Priming Java for Speed

Getting Fast & Staying Fast

Gil Tene, CTO & co-Founder, Azul Systems

High level agenda

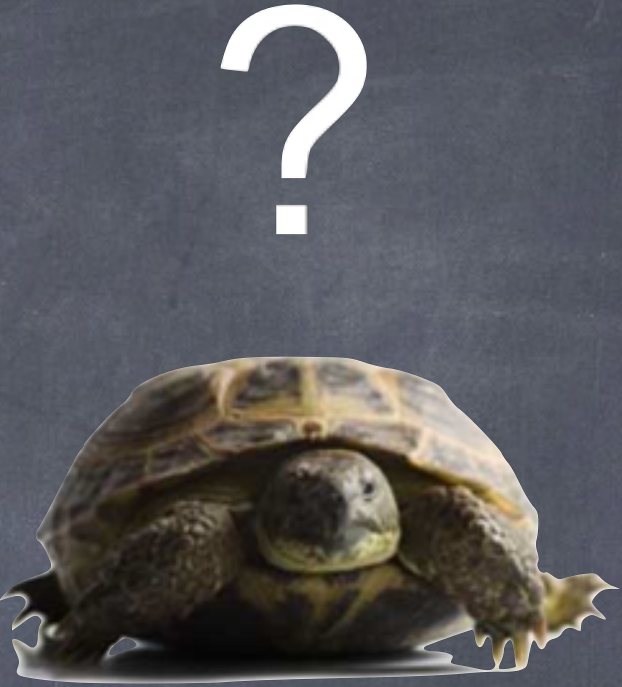
- Intro
- Java realities at “Load Start”
- A whole bunch of compiler optimization stuff
- Deoptimization...
- What we can do about it

About me: Gil Tene

- co-founder, CTO @Azul Systems
- Have been working on "think different" GC approaches since 2002
- At Azul we make JVMs that dramatically improve latency behavior
- As a result, we end up working a lot with low latency trading systems
- And (after GC is solved) the "Load Start" problem seems to dominate concerns

* working on real-world trash compaction issues, circa 2004

Example: Market Open



Are you fast at Market Open?

Market Open

Java at Market Open

Java's "Just In Time" Reality

- Starts slow, learns fast
- Lazy loading & initialization
- Aggressively optimized for the common case
- (temporarily) Reverts to slower execution to adapt

Warmup

Deoptimization

Compiler Stuff

Some simple compiler tricks

Code can be reordered…

```
int doMath(int x, int y, int z) {
   int a = x + y;
   int b = x - y;
   int c = z + x;
   return a + b;
}
```

Can be reordered to:

```
int doMath(int x, int y, int z) {
   int c = z + x;
   int b = x - y;
   int a = x + y;
   return a + b;
}
```

Dead code can be removed

```
int doMath(int x, int y, int z) {
   int a = x + y;
   int b = x - y;
   int c = z + x;
   return a + b;
}
```

Can be reduced to:

```
int doMath(int x, int y, int z) {
   int a = x + y;
   int b = x - y;
   return a + b;
}
```

Values can be propagated

```
int doMath(int x, int y, int z) {
   int a = x + y;
   int b = x - y;
   int c = z + x;
   return a + b;
}
```

Can be reduced to:

```
int doMath(int x, int y, int z) {
   return x + y + x - y;
}
```

Math can be simplified

```
int doMath(int x, int y, int z) {
  int a = x + y;
  int b = x - y;
  int c = z + x;
  return a + b;
}
```

Can be reduced to:

```
int doMath(int x, int y, int z) {
  return x + x;
}
```

Some more compiler tricks

Reads can be cached

```
int distanceRatio(Object a) {
   int distanceTo = a.x - start;
   int distanceAfter = end - a.x;
   return distanceTo/distanceAfter;
}
```

Is the same as

```
int distanceRatio(Object a) {
   int x = a.x;
   int distanceTo = x - start;
   int distanceAfter = end - x;
   return distanceTo/distanceAfter;
}
```

Reads can be cached

```
void loopUntilFlagSet(Object a) {
  while (!a.flag) {
    loopcount++;
  }
}
```

Is the same as:

```
void loopUntilFlagSet(Object a) {
  boolean flagIsSet = a.flag;
  while (!flagIsSet) {
    loopcount++;
  }
}
```

That's what volatile is for...

Writes can be eliminated

Intermediate values might never be visible

```
void updateDistance(Object a) {
   int distance = 100;
   a.x = distance;
   a.x = distance * 2;
   a.x = distance * 3;
}
```

Is the same as

```
void updateDistance(Object a) {
   a.x = 300;
}
```

Writes can be eliminated

Intermediate values might never be visible

```
void updateDistance(Object a) {
  a.visibleValue = 0;
  for (int i = 0; i < 1000000; i++) {
    a.internalValue = i;
  }
  a.visibleValue = a.internalValue;
}
```

Is the same as

```
void updateDistance(Object a) {
  a.internalValue = 1000000;
  a.visibleValue = 1000000;
}
```

Inlining...

```
public class Thing {
   private int x;
   public final int getX() { return x };
}
...
myX = thing.getX();
```

Is the same as

```
Class Thing {
   int x;
}
...
myX = thing.x;
```

Speculative compiler tricks

JIT compilers can do things that static compilers can have a hard time with…

Class Hierarchy Analysis (CHA)

- Can perform global analysis on currently loaded code
- Deduce stuff about inheritance, method overrides, etc.
- Can make optimization decisions based on assumptions
- Re-evaluate assumptions when loading new classes
- Throw away code that conflicts with assumptions before class loading makes them invalid

Inlining works without "final"

```
public class Animal {
   private int color;
   public int getColor() { return color };
}
…
myColor = animal.getColor();
```

Is the same as

```
Class Animal {
   int color;
}
…
myColor = animal.color;
```

THIS (CHA) is why field accessors are free & clean

As long as only one implementer of getColor() exists

More Speculative stuff

- The power of the "uncommon trap"
  - Being able throw away wrong code is very useful
- E.g. Speculatively assuming callee type
  - polymorphic can be "monomorphic" or "megamorphic"
  - E.g. Can make virtual calls direct even without CHA
  - E.g. Can speculatively inline things
- E.g. Speculatively assuming branch behavior
  - We've only ever seen this thing go one way, so....

Untaken path example

“Never taken” paths can be optimized away with benefits:

```
void computeMagnitude(int val) {
   if (val > 10) {
        bias = computeBias(val);
   else {
        bias = 1;
   }
   return Math.log10(bias + 99);
}
```

When all values so far were <= 10 , can be compiled to:

```
void computeMagnitude(int val) {
   if (val > 10) uncommonTrap();
   return 2;
}
```

Implicit Null Check example

All field and array access in Java is null checked

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

is (in equivalent required machine code):

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

But compiler can "hope" for non-nulls, and handle SEGV

```
<Point where later SEGV will appear to throw>
x = foo.x;
```

This is faster *IF* no nulls are encountered...

Deoptimization

Deoptimization:
Adaptive compilation is... adaptive

- Micro-benchmarking is a black art
- So is the art of the Warmup
    - Running code long enough to compile is just the start...
- Deoptimization can occur at any time
    - often occur after you *think* the code is warmed up.
- Many potential causes

Warmup often doesn't cut it...

- Common Example:
  - Trading system wants to have the first trade be fast
  - So run 20,000 "fake" messages through the system to warm up
  - let JIT compilers optimize, learn, and deopt before actual trades
- What really happens
  - Code is written to do different things "if this is a fake message"
  - e.g. "Don't send to the exchange if this is a fake message"
  - JITs optimize for fake path, including speculatively assuming "fake"
  - First real message through deopts...

Fun In a Box:
A simple Deoptimization example

- Deoptimization due to lazy loading/initialization
- Two classes: ThingOne and ThingTwo
- Both are actively called in the same method
- But compiler kicks in before one is ever called
- JIT cannot generate call for uninitialized class
  - So it leaves an uncommon trap on that path...
  - And deopts later if it ever gets there.

https://github.com/giltene/GilExamples/blob/master/src/main/java/FunInABox.java

Deopt example: Fun In a Box

```java
public class FunInABox {
    static final int THING_ONE_THREASHOLD = 20000000;
    .
    .
    .
    static public class ThingOne {
        static long valueOne = 0;

        static long getValue() { return valueOne++; }
    }

    static public class ThingTwo {
        static long valueTwo = 3;

        static long getValue() { return valueTwo++; }
    }

    public static long testRun(int iterations) {
        long sum = 0;

        for(int iter = 0; iter < iterations; iter++) {
            if (iter > THING ONE THREASHOLD)
                sum += ThingOne.getValue();
            else
                sum += ThingTwo.getValue();
        }
        return sum;
    }
```

Deopt example: Fun In a Box

```
Lumpy.local-40%
Lumpy.local-40% java -XX:+PrintCompilation FunInABox
     77   1       java.lang.String::hashCode (64 bytes)
    109   2       sun.nio.cs.UTF_8$Decoder::decodeArrayLoop (553 bytes)
    115   3       java.math.BigInteger::mulAdd (81 bytes)
    118   4       java.math.BigInteger::multiplyToLen (219 bytes)
    121   5       java.math.BigInteger::addOne (77 bytes)
    123   6       java.math.BigInteger::squareToLen (172 bytes)
    127   7       java.math.BigInteger::primitiveLeftShift (79 bytes)
    130   8       java.math.BigInteger::montReduce (99 bytes)
    140   1%      java.math.BigInteger::multiplyToLen @ 138 (219 bytes)
Starting warmup run (will only use ThingTwo):
    147   9       sun.security.provider.SHA::implCompress (491 bytes)
    153  10       java.lang.String::charAt (33 bytes)
    154  11       FunInABox$ThingTwo::getValue (10 bytes)
    154   2%      FunInABox::testRun @ 4 (38 bytes)
    161  12       FunInABox::testRun (38 bytes)
Warmup run [1000000 iterations] took 27 msec..

...Then, out of the box
Came Thing Two and Thing One!
And they ran to us fast
They said, "How do you do?"...

Starting actual run (will start using ThingOne a bit after using
ThingTwo):
   5183  12      made not entrant  FunInABox::testRun (38 bytes)
   5184   2%     made not entrant  FunInABox::testRun @ -2 (38 bytes)
   5184   3%      FunInABox::testRun @ 4 (38 bytes)
   5184  13       FunInABox$ThingOne::getValue (10 bytes)
Test run [200000000 iterations] took 1299 msec...
```

Deopt example: Fun In a Box

```
.
.
.
public static <T> Class<T> forceInit(Class<T> klass) {
    // Forces actual initialization (not just loading) of the class klass:
    try {
        Class.forName(klass.getName(), true, klass.getClassLoader());
    } catch (ClassNotFoundException e) {
        throw new AssertionError(e);  // Can't happen
    }
    return klass;
}

public static void tameTheThings() {
    forceInit(ThingOne.class);
    forceInit(ThingTwo.class);
}
.
.
.
```

https://github.com/giltene/GilExamples/blob/master/src/main/java/FunInABox.java

Deopt example: Fun In a Box

```
Lumpy.local-41%
Lumpy.local-41% java -XX:+PrintCompilation FunInABox KeepThingsTame
     75   1       java.lang.String::hashCode (64 bytes)
    107   2       sun.nio.cs.UTF_8$Decoder::decodeArrayLoop (553 bytes)
    113   3       java.math.BigInteger::mulAdd (81 bytes)
    115   4       java.math.BigInteger::multiplyToLen (219 bytes)
    119   5       java.math.BigInteger::addOne (77 bytes)
    121   6       java.math.BigInteger::squareToLen (172 bytes)
    125   7       java.math.BigInteger::primitiveLeftShift (79 bytes)
    127   8       java.math.BigInteger::montReduce (99 bytes)
    133   1%      java.math.BigInteger::multiplyToLen @ 138 (219 bytes)
Keeping ThingOne and ThingTwo tame (by initializing them ahead of time):
Starting warmup run (will only use ThingTwo):
    140   9       sun.security.provider.SHA::implCompress (491 bytes)
    147  10       java.lang.String::charAt (33 bytes)
    147  11       FunInABox$ThingTwo::getValue (10 bytes)
    147   2%      FunInABox::testRun @ 4 (38 bytes)
    154  12       FunInABox::testRun (38 bytes)
Warmup run [1000000 iterations] took 24 msec..

...Then, out of the box
Came Thing Two and Thing One!
And they ran to us fast
They said, "How do you do?"...

Starting actual run (will start using ThingOne a bit after using
ThingTwo):
   5178  13       FunInABox$ThingOne::getValue (10 bytes)
Test run [200000000 iterations] took 2164 msec...
```

Example causes for depotimization at "Load Start"

- First calls to method in an uninitialized class
- First call to a method in a not-yet-loaded class
- Dynamic branch elimination hitting an unexpected path
- Implicit Null Check hitting a null

Java's "Just In Time" Reality

- Starts slow, learns fast
- Lazy loading & initialization
- Aggressively optimized for the common case
- (temporarily) Reverts to slower execution to adapt

Warmup

Deoptimization

Java's "Just In Time" Reality

What we have

- Starts slow, learns fast
- Lazy loading & initialization
- Aggressively optimized for the common case
- (temporarily) Reverts to slower execution to adapt

What we want

- No Slow Ops or Trades

ReadyNow!
to the rescue

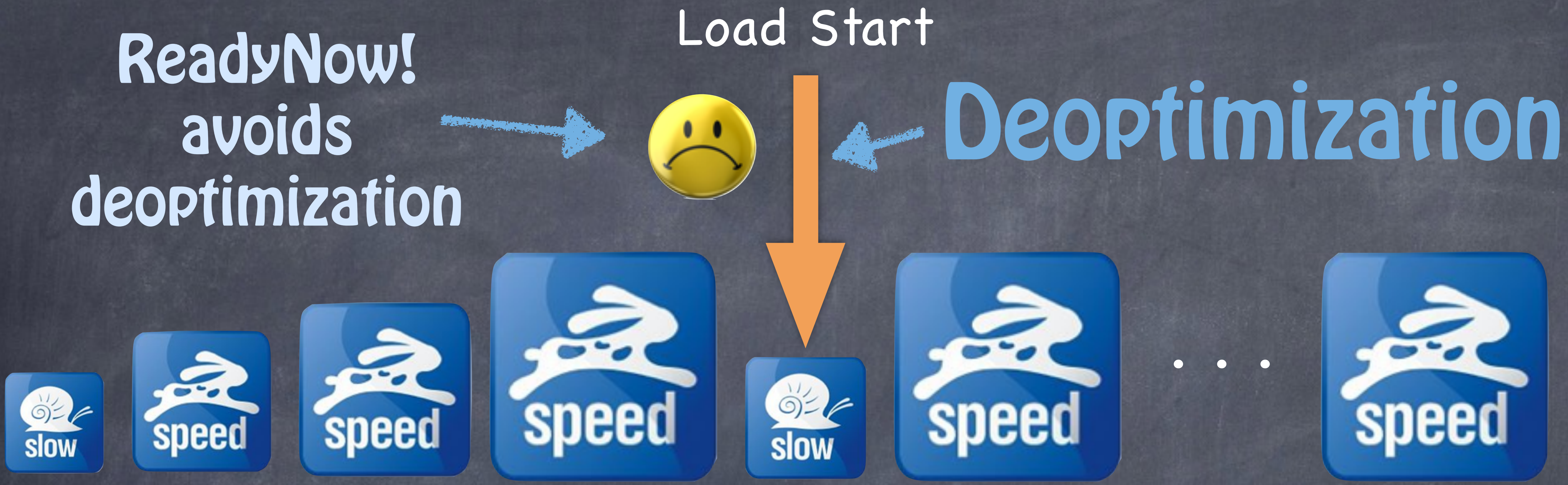
What can we do about it?

- Zing has a new feature set called “ReadyNow!”
  - Focused on avoiding deoptimization while keeping code fast
- First of all, paying attention matters
  - E.g. Some of those dynamic branch eliminations have no benefit…
- Adds optional controls for helpful things like:
  - Aggressive class loading (load when they come into scope)
  - Safe pre-initialization of classes with empty static initializers
  - Per method compiler directives (e.g. disable ImplicitNullChecks)
  - …

Logging and “replaying” optimizations

- Zing’s ReadyNow includes optimization logging
  - Records ongoing optimization decisions and stats
  - records optimization dependencies
  - Establishes “stable optimization state” and end of previous run
- Zing can read prior logs at startup
  - Prime JVM with knowledge of prior stable optimizations
  - Optimizations are applied as their dependencies get resolved
- ReadyNow workflow promotes confidence
  - You’ll know if/when all optimizations have been applied
  - If some optimization haven’t been applied, you’ll know why…

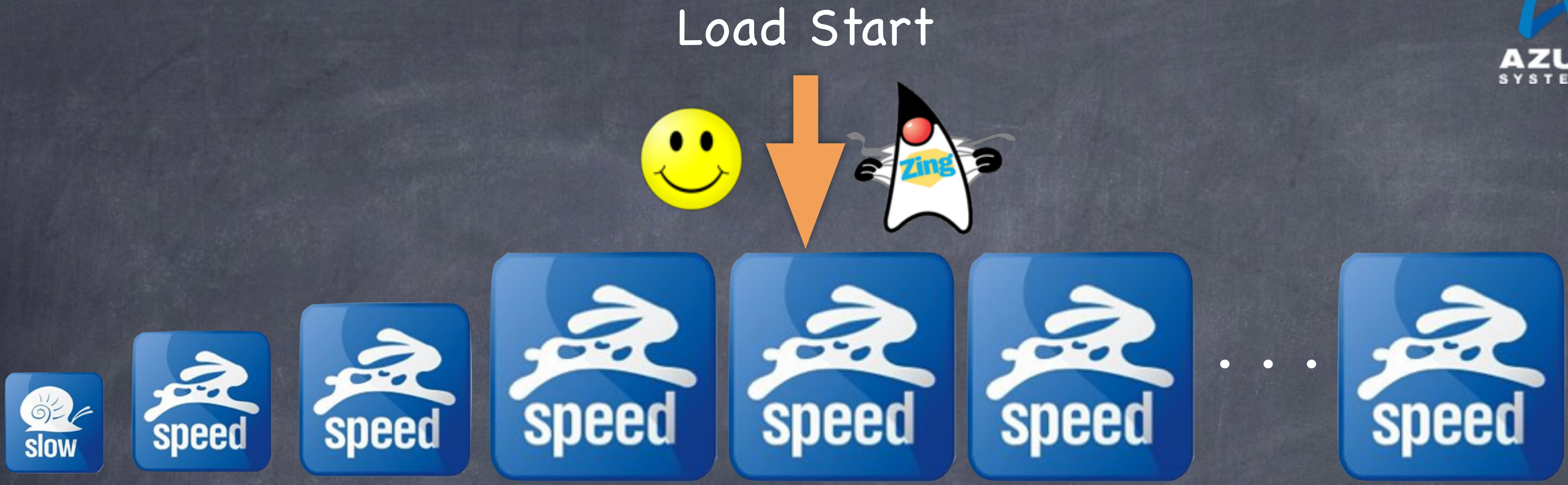
Java at “Load Start”



ReadyNow! avoids deoptimization

Load Start

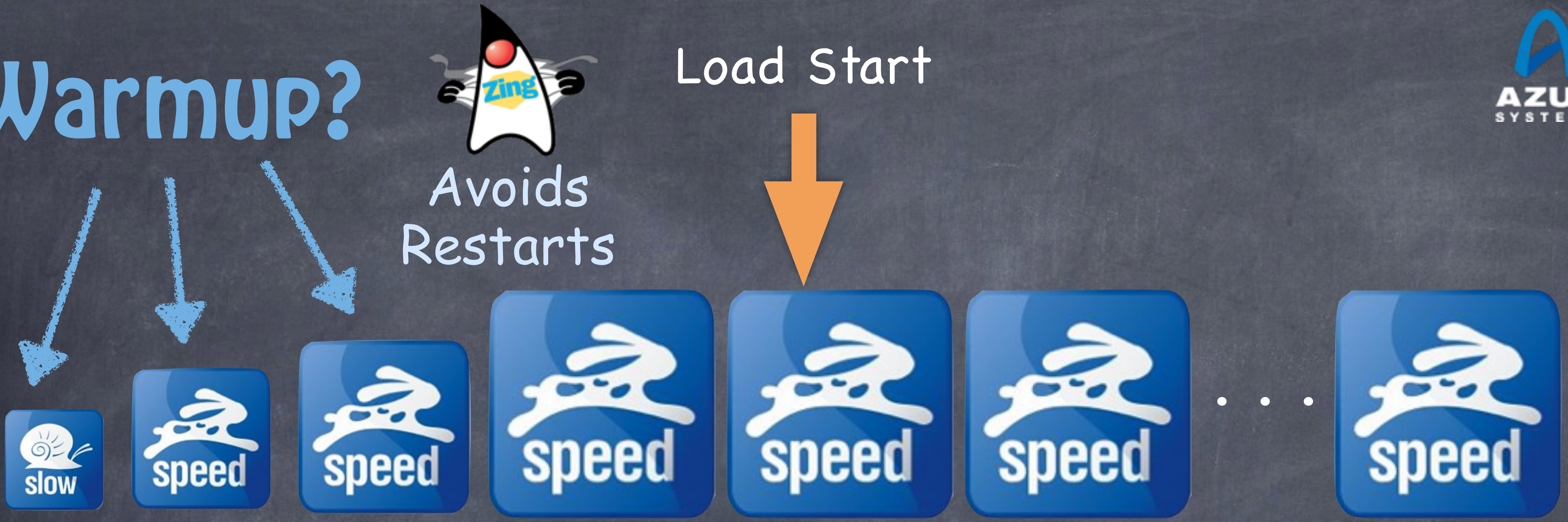
Deoptimization



Java at “Load Start”

With Zing & ReadyNow!

Warmup?

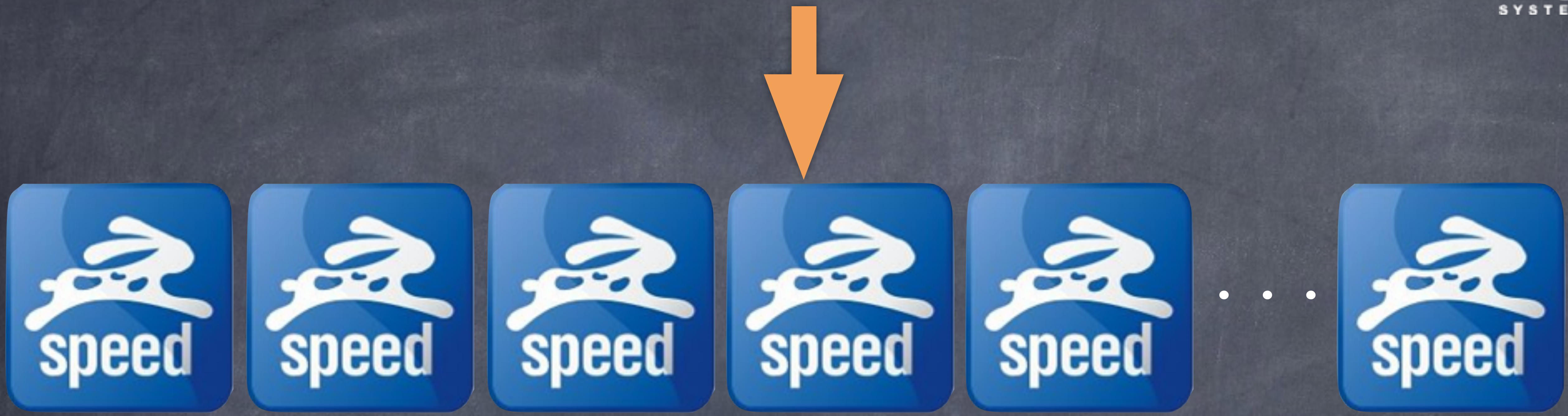


Avoids Restarts

Load Start

Java at “Load Start”

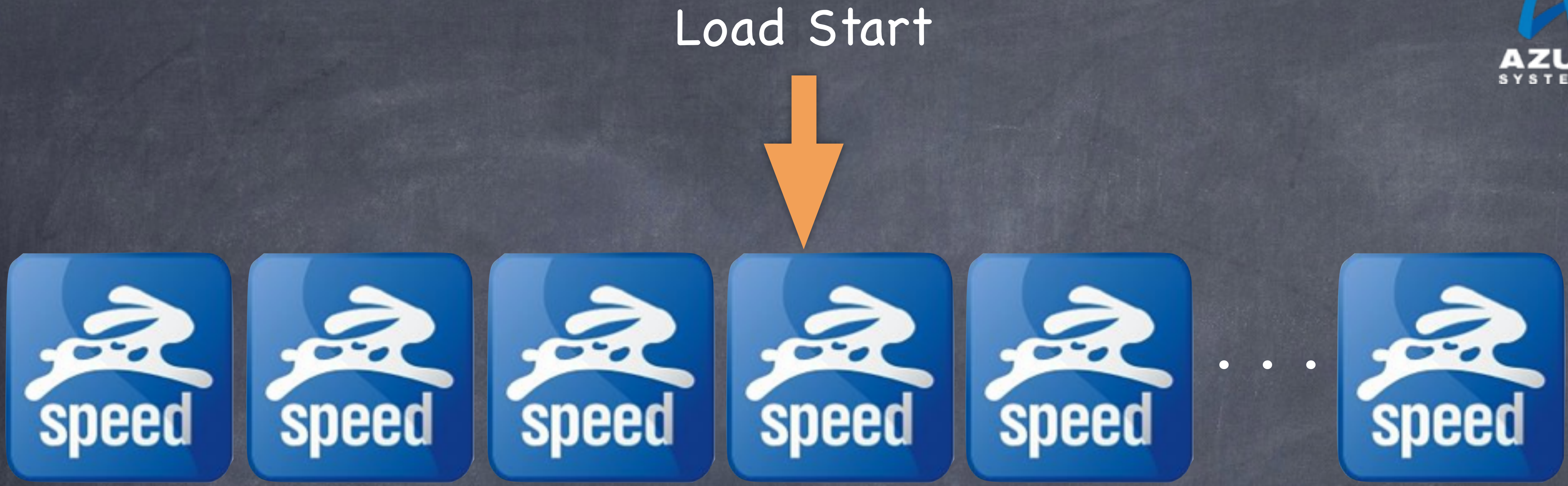
With ReadyNow!


Load Start

Java at “Load Start”

With ReadyNow! and
No Overnight Restarts

Start Fast & Stay Fast



One liner takeaway

Zing: A cure for the Java hiccups