialization and I/O of Java objects
- ✔ Allow direct access to "native" (off-heap) data
- ✔ Allow for explicit source-level representation of compact data-structures

Tech preview with customer feedback on what works and what doesn't so we can inform the standard via JCP process

Benefits:

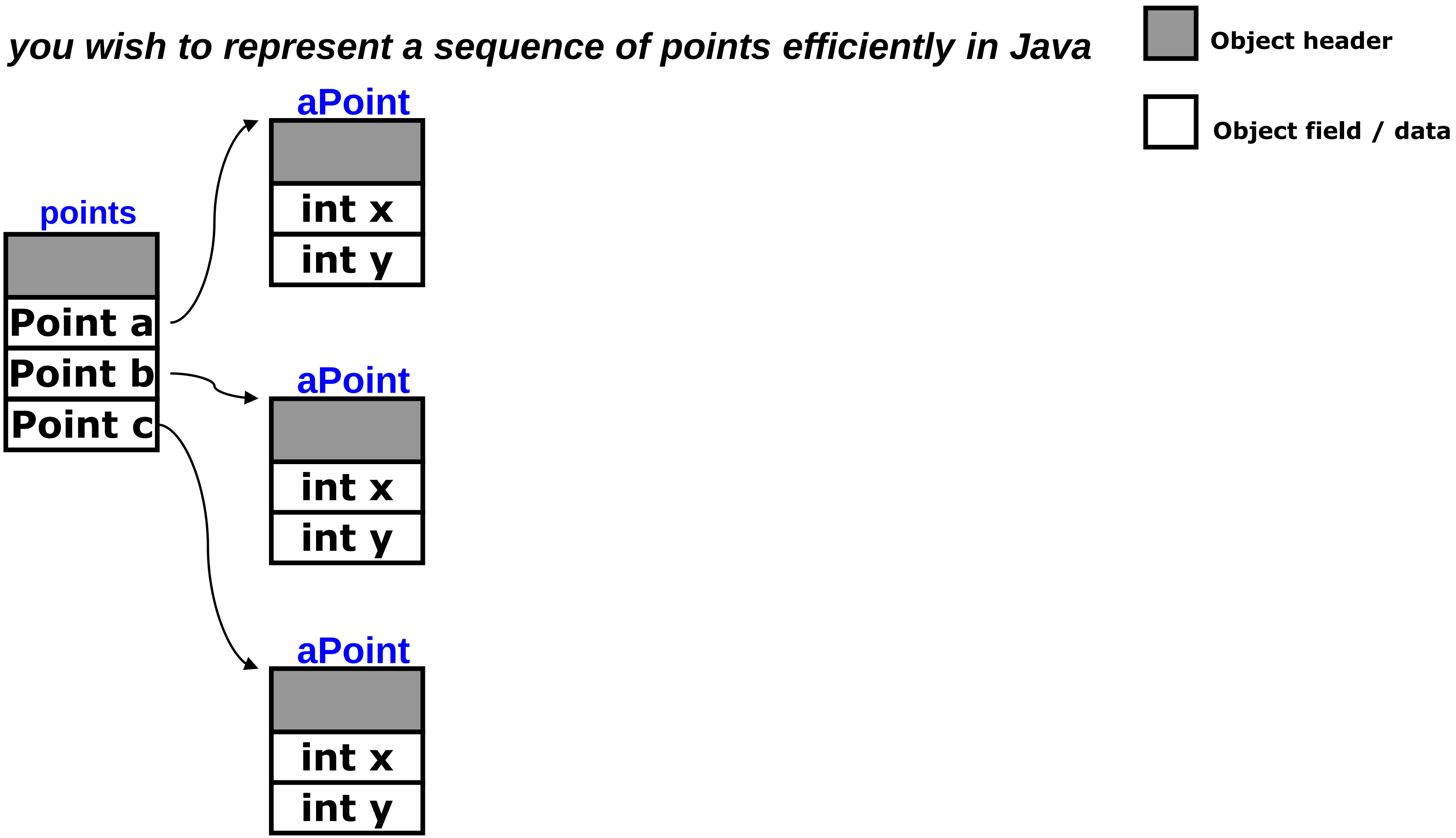
structured data support in Java for improved footprint, enhance serialization performance, enabling fine-grained data management, better inter-language communication

Use case: distributed computing communications

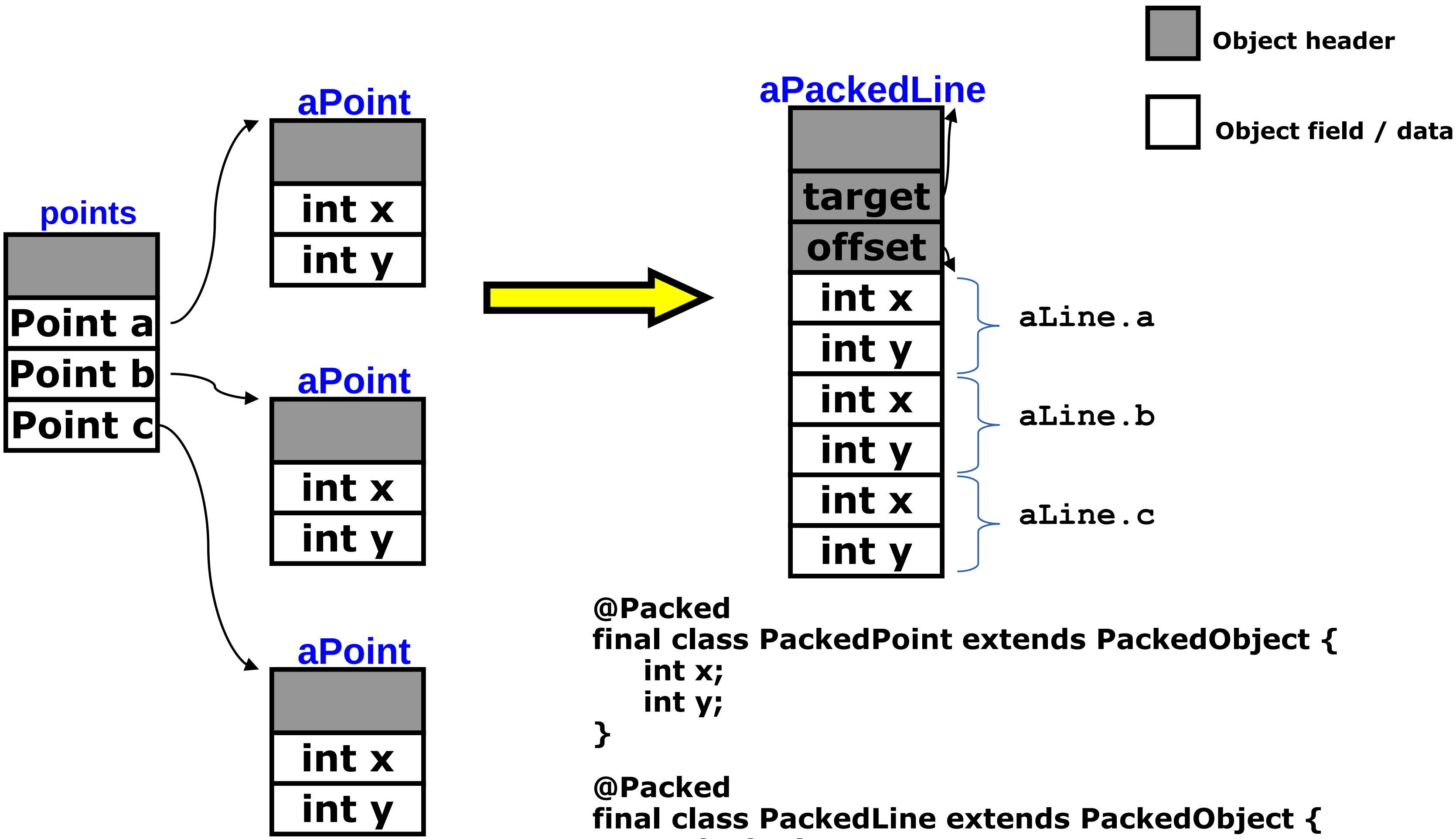


Packed Objects: Heap referenced data

e.g. you wish to represent a sequence of points efficiently in Java



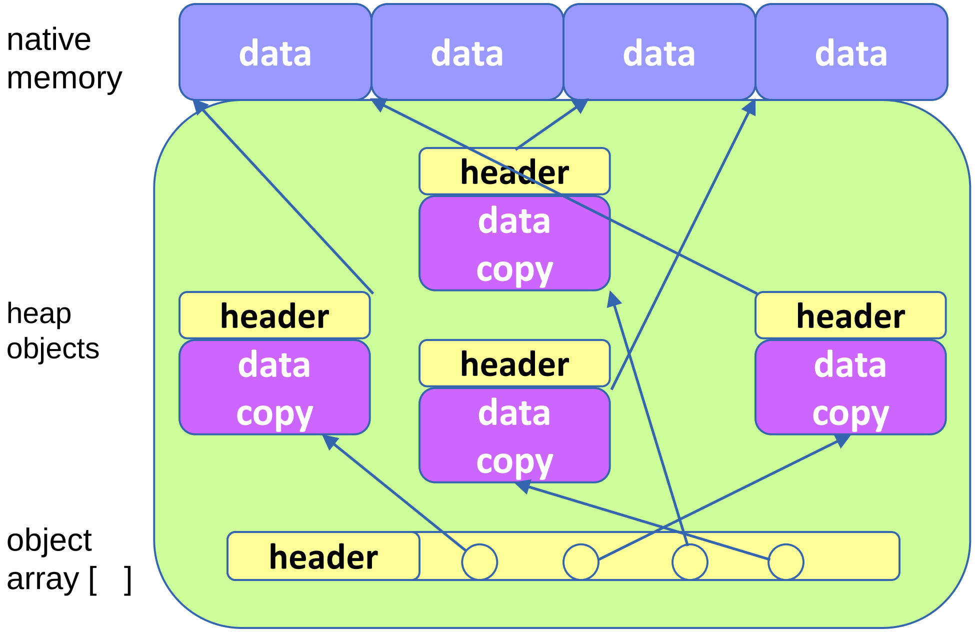
Packed Objects: Heap referenced data



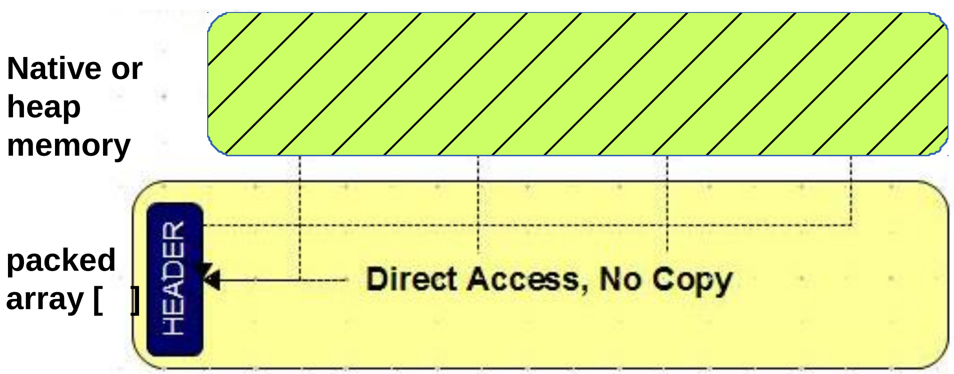

```
@Packed
final class PackedPoint extends PackedObject {
    int x;
    int y;
}

@Packed
final class PackedLine extends PackedObject {
    PackedPoint a;
    PackedPoint b;
    PackedPoint c;
}
```

Packed objects support for native memory

Manipulate native data records from Java



- Java requires memory to be in Java "object" form to be accessed directly
- External data needs to be read into Java heap format to use – conversion is expensive
- Memory bloat due to copies and headers
- Natural object representation looses data locality properties



- **PackedObjects enables direct access to data in arbitrary formats without the redundant copying; no conversion**
- **PackedObjects data can be in native memory or Java heap space**

Code snippets

- Packed class definition

```
@Packed
public final class PackedPrimitives {
   public byte byteField;
   public boolean booleanField
   public double doubleField;
}
```

- On-heap packed allocation

```
// allocate an on-heap ‘struct’ object
PackedPrimitives pp = new PackedPrimitives();
pp.byteField = 0x20;
pp.booleanField = true;
pp.doubleField = 100.7;
```

```
// allocate an off-heap ‘struct’ object and the native memory for its data
PackedPrimitives pp = PackedObject.newNativePackedObject(PackedPrimitives.class, 0);
// directly set fields in native memory
pp.byteField = 0x20;
pp.booleanField = true;
pp.doubleField = 100.7;
…
// free native memory
PackedObject.freeNativePackedObject(pp);
```

"What's next ?"

Polyglot and the virtual machinist

- Developer ecosystem has expanded to include more than "Just Java"
  - Ruby, Python, PHP, JavaScript frequently used in many of our customer applications
  - right tool for the job, programmer efficiency , joy, skills
- PaaS's like CloudFoundry is a polyglot platform for consuming services
- JavaScript is prevalent in most if not all customer applications
  - Multi-channel delivery of customer value drives adoption
  - Web Applications, NoSQL and JSON use a factor
  - Mobile primary driver for adoption
- Java virtual machines have high performance modern implementations

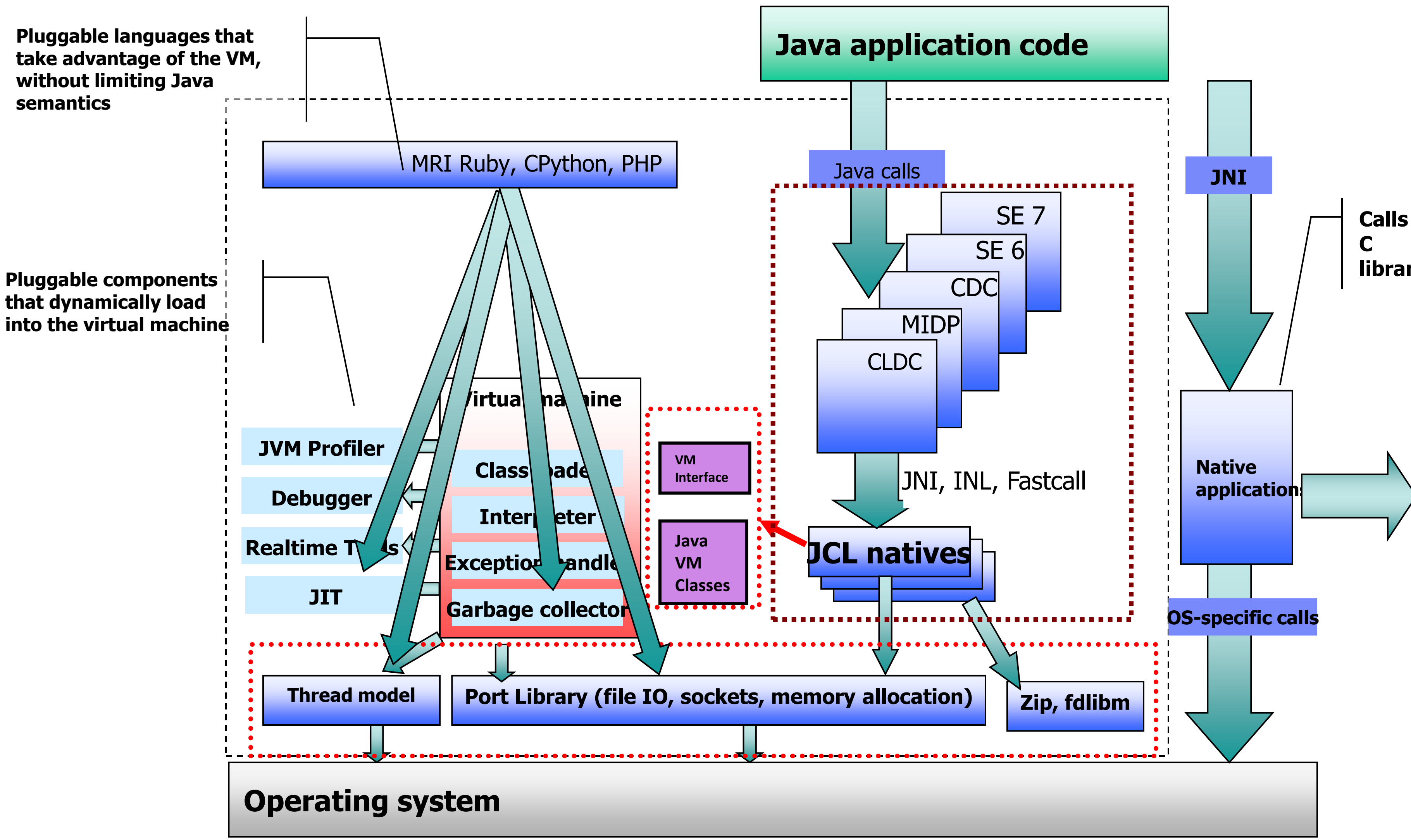
So

Why has the universe not adopted the JVM as the "one true" runtime for best performance of scripting languages ?

How do we best leverage this investment in Java to enable other languages ?

In our runtimes, memory model is flexible and configurable beyond what Java can specify , our JIT is multi-language enabled, and VM support layers are portable so we are exploring what new interfaces the VM platform needs to support for scripting and other dynamic languages

J9 Architecture



So…

- “Next Generation” Virtual Machines will

- Enable higher and higher level of scalability and performance via better GC, JITs
- Enable better interop with native memory, optimized layouts new data types
- Language support for innovation with compatibility migration help
- Speak more languages fluently

Questions