Java without the GC pauses

Keeping up with Moore’s law and living in a virtualized world

Gil Tene, CTO & Co-Founder

Azul Systems

High Level Agenda

1. Application Memory trends
2. GC Discussion
3. Java in a Virtual world

Application memory

Memory.

- How many of you use heap sizes of:
  - Larger than ½ GB?
  - Larger than 1 GB?
  - Larger than 2 GB?
  - Larger than 4 GB?
  - Larger than 10 GB?
  - Larger than 20 GB?
  - Larger than 100 GB?

Reality check: memory in 2011

- Prices from common web-based server store (March, 2011)

24 vCore, 96GB server	~$6.5K
32 vCore, 256GB server	~$20K
48 vCore, 512GB server	~$35K
48 vCore, 1TB server	~$70K

- Cheap (<$2/GB/Month), and roughly linear to ~1TB

Maybe 2-4GB is simply enough?

- We hope not (or we'll all have to look for new jobs soon)
- Plenty of evidence to support pent up need for more heap
- Common use of lateral scale across machines
- Common use of "lateral scale" within machines
- Use of "external" memory with growing data sets
  - Databases certainly keep growing
  - External data caches (memcache, JCache, JavaSpaces)
- Continuous work on the never ending distribution problem
  - More and more reinvention of NUMA
  - Bring data to compute, bring compute to data

How much memory do applications need?

- “640K ought to be enough for anybody”

WRONG! So what’s the right number?

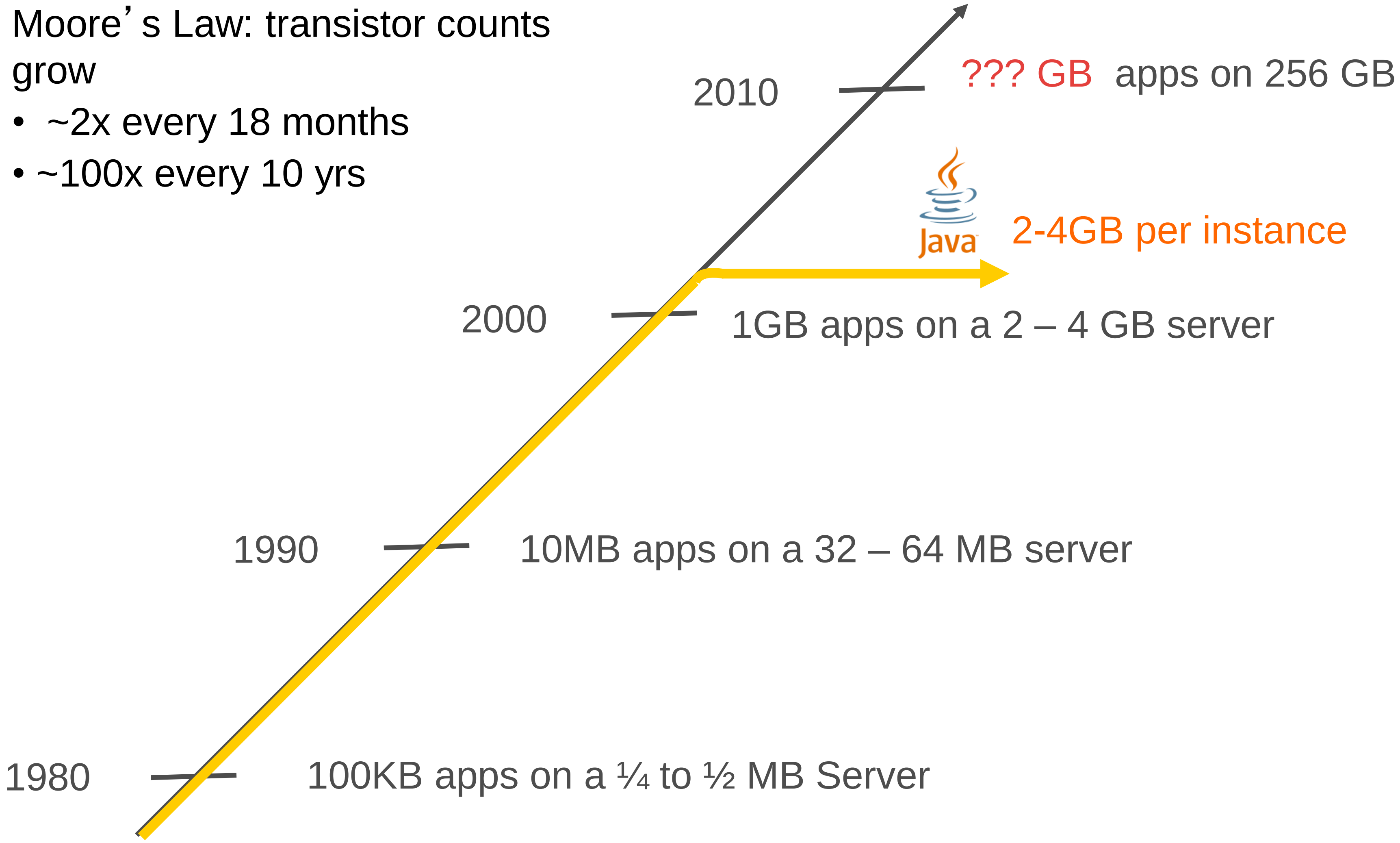
6,400K?	(6.4MB)?
64,000K?	(64MB)?
640,000K?	(640MB)?
6,400,000K?	(6.4GB)?
64,000,000K?	(64GB)?

- There is no right number.
- Target moves at ~100x per decade.

“Tiny” application history

Moore’s Law: transistor counts grow

- ~2x every 18 months
- ~100x every 10 yrs



2010

??? GB apps on 256 GB

2-4GB per instance

2000

1GB apps on a 2 – 4 GB server

1990

10MB apps on a 32 – 64 MB server

1980

100KB apps on a ¼ to ½ MB Server

New programming models?

- The coherent, shared memory SMP model has endured
  - That’s how people program. Still...
- In the past 40 years, new programming models proposed
  - Whenever we run into a new “architectural limit”
  - Usually involve some sort of “loosely coupled memory”
  - Models that are generally useful for “mega-scale” (which moves)
  - They don’t survive (for long) within a physical machine…
- 64KB not enough? (early 1980s)
  - 20 bit segmented memory for 16 bit processors (birth of x86)
- 640KB not enough? (early 1990s)
  - 32 bit operating systems, even in the “commodity” world

Why ~2-4GB?
It’s all about GC (and only GC)

- Seems to be the practical limit for responsive applications
- A 100GB heap won’t crash. It just periodically “pauses” for several minutes at a time.
- [Virtually] All current commercial JVMs will exhibit a periodic multi-second pause on a normally utilized 2-5GB heap.
  - It’s a question of “When”, not “If”.
  - GC Tuning only moves the “when” and the “how often” around
- [Concurrent] Compaction seems to be “hard to do”…
- Without solving compaction, can’t seem to solve GC
- Without solving GC, can’t use >~$500 worth of H/W.

Garbage Collection: the new 640K

- Responsiveness
  - Compaction is inevitable
  - Existing Java runtimes perform compaction as stop-the-world
  - The inevitable pause times are linear to memory heap sizes
  - Delay games are the only current tuning strategy
- Scale:
  - Responsiveness requirements limit heap sizes
  - Limited heap sizes limit scale, sustainable throughput
  - CPU core use limited by heap
  - Throughput, Latency, and Scale all fighting each other
- Result:
  - Instance sprawl is the ONLY way to add or use capacity
  - 2011: It takes >30 2GB JVMs to fill up a $7K server….
  - This is getting embarrassing…

GC discussion

Pauses.

- How many of you have seen GC pauses:
  - Larger than ½ sec?
  - Larger than 1 sec?
  - Larger than 2 sec?
  - Larger than 4 sec?
  - Larger than 10 sec?
  - Larger than 20 sec?
  - Larger than 60 sec?

Some Basic Terminology: What is a concurrent collector?

A Concurrent Collector performs garbage collection work concurrently with the application’s own execution

A Parallel Collector uses multiple CPUs to perform garbage collection

Terminology

Useful terms for discussing concurrent collection

- Mutator
  - Your program…
- Parallel
  - Can use multiple CPUs
- Concurrent
  - Runs concurrently with program
- Pause time
  - Time during which mutator is not running any code
- Generational
  - Collects young objects and long lived objects separately.
- Promotion
  - Allocation into old generation
- Marking
  - Finding all live objects
- Sweeping
  - Locating the dead objects
- Compaction
  - Defragments heap
  - Moves objects in memory
  - Remaps all affected references
  - Frees contiguous memory regions

Metrics

Useful metrics for discussing concurrent collection

- Heap population (aka Live set)
  - How much of your heap is alive
- Allocation rate
  - How fast you allocate
- Mutation rate
  - How fast your program updates references in memory
- Heap Shape
  - The shape of the live object graph
  - * Hard to quantify as a metric...
- Object Lifetime
  - How long objects live
- Cycle time
  - How long it takes the collector to free up memory
- Marking time
  - How long it takes the collector to find all live objects
- Sweep time
  - How long it takes to locate dead objects
  - * Relevant for Mark-Sweep
- Compaction time
  - How long it takes to free up memory by relocating objects
  - * Relevant for Mark-Compact

The “hard” things to do in GC

- Robust concurrent marking
  - Refs keep changing
  - Multi-pass marking sensitive to mutation rate
  - Weak, Soft, Final references “hard” to deal with concurrently
- [Concurrent] Compaction…
  - It’s not the moving of the objects…
  - It’s the fixing of all those references that point to them
  - How do you deal with a mutator looking at a stale reference?
  - If you can’t, then remapping is a STW operation
- Without solving Compaction, GC won’t be solved
  - All current commercial server JVMs and GCs perform compaction
  - (Azul ships the only commercial JVMs that concurrently compact)

Garbage Collection & Compaction

eventually, ALL collector compact the heap

- Compaction is inevitable
  - And compacting anything requires scanning/fixing all references to it
  - Usually the worst possible thing that can happen in GC
- You can delay compaction, but not get rid of it
- Delay tactics focus on getting "easy empty space" first
  - This is the focus for the vast majority of GC tuning
- Most objects die young [Generational]
  - So collect young objects only, as much as possible
  - But eventually, some old dead objects must be reclaimed
- Most old dead space can be reclaimed without moving it [e.g. CMS]
  - So track dead space in lists, and reuse it in place
  - But eventually, space gets fragmented, and needs to be moved
- Much of the heap is not "popular" [e.g. G1]
  - A non popular region will only be pointed to from a small % of the heap
  - So compact non-popular regions in short stop-the-world pauses
  - But eventually, popular objects and regions need to be compacted

HotSpot™ CMS

Collector mechanism classification

- Stop-the-world compacting new gen (ParNew)
- Mostly Concurrent, non-compacting old gen (CMS)
  - Mostly Concurrent marking
    - Mark concurrently while mutator is running
    - Track mutations in card marks
    - Revisit mutated cards (repeat as needed)
    - Stop-the-world to catch up on mutations, ref processing, etc.
  - Concurrent Sweeping
  - Does not Compact (maintains free list, does not move objects)
- Fallback to Full Collection (Stop the world).
  - Used for Compaction, etc.

HotSpot “Garbage First” (aka G1)

Collector mechanism classification

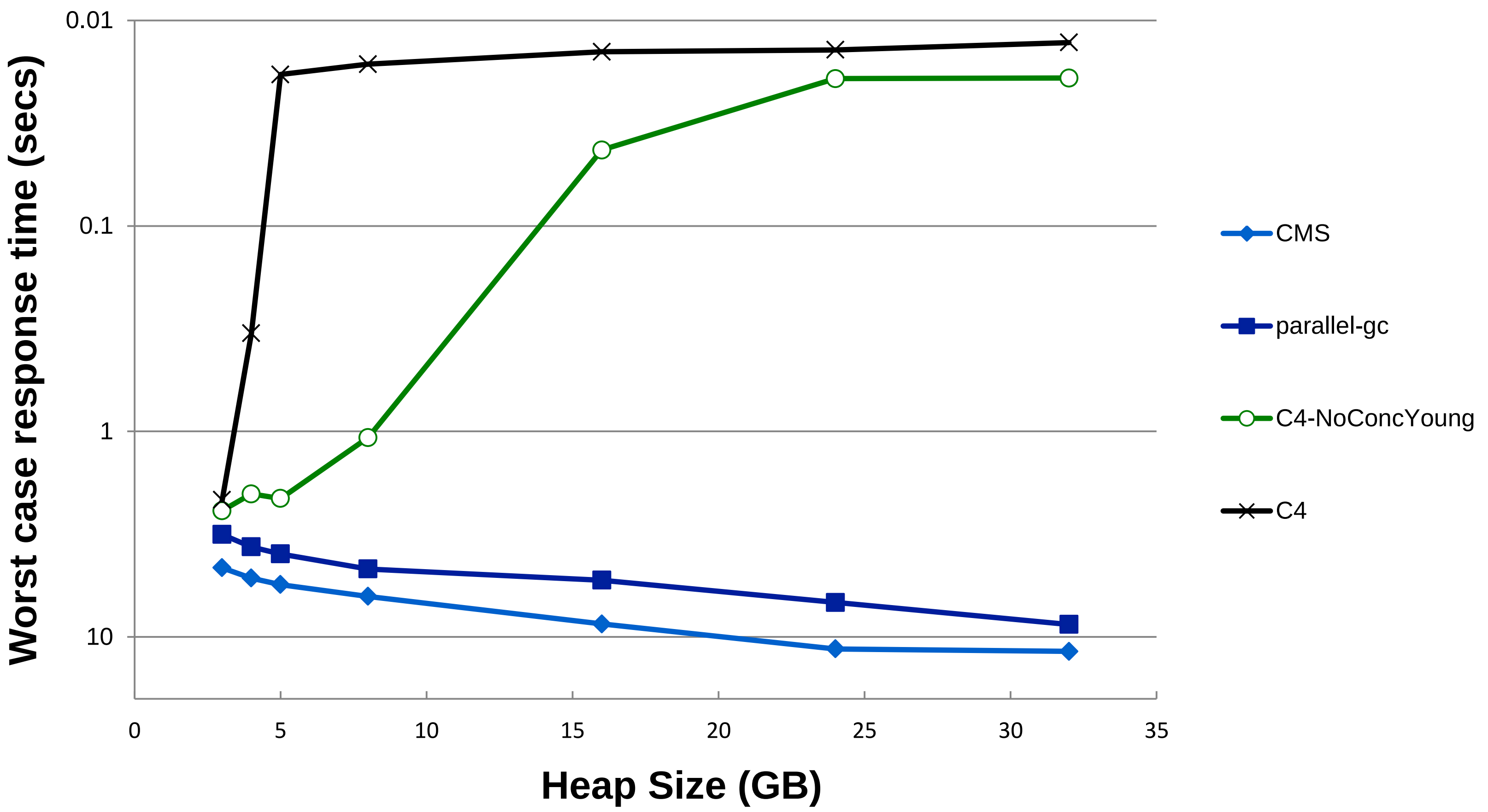
- Experimental
  - -XX:+UnlockExperimentalVMOptions -XX:+UseG1GC
- Stop-the-world compacting new gen
- Mostly Concurrent, old gen marker
  - Mostly Concurrent marking
  - Tracks inter-region relationships in remembered sets
- Stop-the-world incremental compacting old gen
  - Objective: “Avoid, *as much as possible*, having a Full GC…”
  - Compact sets of regions that can be scanned in limited time
  - Delay compaction of popular objects, popular regions
- Fallback to Full Collection (Stop the world)
  - Used for compacting popular objects, popular regions

Azul's "C4" Collector

Continuously Concurrent Compacting Collector

- Concurrent, compacting new generation
- Concurrent, compacting old generation
- Concurrent guaranteed-single-pass marker
  - Oblivious to mutation rate
  - Concurrent ref (weak, soft, final) processing
- Concurrent Compactor
  - Objects moved without stopping mutator
  - References remapped without stopping mutator
  - Can relocate entire generation (New, Old) in every GC cycle
- No stop-the-world fallback
  - Always compacts, and always does so concurrently
- It's elastic…

Sample responsiveness improvement



- SpecJBB + Slow churning 2GB LRU Cache
- Live set is ~2.5GB across all measurements
- Allocation rate is ~1.2GB/sec across all measurements

Java GC tuning is important…

Examples of actual command line GC tuning parameters:

```
Java -Xmx12g -XX:MaxPermSize=64M -XX:PermSize=32M -XX:MaxNewSize=2g
        -XX:NewSize=1g -XX:SurvivorRatio=128 -XX:+UseParNewGC
        -XX:+UseConcMarkSweepGC -XX:MaxTenuringThreshold=0
        -XX:CMSInitiatingOccupancyFraction=60 -XX:+CMSParallelRemarkEnabled
        -XX:+UseCMSInitiatingOccupancyOnly -XX:ParallelGCThreads=12
        -XX:LargePageSizeInBytes=256m …
```

```
Java –Xms8g –Xmx8g –Xmn2g -XX:PermSize=64M -XX:MaxPermSize=256M
        -XX:-OmitStackTraceInFastThrow -XX:SurvivorRatio=2 -XX:-UseAdaptiveSizePolicy
        -XX:+UseConcMarkSweepGC -XX:+CMSConcurrentMTEnabled
        -XX:+CMSParallelRemarkEnabled -XX:+CMSParallelSurvivorRemarkEnabled
        -XX:CMSMaxAbortablePrecleanTime=10000 -XX:+UseCMSInitiatingOccupancyOnly
        -XX:CMSInitiatingOccupancyFraction=63 -XX:+UseParNewGC –Xnoclassgc …
```

The complete guide to Zing GC tuning

```
java -Xmx40g
```

C4 highlights: a taste of the secret sauce

- A Loaded Value Barrier (LVB) is central to the algorithm
  - Every Java reference is verified as “sane” when loaded
  - “Non-sane” refs are fixed in a self-healing barrier
- Refs that have not yet been “marked through” are caught
  - Guaranteed single pass concurrent marker
- Refs that point to relocated objects are caught
  - Lazily (and concurrently) remap refs, no hurry
  - Relocation and remapping are both concurrent
- We use “quick release” to recycle memory
  - Forwarding information is kept outside of object pages
  - We release physical memory immediately upon relocation
  - “Hand-over-hand” compaction without requiring empty memory
- We use new virtual memory ops in an enhanced kernel…

A virtual world

Java & Virtualization

- How many of you use virtualization?
  i.e. VMWare, KVM, Xen, desktop virtualization (Fusion, Parallels, VirtualBox, etc.)

- How many of you use it for production applications?

- How many of you think that virtualization will make your application run faster?

The Virtualization Tax

- Virtualization is universally considered a “tax”
- The Focus is on measuring and reducing overhead
- Everyone hopes to get to
  “virtually the same as non-virtualized”
- Plenty of infrastructure benefits
- But what are the application benefits?

Problem:
Java Runtimes are already limited

Common Java Runtime Limitations

- Responsiveness
- Scale and complexity
- Rigid, non-elastic, and inefficient
- Sensitivity to load, fragility
- Poor production-time visibility

These are “pre-existing conditions”

Problem:
Virtualization only makes things worse

- Moving to virtualized environments:
  - Nobody expects applications to run faster or better
  - Best hope is that virtualization “won’t hurt too much”
- Common published virtualization best practices for Java:
  - Use one JVM process per Guest OS
  - Use the fewest cores you can get away with
  - Turn off ballooning, avoid elasticity
- Tier-1 and some Tier-2 applications avoid virtualization
  - No ***Application*** benefits expected
  - Application behavior considered more important than virtualization benefits to infrastructure
- But, what if ……..?

What if……

- What if virtualization actually made applications better?
- What if virtualization was the way to solve the pre-existing Java limitations?

The Zing Platform: Virtualization with ***Application*** Benefits

If you want to:…

- ***Improve*** response times:
- ***Increase*** Transaction rates:
- ***Increase*** Concurrent users:
- ***Forget*** about GC pauses:
- Eliminate daily restarts:
- Elastically grow during peaks:
- Elastically shrink when idle:
- Gain production visibility:

Zing Platform

On virtualized

commodity H/W

The unfair advantage of solving GC

A completely unfair comparison

Compare two JVMs

- Running the same application
- Running on same hardware
- See how far each goes before it breaks

Head to Head comparison

Same hardware, same application

- LifeRay Portal on JBoss
- Simple client load
  - ~10 sec. think times
  - ~40 MB temporary data generated per ~300ms transaction
  - ~20 MB session state
  - very slow churning LRU background load (@<20MB/sec)
- Sustainable SLA requirement:
  - Worst case < 10 sec.
  - 99.9% < 5 sec.
  - 90% < 1 sec.
  - Pushing pauses out of test window run not allowed.

Head to Head comparison

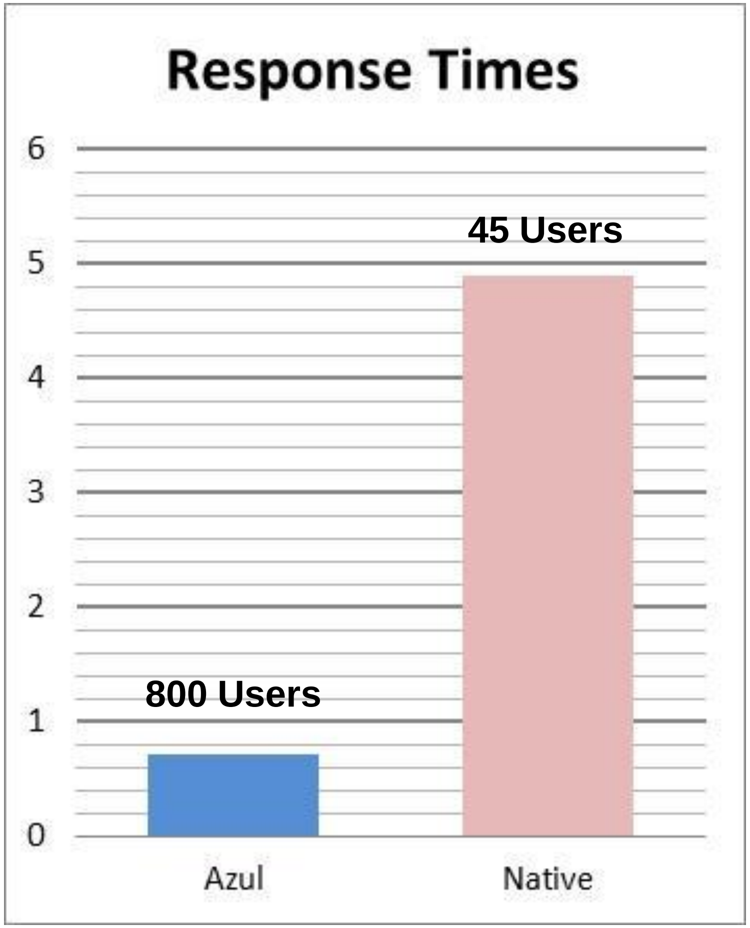
Same hardware, same application

- Hardware
  - 2x Intel Xeon 5620 (12 cores), 96GB
  - ~$6,500 as of March 2011… (~$1.75/GB/month)
- “Native” (aka “non-Virtualized”):
  - Fedora Core 12
  - Native HotSpot JVM
- Virtualized:
  - VMWare ESX 4.0
  - Zing Virtual Appliance
  - Fedora Core 12 (running as VMWare guest)
  - Zing JVM

Head to Head comparison

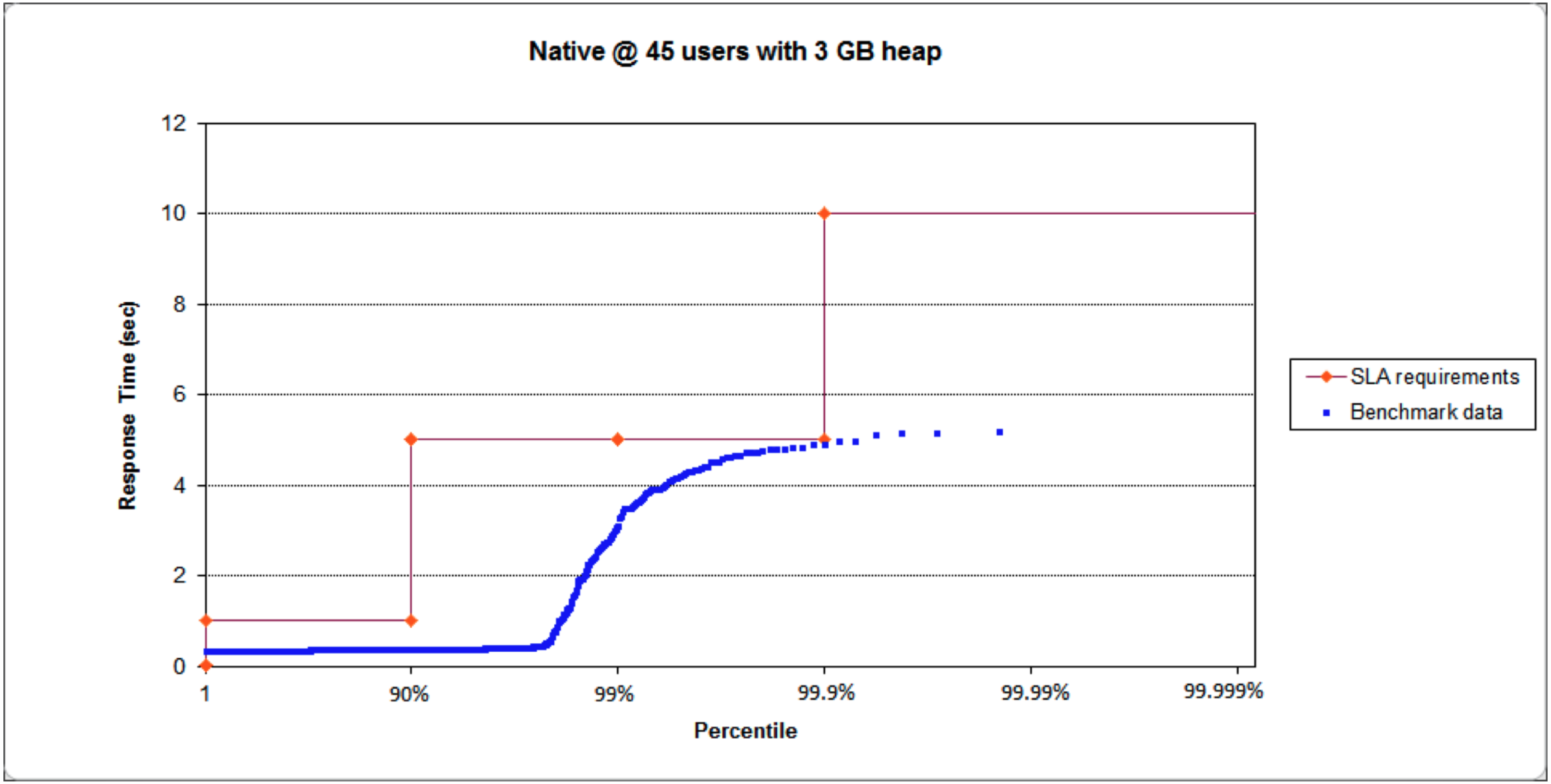
Same hardware, same application

- >17x more concurrent users
- >6x better response times



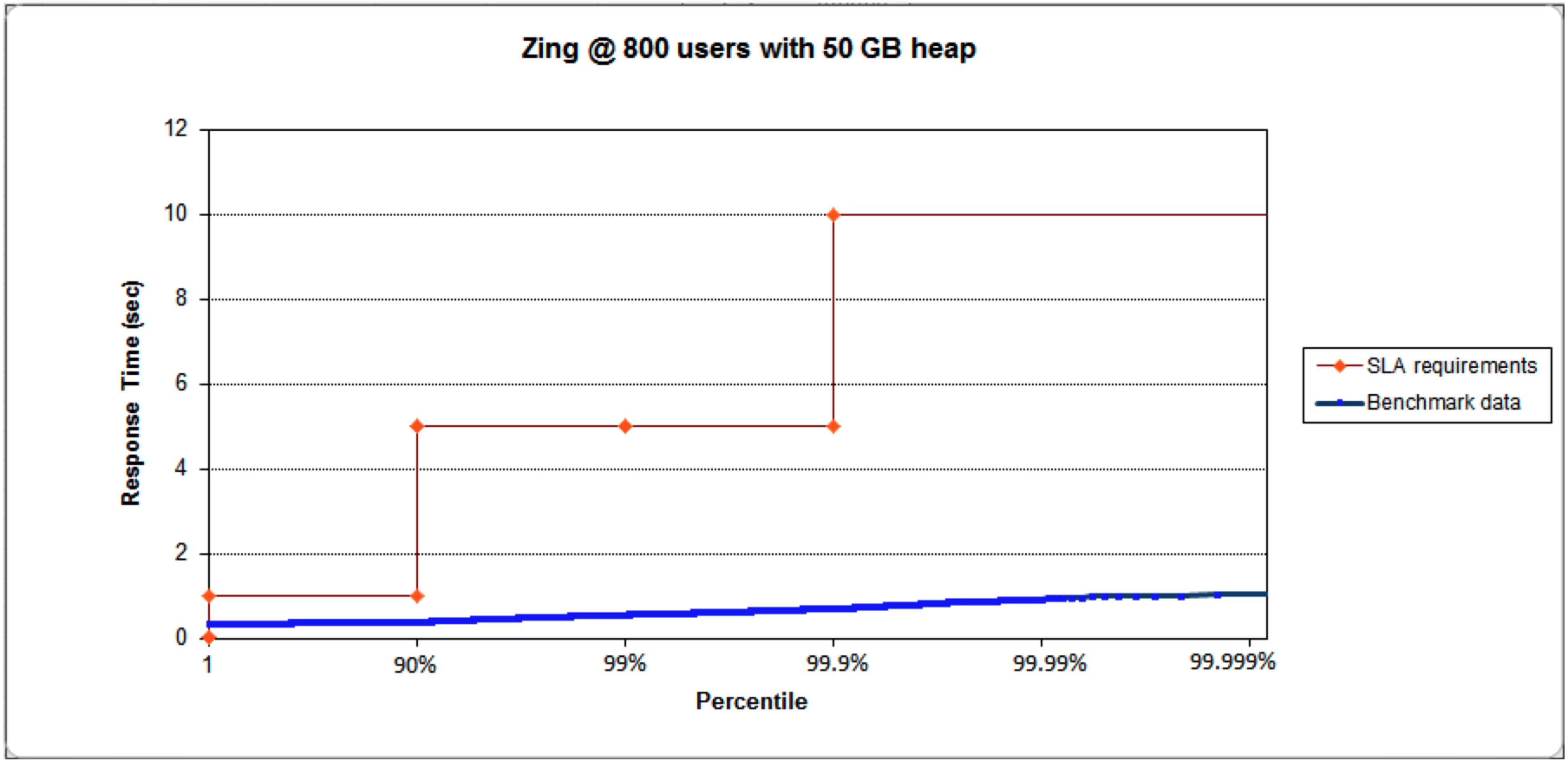
*** LifeRay portal on JBoss @ 99.9% SLA of 5 second response times**

Native JVM: Peaks at 45 concurrent users



*** LifeRay portal on JBoss @ 99.9% SLA of 5 second response times**

Zing JVM: smooth @ 800 concurrent users



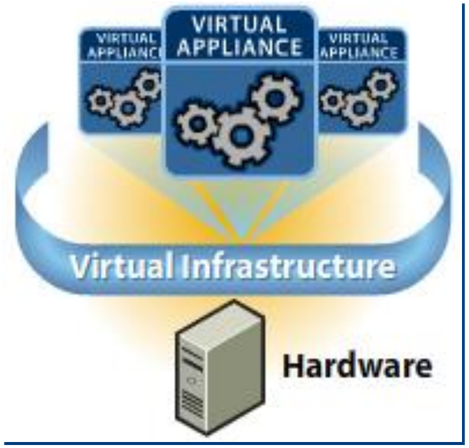
Zing Platform Components

Zing Java Virtual Machine

Virtualized Java Runtime

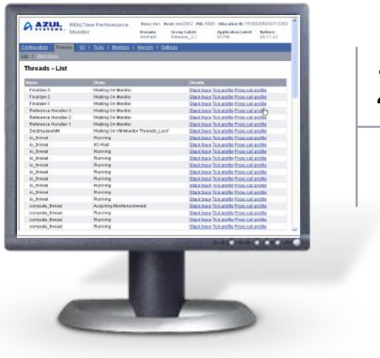
Zing Java Virtual Appliance

Java-Optimized Execution Environment



Zing Resource Controller

Centralized Monitoring & Mgmt

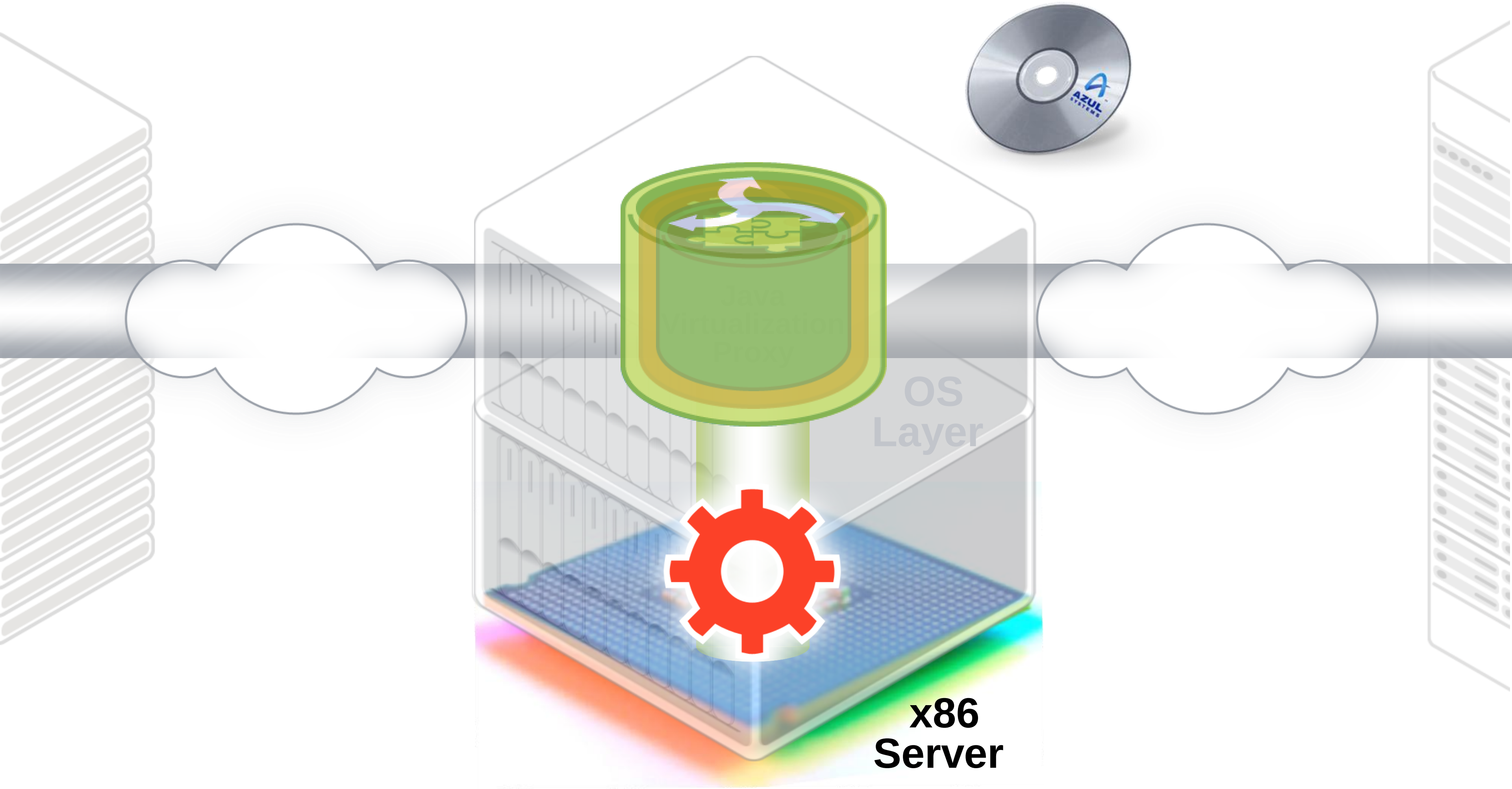


Zing Vision

Non-intrusive Visibility

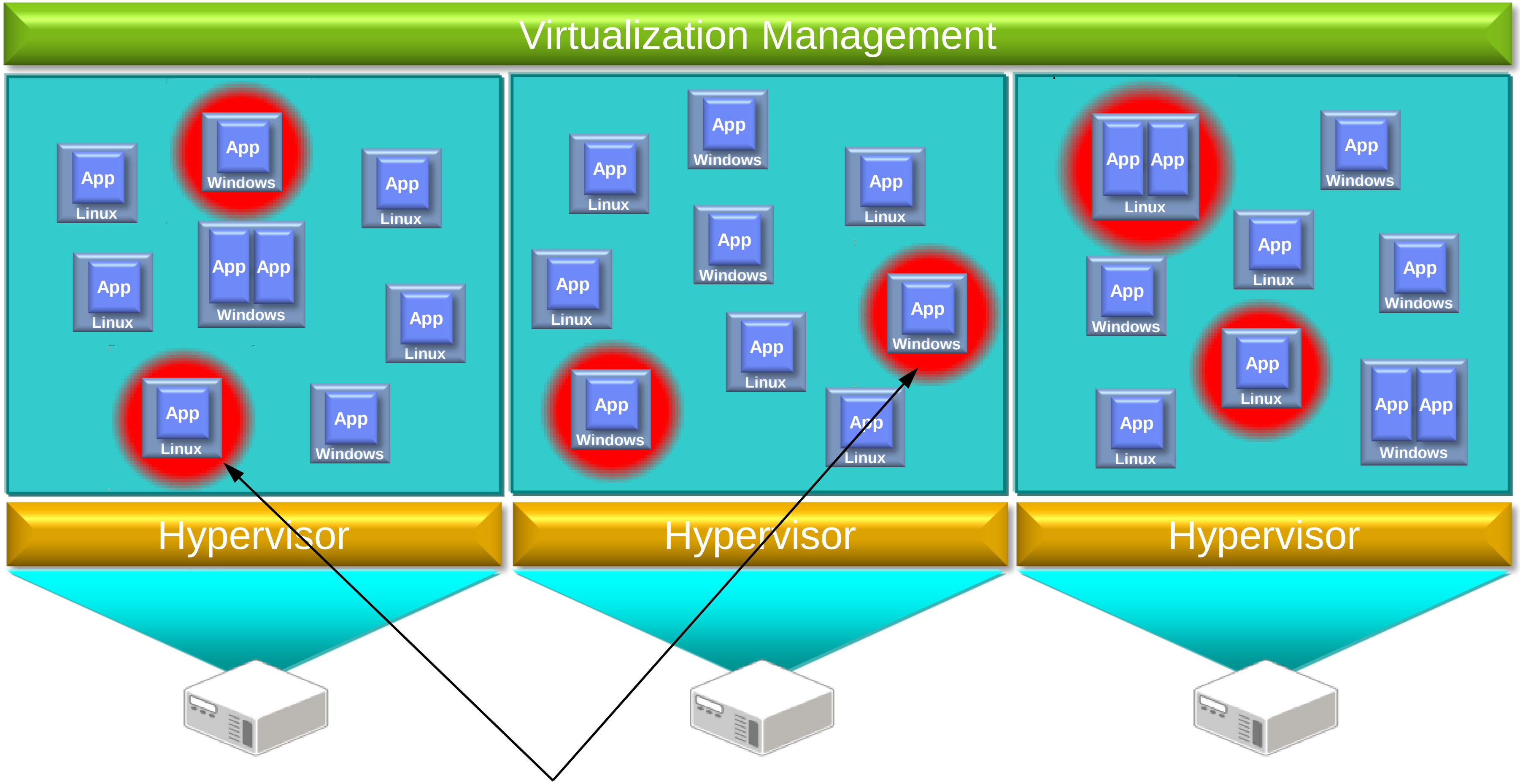
Zing - Java Runtime Virtualization

Liberating Java From the Rigidities of the OS



Current Virtualized Java Deployments

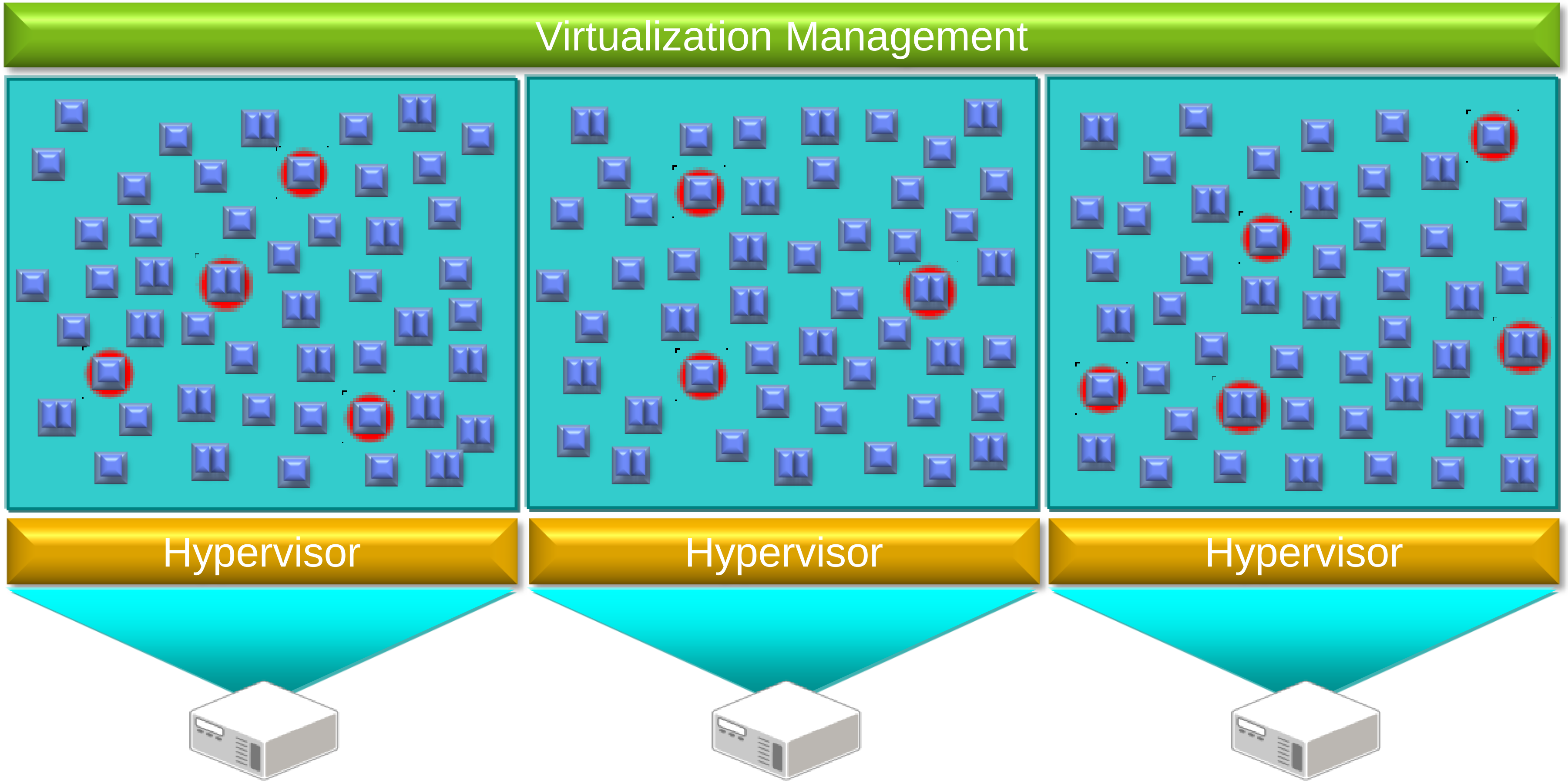
Limited scalability, many instances to manage, Inefficient use of resources



Today’s JVMs are each limited to ~3-4 GBytes of memory before response times become unacceptable, ***limiting application instance scalability, throughput & resiliency***

Current Virtualized Java Deployments

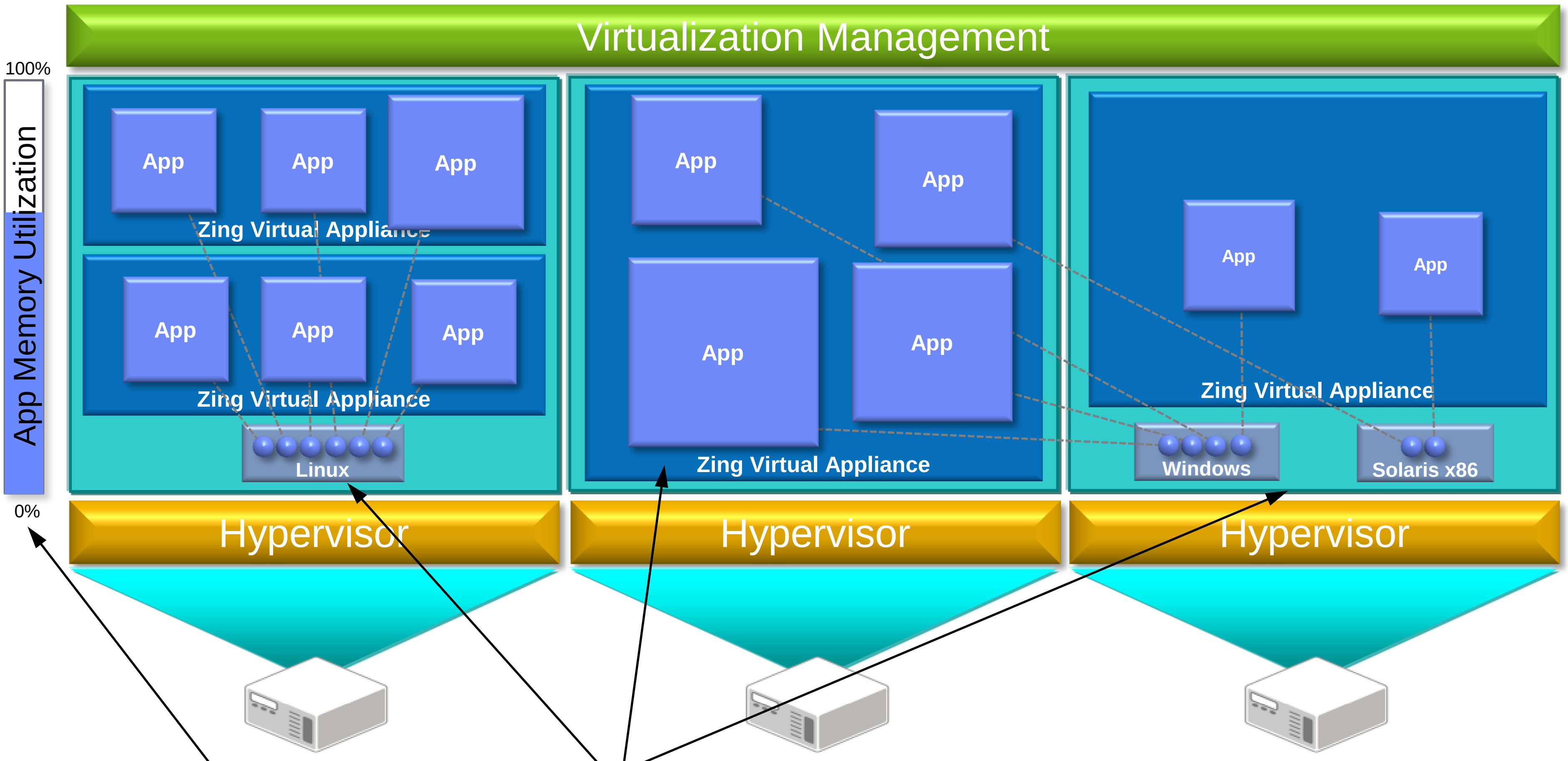
Limited scalability, Too many instances to manage, Inefficient use of resources



~50-100 OS and JVM instances are required to fully utilize a $10K-$20K commodity server.

A Better Way: Zing Elastic Deployments

Elastic app scalability, simplified deployments, efficient use of resources



[illegible] ***scalable applications, simplified deployments, and efficient use of resources***

Using virtualization to solve GC

New Opportunities for Application Innovation

- Better application metrics across the board
  - Responsiveness, capacity, stability, scalability, etc…
- Virtualized
  - Business critical applications can use virtualization to ***improve*** application metrics, not just save money on infrastructure
  - Catalyst for rapid adoption and transition to H/W virtualization
- Elastic
  - True shared headroom, for both memory and CPU
  - Dramatically improve stability, availability
  - Better resource utilization, more efficient
- Cloud-ready
  - Enables ***performant*** move to cloud deployment
  - Allows applications to leverage elastic cloud resources
  - Strong match for Multi-tenant and PAAS environments

Q & A

Gil Tene
CTO, Azul Systems