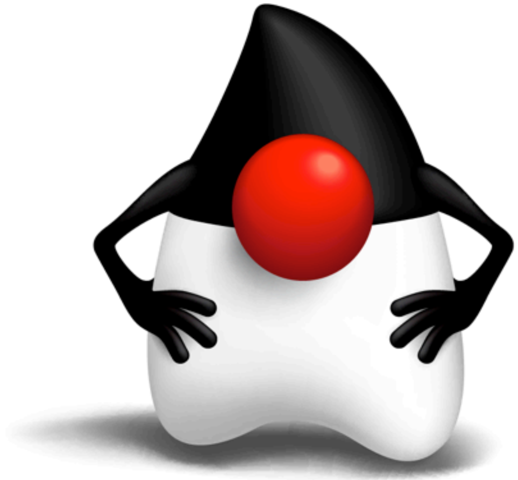
Java At Speed: Building a JVM For Modern Workloads

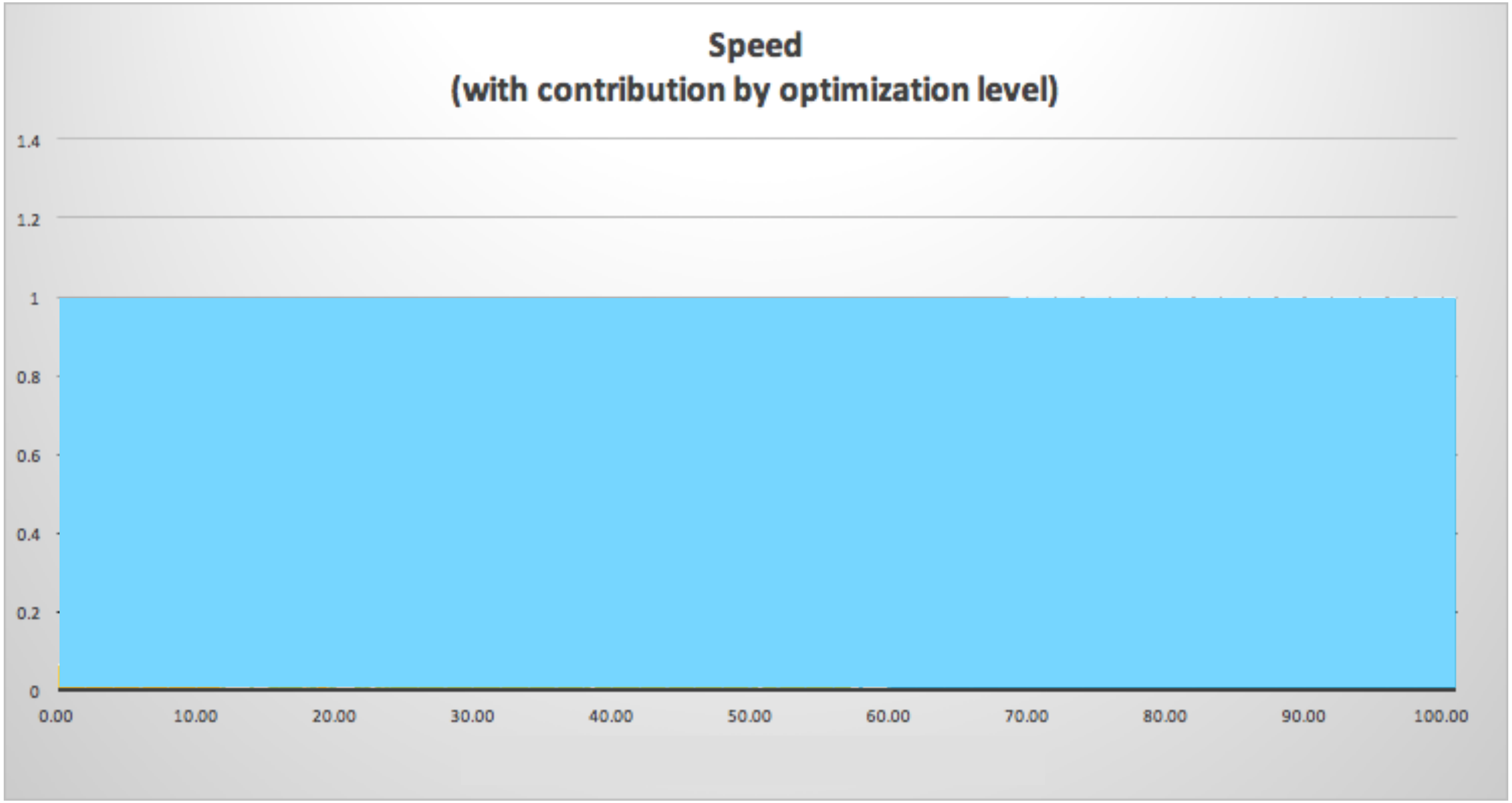
Simon Ritter
Deputy CTO, Azul Systems
azul.com

@speakjava

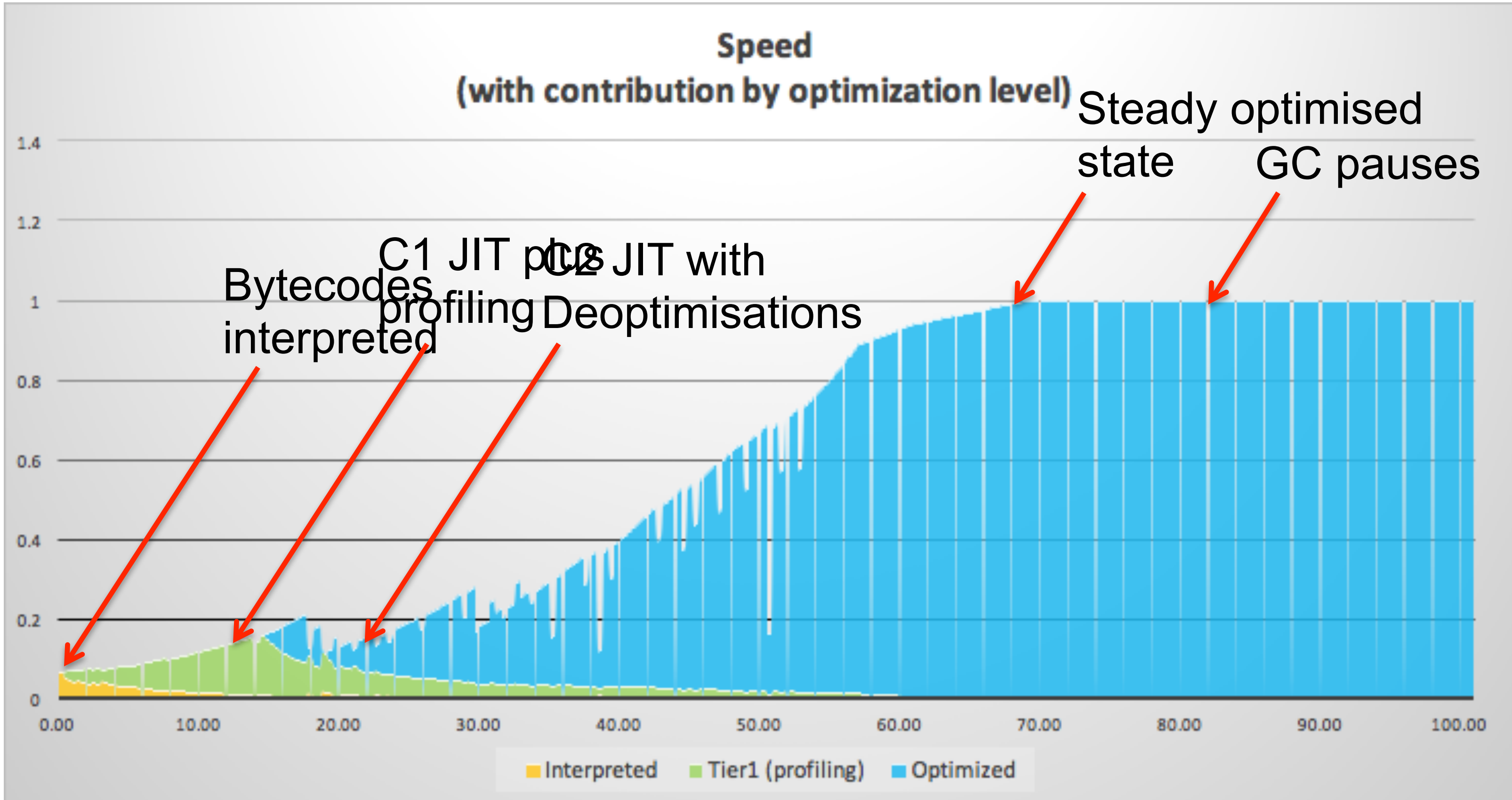
Speed In The Java World



JVM Performance Graph: Ideal



JVM Performance Graph: Reality

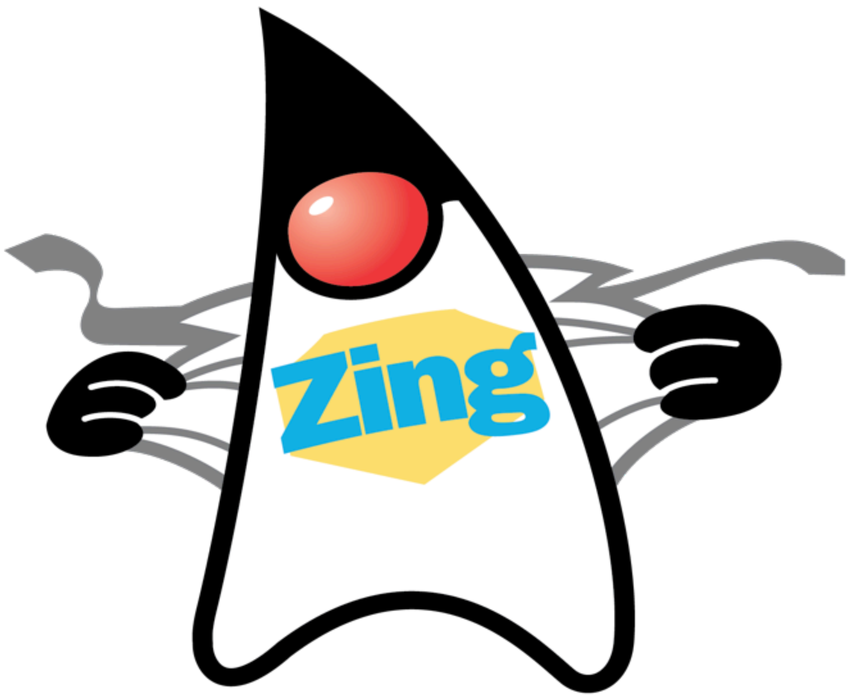


Big JVM Challenges

Managed runtime environment

1. The Garbage Collector
   – Inherently non-deterministic
   – Pause times can be big for most algorithms
2. Bytecodes, not machine code
   – Adaptive compilation strategies
   – Speed of code ‘warm-up’

What If There Was A Better JVM?



There is:

Zing

Azul Zing JVM

- Based on OpenJDK source code
- Passes all Java SE TCK/JCK tests
  - Drop in replacement for other JVMs
- Only one garbage collector: C4
  - Works in conjunction with Zing System Tools
- C2 JIT compiler replaced with Falcon
- ReadyNow! warm up elimination technology

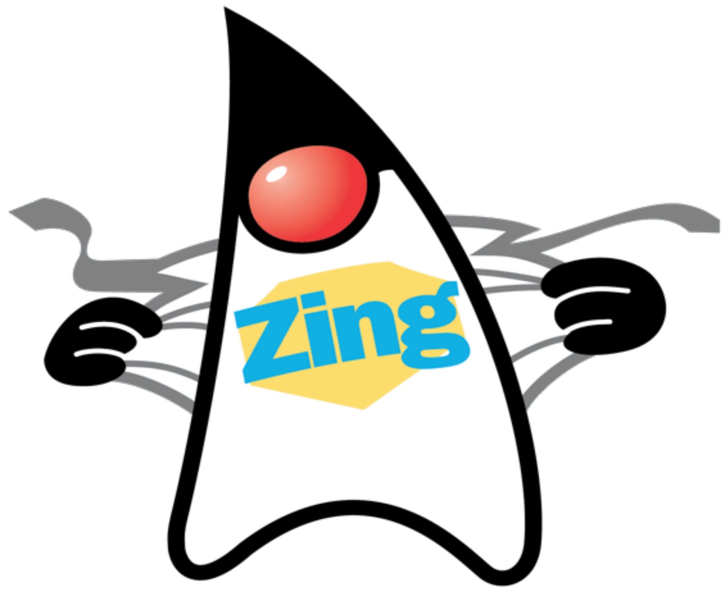
Zing System Tools

- Enables better memory management for JVM
- Allocation of new pages comes from ZST
  - ZST knows cache status
  - Newly allocated pages for TLAB are ‘hot’
  - Not like standard JVM
- Memory freed by JVM is returned to operating system
- Other clever tricks like Quick Release
- Only supported on Linux

Zing: Solving The GC Problem

- What Zing does NOT have
  - Serial, parallel, CMS or G1 collector
  - Full compacting old-generation fallback
  - Pause times proportional to heap size
- Zing: Continuous Concurrent Compacting Collector (C4)
  - Uses read-barrier rather than write-barrier
  - No stop-the-world pauses
  - The bigger the heap, the better the results

Azul Continuous Concurrent Compacting Collector (C4)



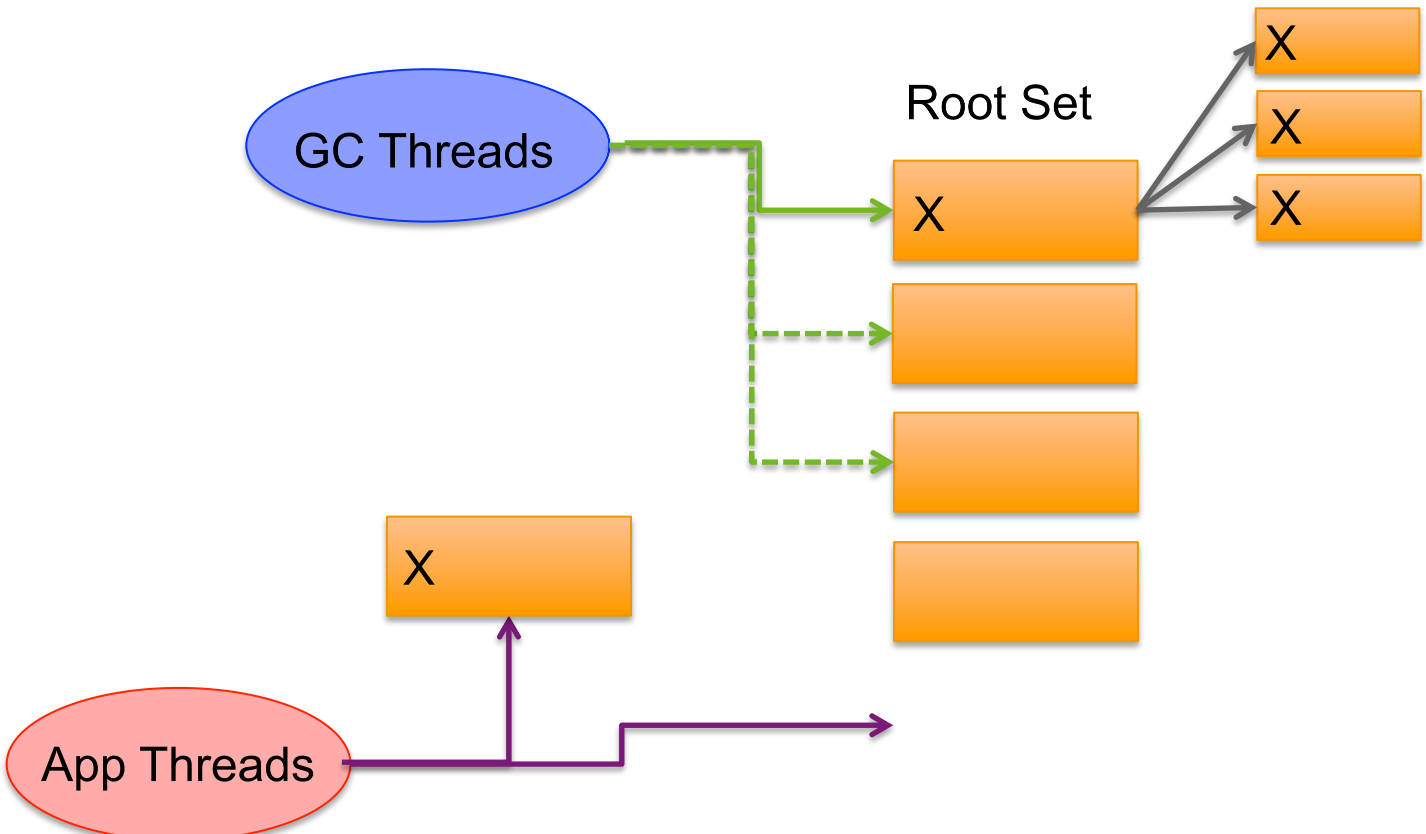
C4 Basics

- Generational (young and old)
  - Uses the same GC collector for both
  - For efficiency rather than pause containment
    - Weak generational hypothesis makes sense
- Concurrent, parallel and compacting
  - But no STW compacting fallback
- Algorithm is mark, relocate, remap

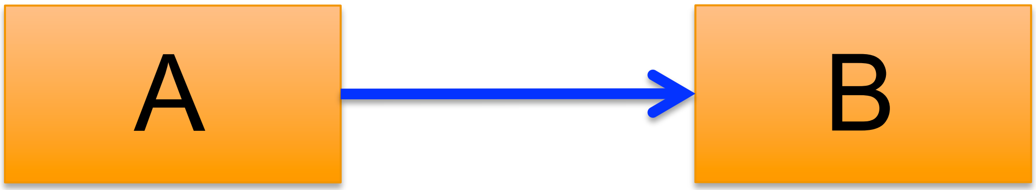
Loaded Value Barrier

- Read barrier
  - Tests all object references as they are loaded
- Enforces two invariants
  - Reference is marked through
  - Reference points to correct object position
- Allows for concurrent marking and relocation
- Minimal performance overhead
  - Test and jump (2 instructions)
  - x86 architecture reduces this to one micro-op

Concurrent Mark Phase

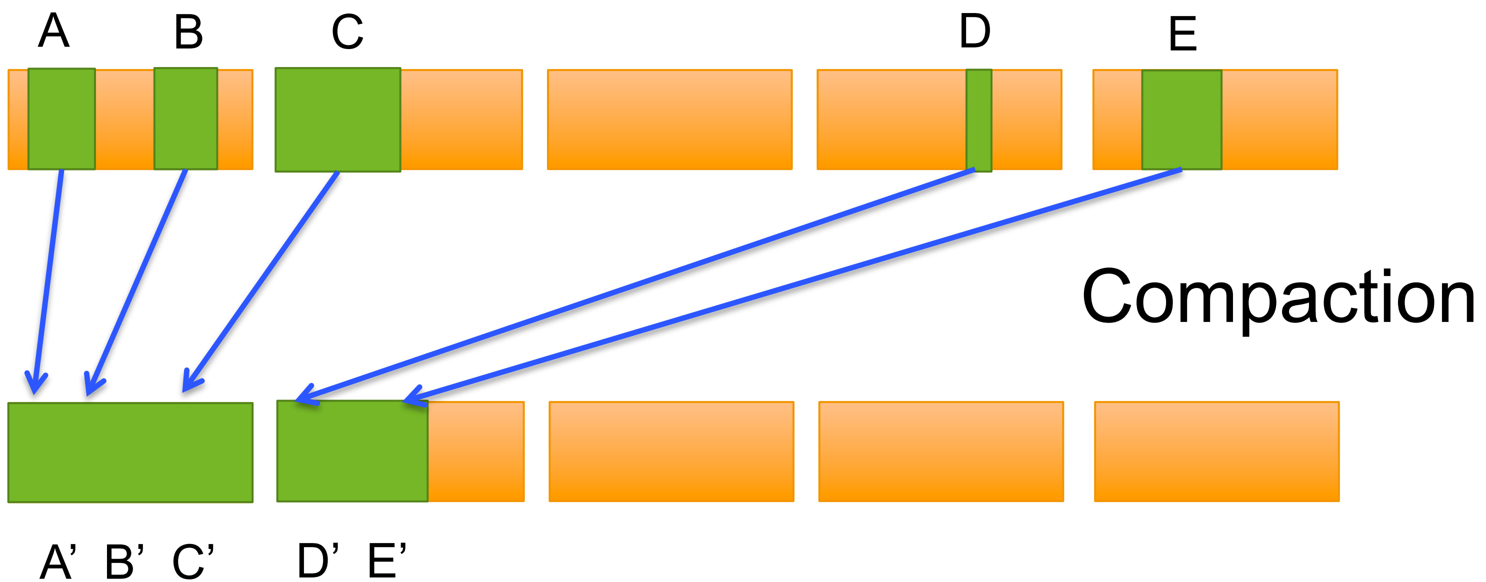


Loaded Value Barrier: Marking



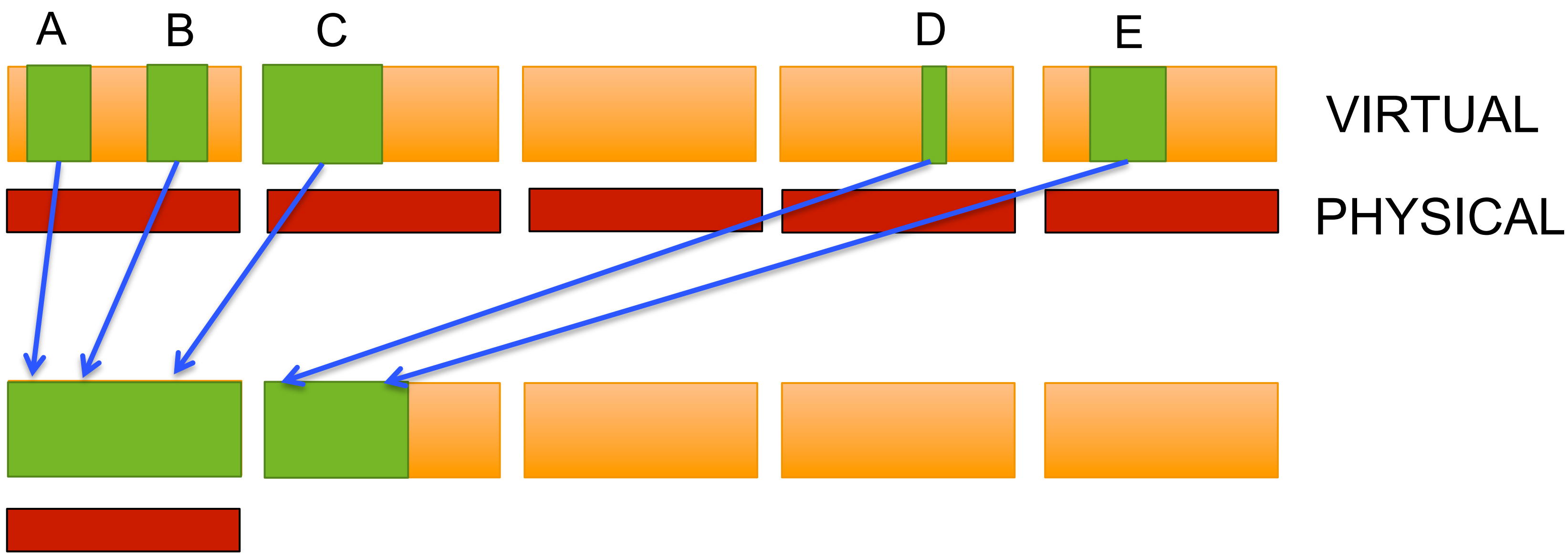

```
if (phase == MARKING && B not marked) {
    mark(B);
    addGCRootSet(B);
}

return B;
```

Relocation Phase



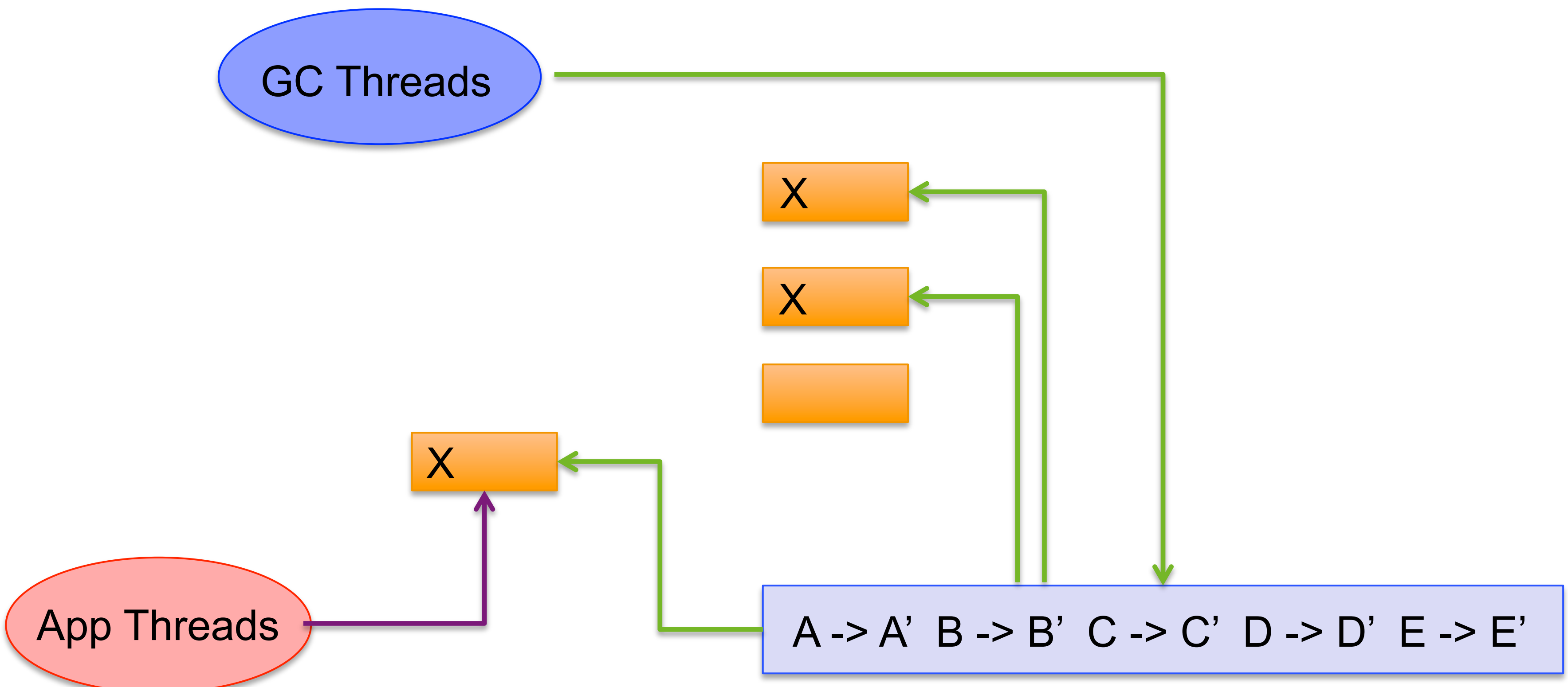
A -> A' B -> B' C -> C' D -> D' E -> E'

Quick Release

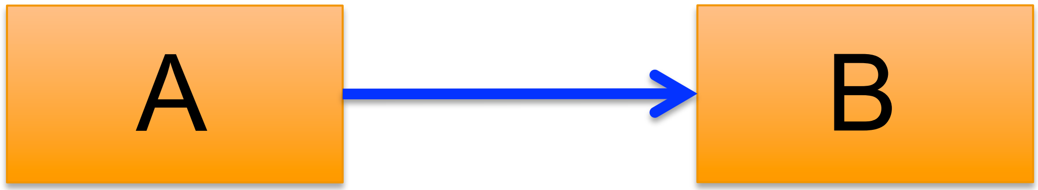


A -> A' B -> B' C -> C' D -> D' E -> E'

Remapping Phase



Loaded Value Barrier: Remapping

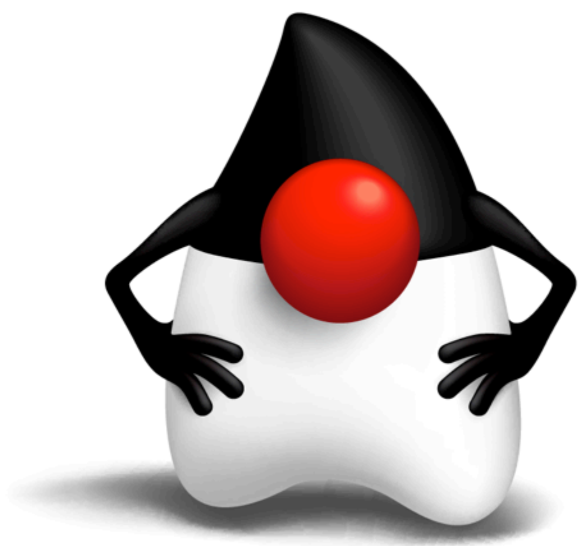



```
if (phase == REMAP && checkSideTable(B)) {
  remap(B);
  removeSideTable(B)
}

return B;
```

Zing: Big Heaps, No Problem

- Scales to 8Tb heap
  – No degradation in pause times
- Use one big heap, rather than many small heaps
  – Less JVMs means more efficiency
- Zing does not require big heaps
  – But works well with them

GC Tuning



Non-Zing GC Tuning Options

GC Tuning Used To Be Hard

```
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 …
```

GC Tuning Used To Be Hard

```
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 …
```

GC Tuning With Zing

java -Xmx1g

java -Xmx10g

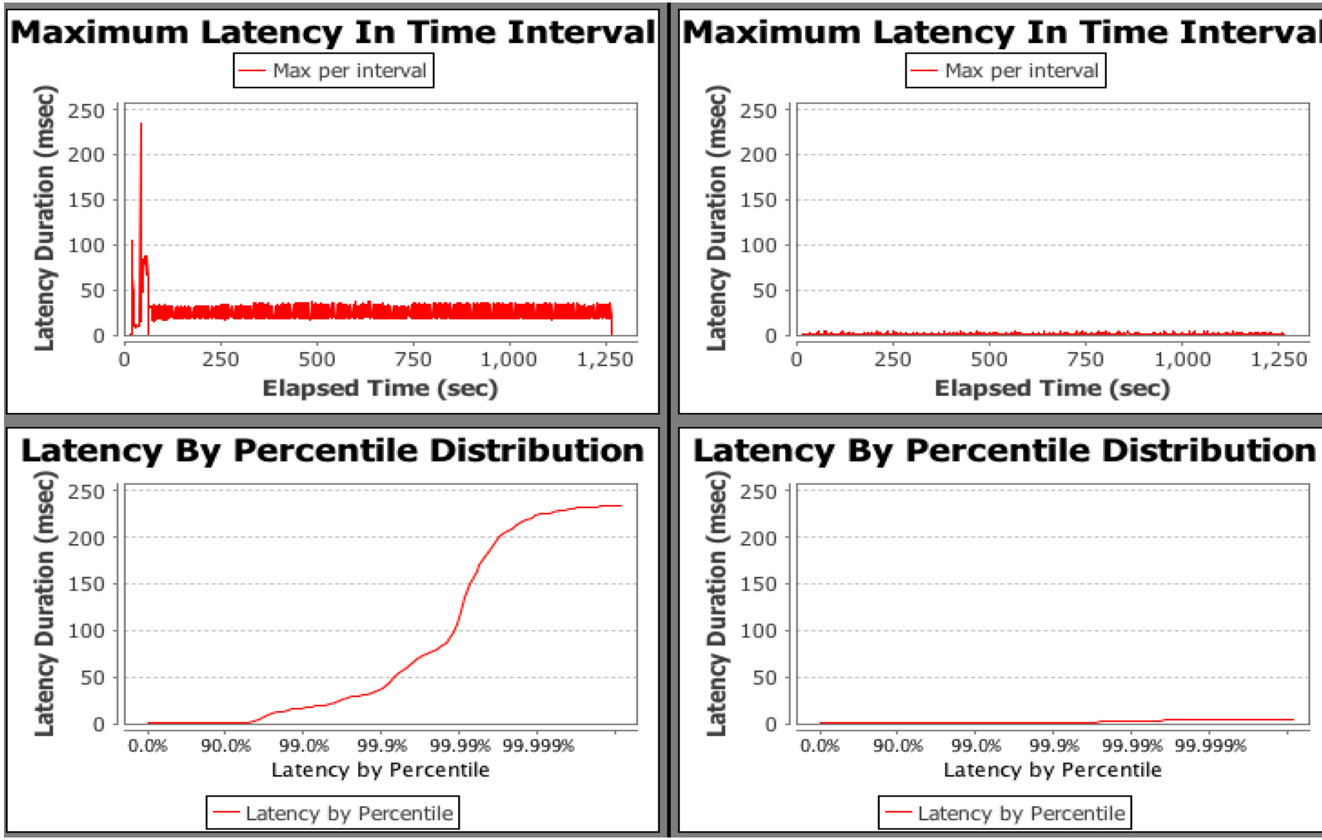
java -Xmx100g

java -Xmx8t

Measuring Platform Performance

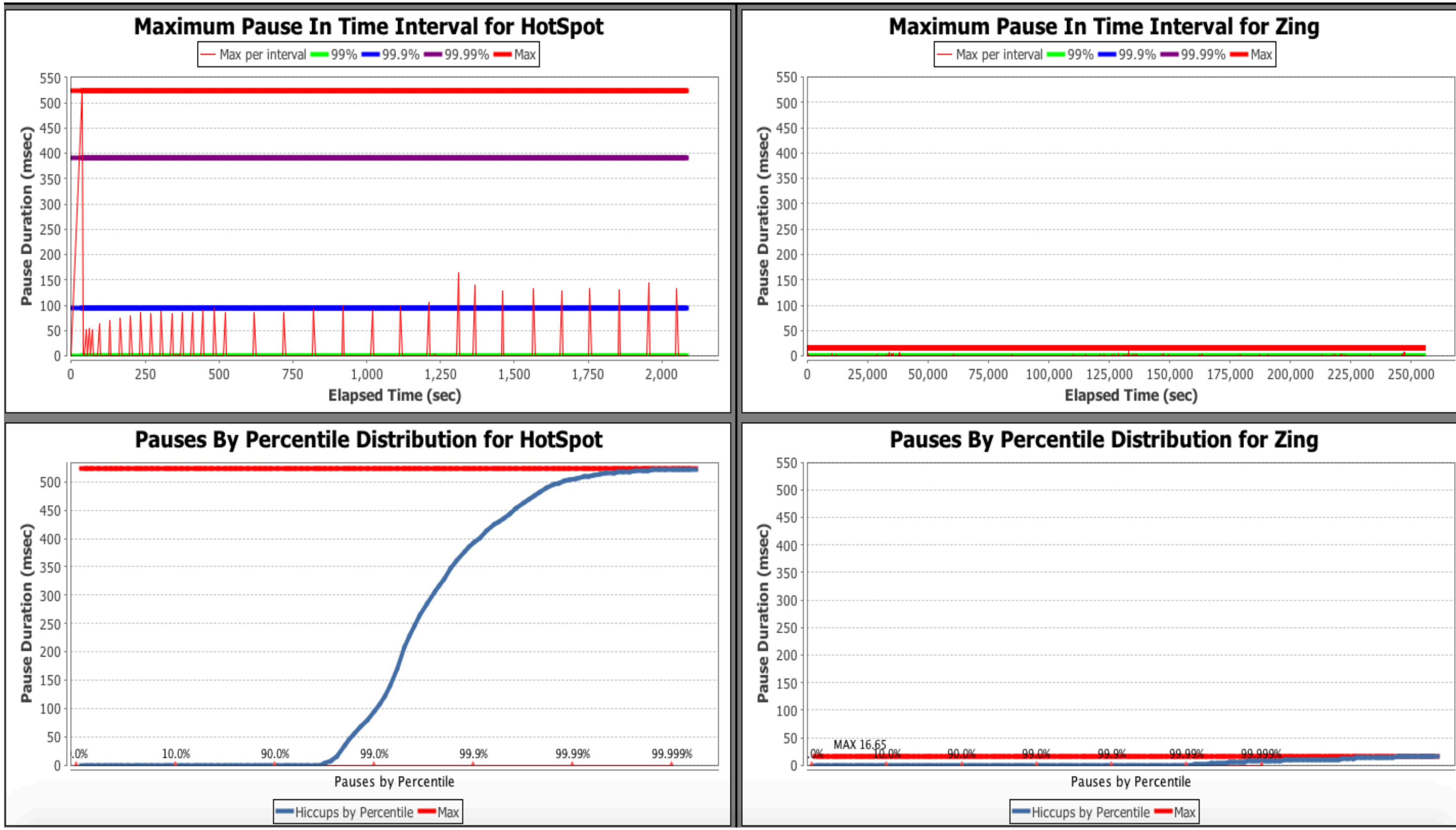
- jHiccup
- Spends most of its time asleep
  - Minimal effect on perfomance
  - Wakes every 1 ms
  - Records delta of time it expects to wake up
  - Measured effect is what would be experienced by your application
- Generates histogram log files
  - These can be graphed for easy evaluation

Small Heap, Small Latency



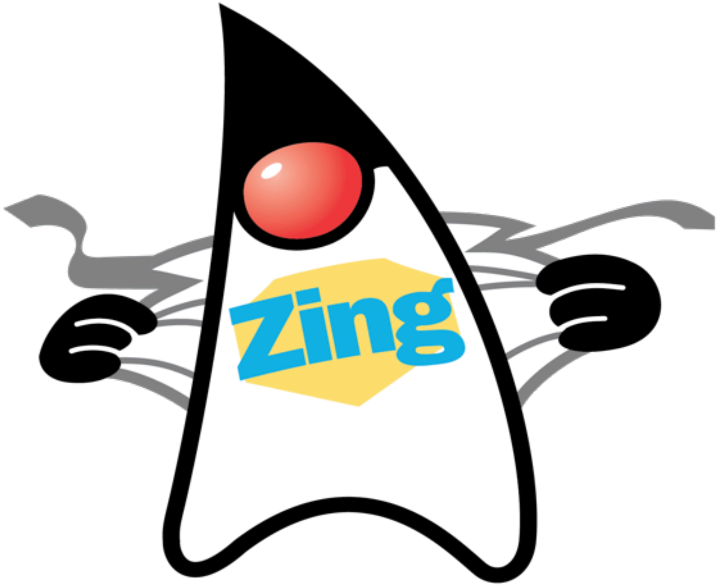
Hazelcast 2-node system with 1Gb heap Hotspot v. Zing

Big Heap, Small Latency



Cassandra with 60Gb heap Hotspot v. Zing

Azul Falcon JIT Compiler



AOT And JIT

- Ahead of Time (Static) compilation
  - Restricted optimisations due to runtime changes
    - Classloading
- Just in Time (Adaptive) compilation
  - Complete picture of loaded classes
  - More aggressive optimisations
    - Method inlining
    - Escape analysis

Speculative Optimisations

- JIT compilers can do things that static compilers are not able to do
  - However, these can sometimes affect performance
- Speculative optimisations
  - Optimise code based on what has happened so far
    - Likely to continue
    - But may not...
  - If speculation is false compiled code is thrown away
    - Revert to interpreted mode
    - Recompile if necessary

Untaken Path Example

```
int computeMagnitude(int val) {
  if (val >= 10)
    bias = computeBias(val);
  else
    bias = 1;

  return Math.log10(bias + 99);
}
```

Profiling shows val always less than 10 (so far)

```
int computeMagnitude(int val) {
  if (val >= 10)
    uncommonTrap();

  return 2;
}
```

Speculative optimised code compiled as if it was this

Implicit Null Check

- All field and array references are implicitly null checked

```
x = foo.y;
```

is really compiled as

```
if (foo == null)
  throw new NullPointerException();
x = foo.y;
```

- Compiler can hope for non-nulls and handle the SEGV separately
  – Faster code path

Adaptive Compilation Challenges

- Traditionally three options for running bytecodes
  - Fully interpreted
  - C1 (Client compiler):
    - Fast warmup, lower optimal level
  - C2 (Server compiler):
    - Slower warmup, higher optimal level
- Application takes time from starting to optimal level of performance

Advancing Adaptive Compilation

- Azul Falcon JVM compiler
  - Based on latest compiler research
  - LLVM project
- Better performance
  - Better intrinsics
  - More inlining
  - Fewer compiler excludes
- Targeted as replacement for C2 compiler

Simple Code Example

- Simple array summing loop
  - A modern compiler will use vector operations for this

```
private int sumLoop(int[] a) {
    int sum = 0;
    for (int i = 0; i < a.length; i++) {
        sum += a[i];
    }
    return sum;
}
```

More Complex Code Example

- Conditional array cell addition loop
  - Hard for compiler to identify for vector instruction use

```
private void addArraysIfEven(int a[], int b[]) {
    if (a.length != b.length) {
        throw new RuntimeException("length mismatch");
    }
    for (int i = 0; i < a.length; i++) {
        if ((b[i] & 0x1) == 0) {
            a[i] += b[i];
        }
    }
}
```

Traditional JVM JIT

Per element jumps
2 elements per iteration

```java
private void addArraysIfEven(int a[], int b[]) {
    if (a.length != b.length) {
        throw new RuntimeException("length mismatch");
    }
    for (int i = 0; i < a.length; i++) {
        if ((b[i] & 0x1) == 0) {
            a[i] += b[i];
        }
    }
}
```

		0x3001067f	addl %ecx, 12(%rsi)	0x014e0c
		0x30010682	movl $1, %edi	0xbf01000000
		0x30010687	cmpl $1, %eax	0x83f801
		0x3001068a	je 56 ; ABS: 0x300106c4	0x7438
		0x3001068c	subq %rdi, %rax	0x4829f8
		0x3001068f	leaq 16(%rdx,%rdi,4), %rcx	0x488d4cba10
		0x30010694	leaq 16(%rsi,%rdi,4), %rdx	0x488d54be10
		0x30010699	nopl (%rax)	0x0f1f8000000000
16.84%	1,286	0x300106a0	movl -4(%rcx), %esi	0x8b71fc
6.10%	466	0x300106a3	testb $1, %sil	0x40f6c601
		0x300106a7	jne 3 ; ABS: 0x300106ac	0x7503
7.61%	581	0x300106a9	addl %esi, -4(%rdx)	0x0172fc
29.41%	2,246	0x300106ac	movl (%rcx), %esi	0x8b31
2.25%	172	0x300106ae	testb $1, %sil	0x40f6c601
		0x300106b2	jne 2 ; ABS: 0x300106b6	0x7502
8.00%	611	0x300106b4	addl %esi, (%rdx)	0x0132
29.73%	2,271	0x300106b6	addq $8, %rcx	0x4883c108
		0x300106ba	addq $8, %rdx	0x4883c208
		0x300106be	addq $-2, %rax	0x4883c0fe
		0x300106c2	jne -36 ; ABS: 0x300106a0	0x75dc
0.03%	2	0x300106c4	addq $24, %rsp	0x4883c418
0.03%	2	0x300106c8	retq	0xc3
		0x300106c9	movq %rsi, 16(%rsp)	0x4889742410
		0x300106ce	movq %rdx, 8(%rsp)	0x4889542408
		0x300106d3	movabsq $805334400, %rax	0x48b8806d003000000000
		0x300106dd	callq *%rax	0xffd0

Falcon JIT

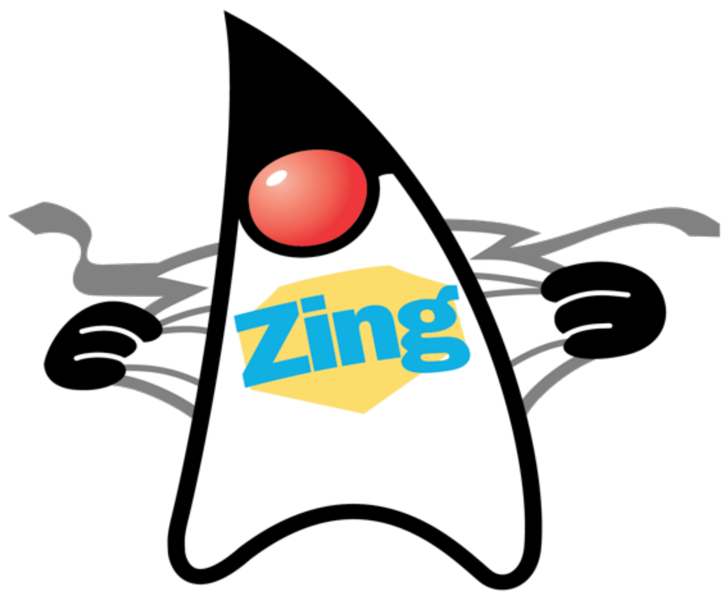
Using AVX2 vector instructions
32 elements per iteration

```java
private void addArraysIfEven(int a[], int b[]) {
    if (a.length != b.length) {
        throw new RuntimeException("length mismatch");
    }
    for (int i = 0; i < a.length; i++) {
        if ((b[i] & 0x1) == 0) {
            a[i] += b[i];
        }
    }
}
```

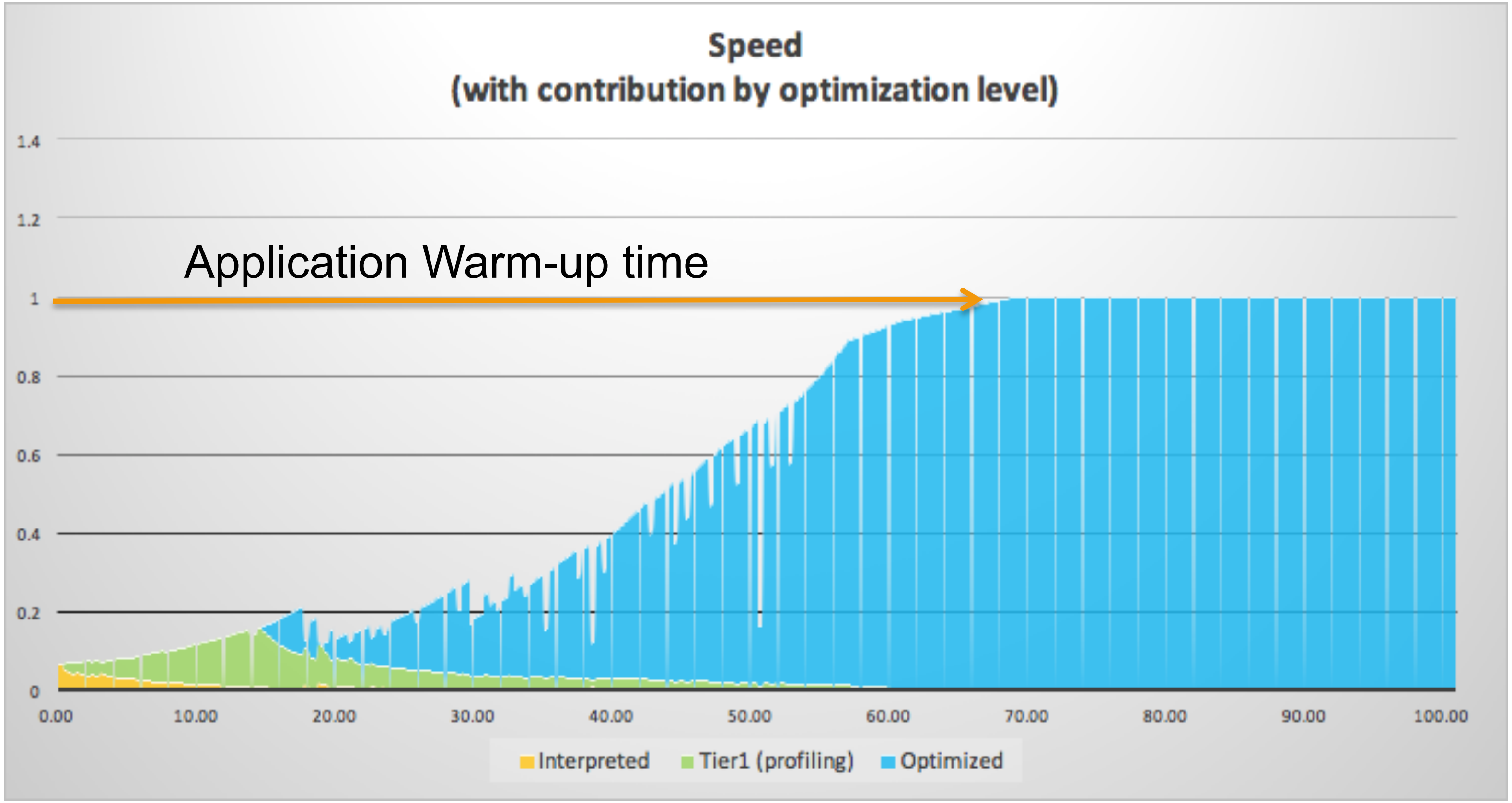
		0x3001455b	movq %rdi, %rbx
		0x3001455e	nop
0.15%	4	0x30014560	vmovdqu -96(%r11), %ymm2
12.31%	320	0x30014566	vmovdqu -64(%r11), %ymm3
0.50%	13	0x3001456c	vmovdqu -32(%r11), %ymm4
2.04%	53	0x30014572	vmovdqu (%r11), %ymm5
0.31%	8	0x30014577	vpand %ymm0, %ymm2, %ymm6
4.54%	118	0x3001457b	vpand %ymm0, %ymm3, %ymm7
0.69%	18	0x3001457f	vpand %ymm0, %ymm4, %ymm8
1.35%	35	0x30014583	vpand %ymm0, %ymm5, %ymm9
0.42%	11	0x30014587	vpcmpeqd %ymm1, %ymm6, %ymm6
2.58%	67	0x3001458b	vpmaskmovd -96(%rcx), %ymm6, %ymm10
3.58%	93	0x30014591	vpcmpeqd %ymm1, %ymm7, %ymm7
2.12%	55	0x30014595	vpmaskmovd -64(%rcx), %ymm7, %ymm11
12.12%	315	0x3001459b	vpcmpeqd %ymm1, %ymm8, %ymm8
1.50%	39	0x3001459f	vpmaskmovd -32(%rcx), %ymm8, %ymm12
3.69%	96	0x300145a5	vpcmpeqd %ymm1, %ymm9, %ymm9
1.81%	47	0x300145a9	vpmaskmovd (%rcx), %ymm9, %ymm13
12.27%	319	0x300145ae	vpaddd %ymm2, %ymm10, %ymm2
0.58%	15	0x300145b2	vpaddd %ymm3, %ymm11, %ymm3
0.19%	5	0x300145b6	vpaddd %ymm4, %ymm12, %ymm4
0.58%	15	0x300145ba	vpaddd %ymm5, %ymm13, %ymm5
3.27%	85	0x300145be	vpmaskmovd %ymm2, %ymm6, -96(%rcx)
7.15%	186	0x300145c4	vpmaskmovd %ymm3, %ymm7, -64(%rcx)
13.65%	355	0x300145ca	vpmaskmovd %ymm4, %ymm8, -32(%rcx)
4.58%	119	0x300145d0	vpmaskmovd %ymm5, %ymm9, (%rcx)
6.81%	177	0x300145d5	subq $-128, %r11
0.69%	18	0x300145d9	subq $-128, %rcx
0.31%	8	0x300145dd	addq $-32, %rbx
		0x300145e1	jne -135 ; ABS: 0x30014560
		0x300145e7	testl %r9d, %r9d
		0x300145ea	jne -356 ; ABS: 0x3001448c

Broadwell E5-2690-v4

ReadyNow!



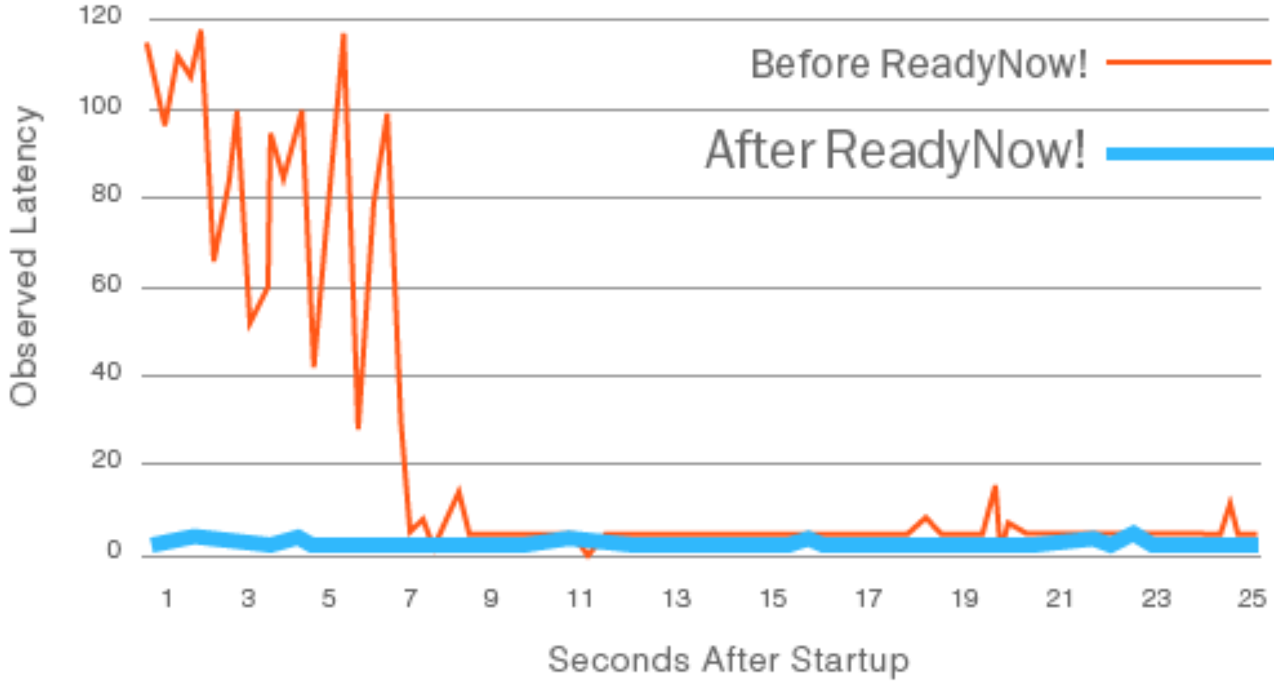
Traditional JVM



ReadyNow! Solution

- Save JVM JIT profiling information
  - Classes loaded
  - Classes initialised
  - Instruction profiling data
  - Speculative optimisation failure data
- Data can be gathered over much longer period
  - JVM/JIT profiles quickly
  - Significant reduction in deoptimisations
- Able to load, initialise and compile most code before `main()`

Effect Of ReadyNow!

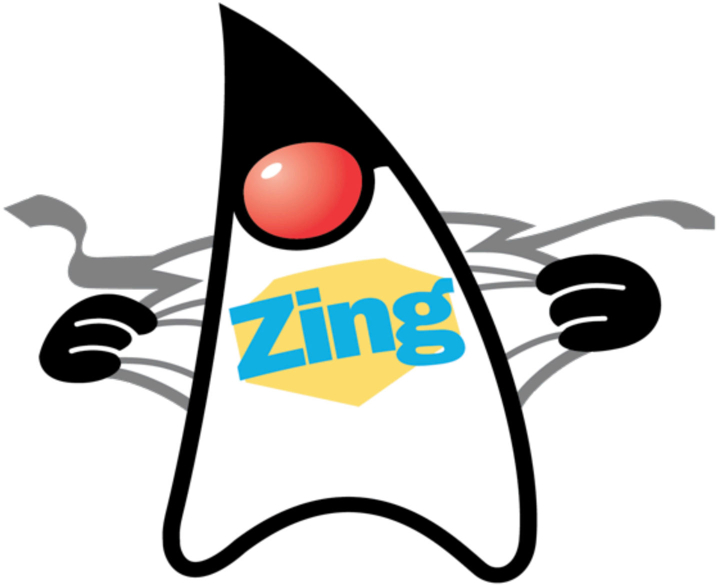


Customer application

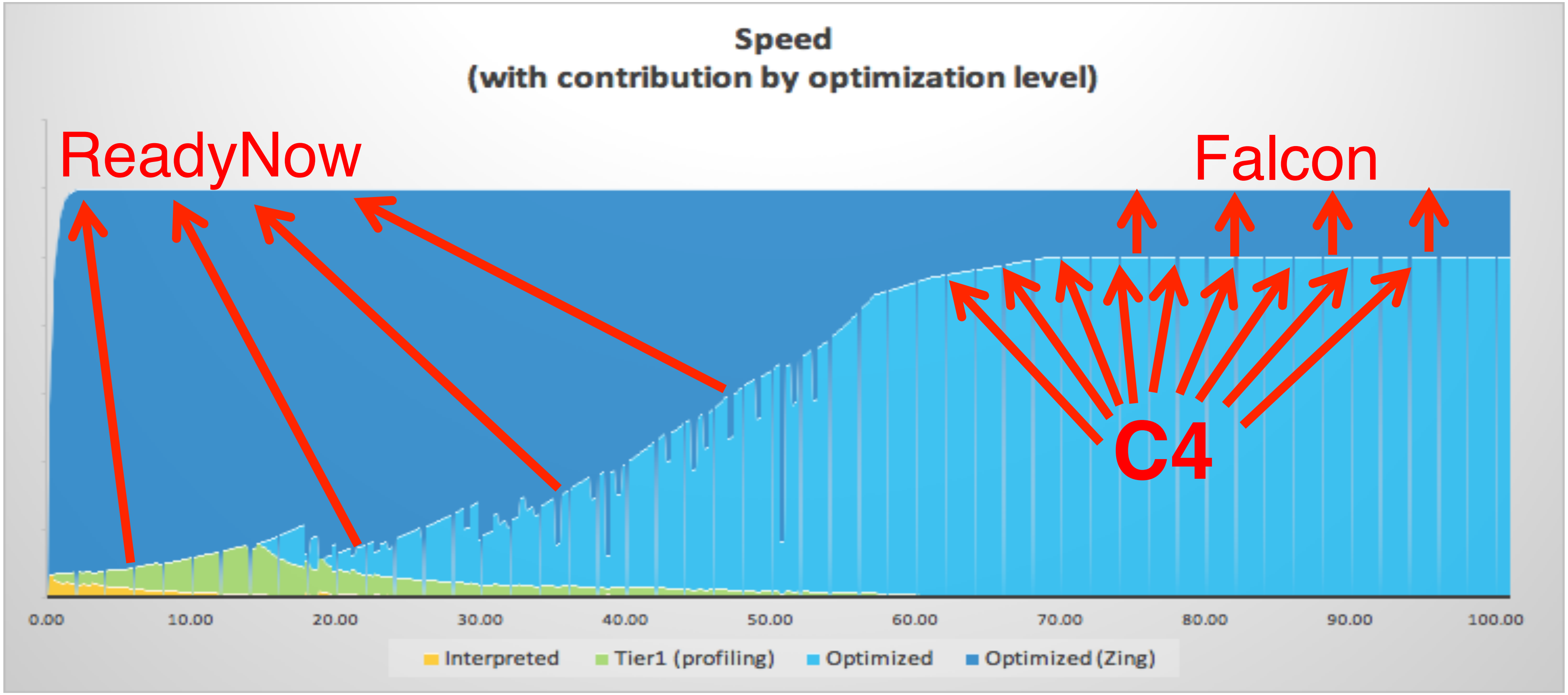
Compile Stashing

- New and rapidly improving feature for Zing
- Speculative optimisations are responsible for over half of speed improvement in JVM applications
- Zing 'stashes' compiled code for methods
  – Including speculative optimisations
- JVM requests native compiled code for method
  – Provides precise description (profile, types, etc.)

Summary



JVM Performance Graph: Zing



The Zing JVM

- Start fast
- Go faster
- Stay fast

- Simple replacement for other JVMs
  - No recoding necessary

Try Zing free for 30 days:

azul.com/zingtrial

Questions?

Simon Ritter
Deputy CTO, Azul Systems
azul.com

@speakjava

Zing: Supported Platforms

- Linux:
  - RHEL 7.0, 6.0, 5.9 or later
  - CentOS 7.0, 6.0, 5.9 or later
  - Oracle Linux 7.0, 6.0 or later
  - Red Hat MRG Realtime
  - SLES 12 SP1, 11 SP3, SP2 and SP1
  - Ubuntu 16.04 LTS, 14.04 and 12.04 LTS
  - Debian: Wheezy and Jessie
- Hypervisors: VMware, KVM
- Cloud: Amazon AWS, Docker
- Multiple Private Clouds
- Java Versions: 6, 7, 8
- Hardware: Intel/AMD x64