Shenandoah: Theory and Practice

Christine Flood
Roman Kennke
Principal Software Engineers
Red Hat

Shenandoah

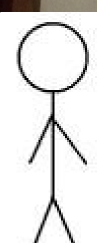
Christine Flood Roman Kennke
Principal Software Engineers
Red Hat

Shenandoah

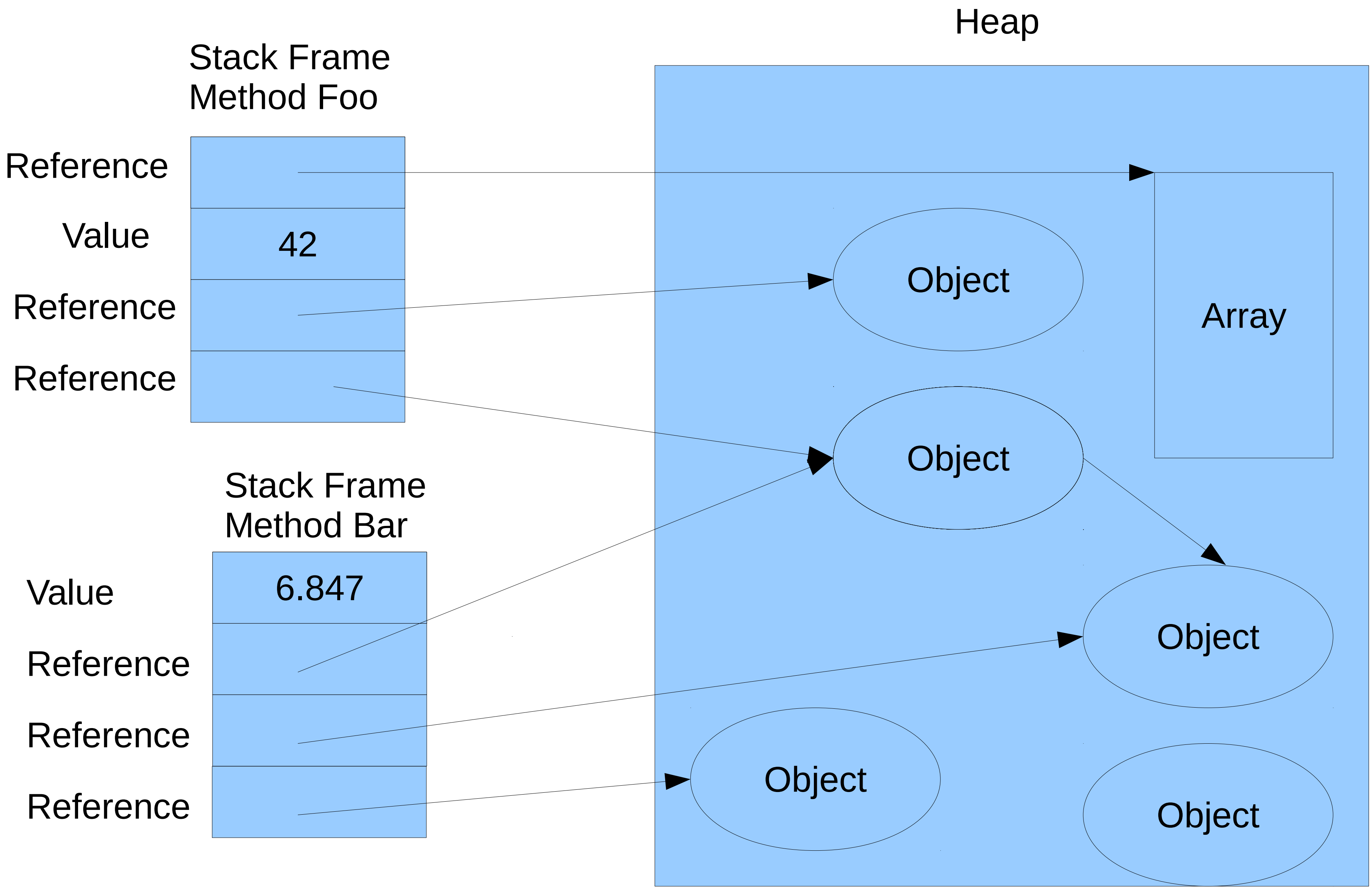
- Why do we need it?
- What does it do?
- How does it work?
- What's the current state?
- What's left to do?
- Performance

GC is like an omniscient organizer for program memory.

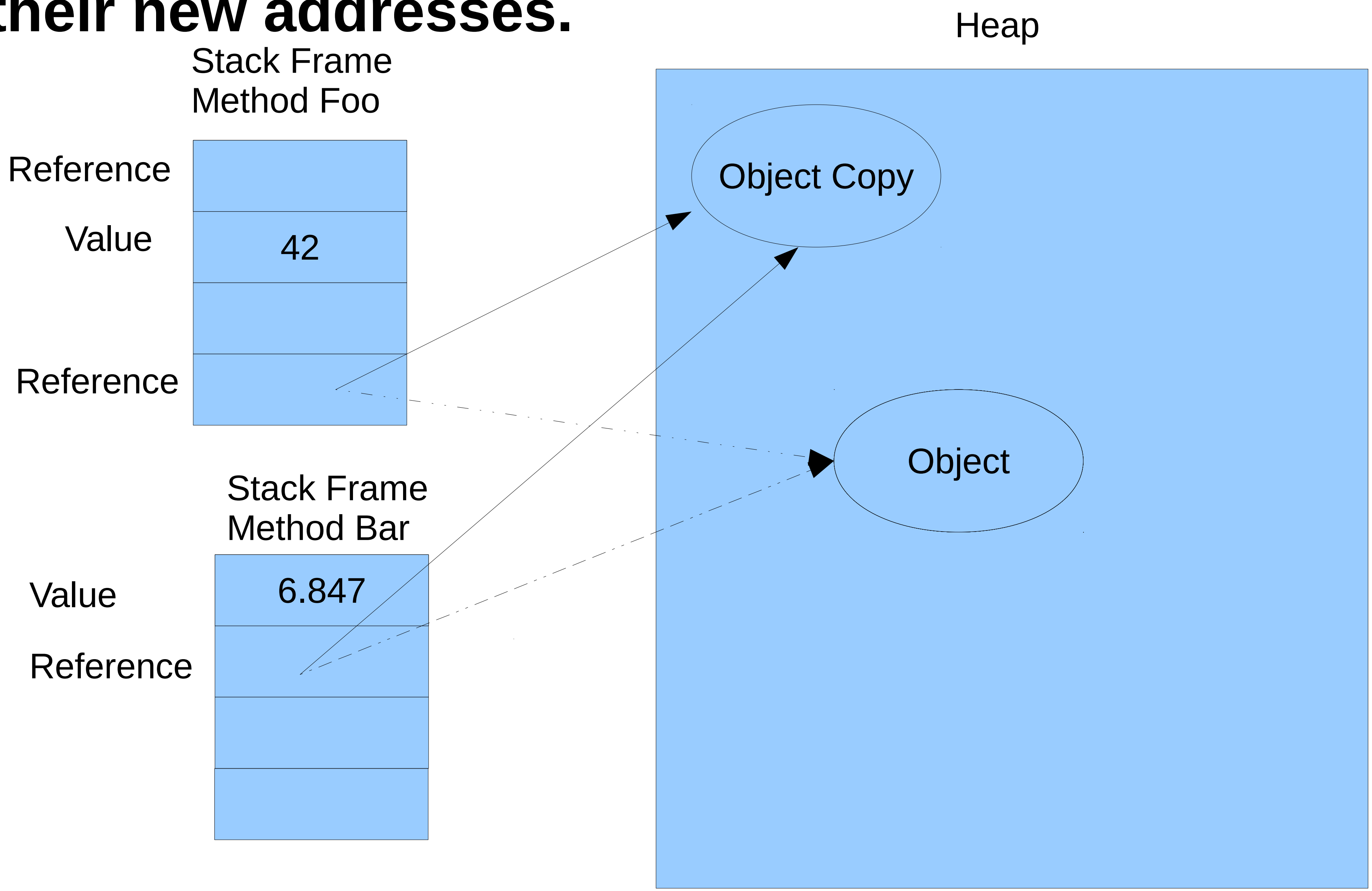
I bet that's your messy pantry isn't it?



Java execution



When we reorganize objects we need to copy the objects and update the stack locations to point to their new addresses.



Why yet another garbage collector?

- OpenJDK already has 4 collectors:
  - Serial
  - Parallel
  - Concurrent Mark Sweep
  - G1

Why yet another garbage collector?

- OpenJDK already has 4 collectors:
  - Serial (minimal collector)
  - Parallel (high throughput)
  - Concurrent Mark Sweep (low pause time, but...)
  - G1 (low/managed pause time, but...)

But?

- All existing collectors must (occasionally) compact old-gen or the whole heap
- .. and therefore stop the world
- …. for a long time, if heap is large

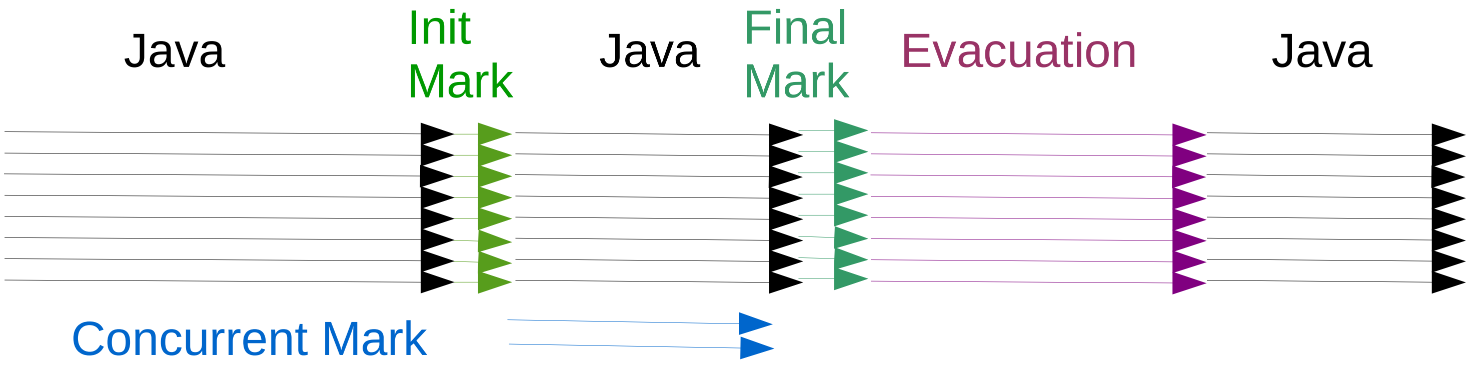
Shenandoah!

- Aims to reduce GC pause times
- Goal: <10ms pauses for >100GB heaps
- More precisely:
  - Make GC pauses independent of heap size
- Long-term goal: pauseless GC

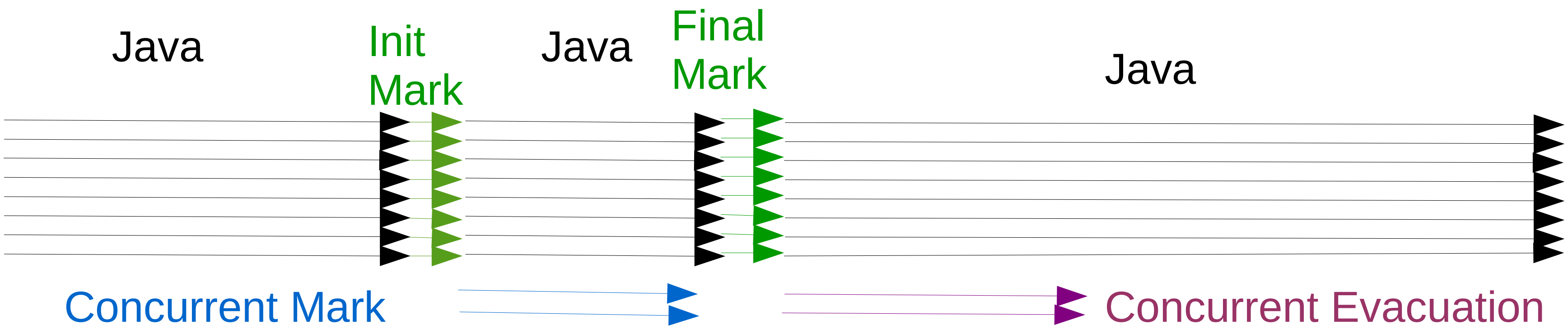
How do we do it?

- Evacuate concurrently with Java threads

Garbage-First (G1)

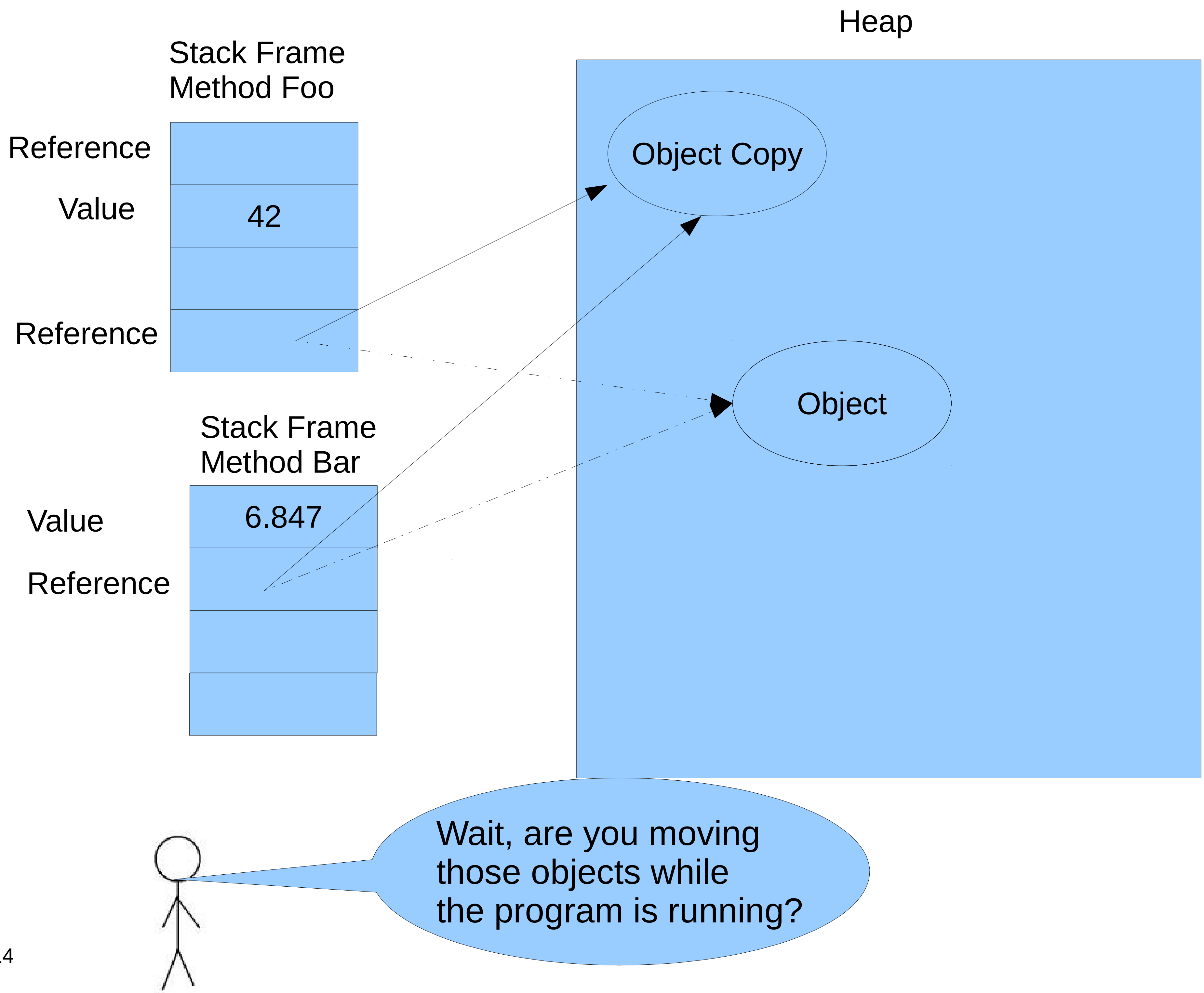


Shenandoah: Current implementation



We choose our collection set to
Minimize amount of copying.

We have a plan for removing
all of the stop the world pauses.



Heap
Stack Frame
Method Foo
Reference
Value
42
Reference
Object Copy
Object
Stack Frame
Method Bar
Value
6.847
Reference
Wait, are you moving
those objects while
the program is running?

How do we do that?

We recycle an idea from the 1980's and add a level of indirection.

Forwarding Pointers based on Brooks Pointers

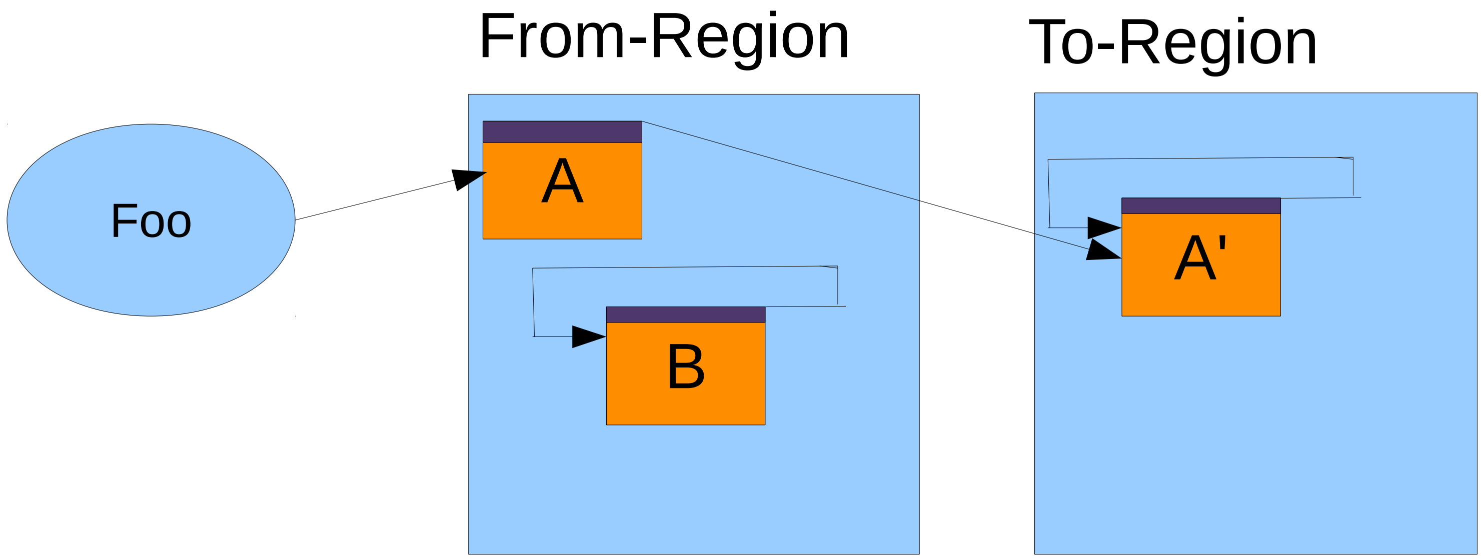
- Rodney A. Brooks “Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware”

  1984 Symposium on Lisp and Functional Programing

Forwarding Pointer

- Object layout inside the JVM remains the same.
- Third party tools can still walk the heap.
- Can choose GC algorithm at run time.
- We hope to one day be able to take advantage of unused space in double word aligned objects when possible.

Forwarding Pointers

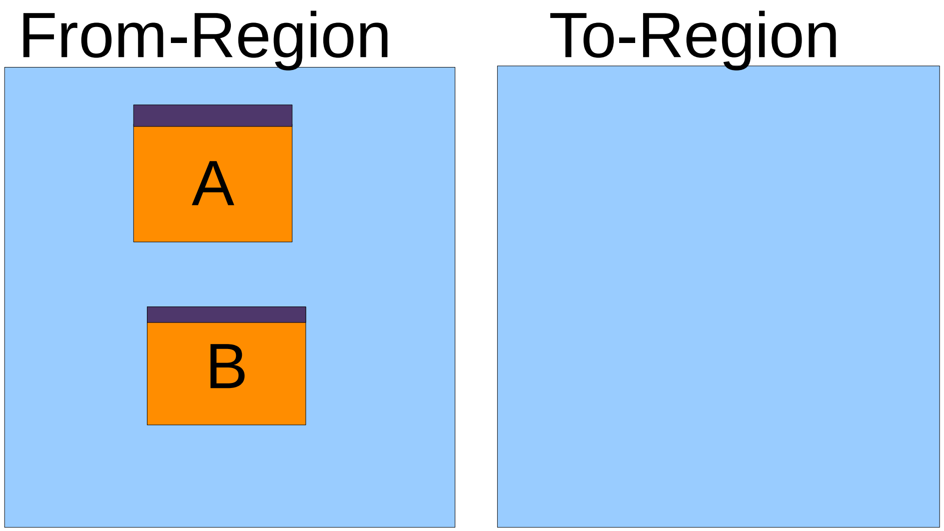


Any reads or writes of A will now be redirected to A'. We don't need to update Foo immediately.

How to move an object while the program is running.

- Read the forwarding pointer to from space.
- Allocate a temporary copy of the object in to space.
- Copy the data.
- CAS the forwarding pointer.
  - If you succeed carry on.
  - If you fail, use the copy that was placed by the thread that beat you and recycle your temporary copy.

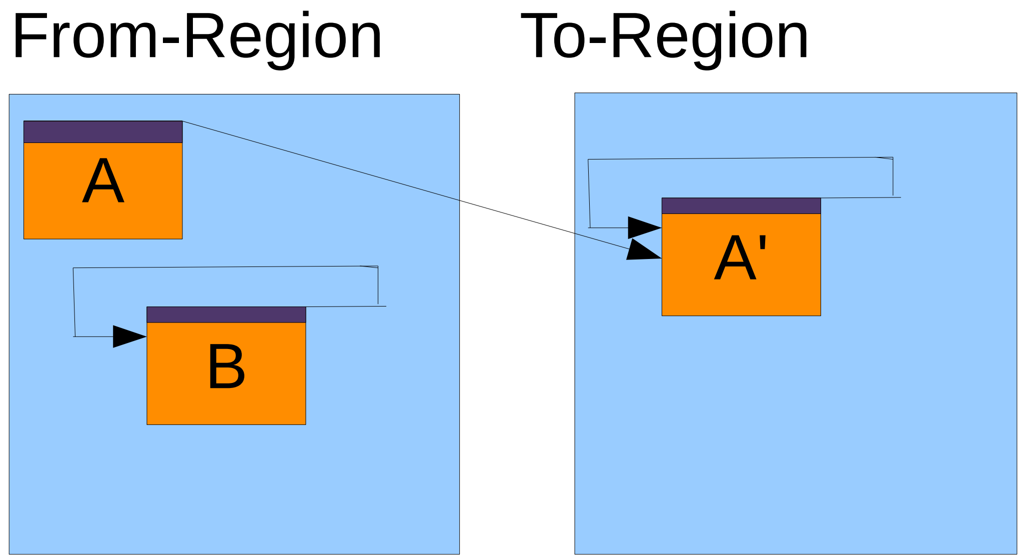
Forwarding Pointers



Reading an object in a From-region doesn't trigger an evacuation.

Note: If reads were to cause copying we might have a “read storm” where every operation required copying an object. Our intention is that since we are only copying on writes we will have less bursty behavior.

Forwarding Pointers



Writing an object in a From-Region will trigger an evacuation of that object to a To-Region and the write will occur in there.

How does Java code know where the real object is?

- Reads, writes, amps and some others are wrapped by code that ensures the correct objects are accessed:
- Read barriers
- Write barriers
- Acmp / cmpxchg barriers

Read Barriers

- Read the forwarding pointer to access the forwarded object.
- Does not trigger evacuation
- If a write occurs concurrently, it's a race, but it's been a race before :-)
- Usually compiles into a single mov instruction

Write Barriers

- Ensures that writes only happen in to-space
- It does so by speculatively making a copy, then CASing the forwarding pointer in the object
- If CAS succeeds, we win. If not, we roll back the allocation, and use whatever the other thread did
- … but only for objects in collection set, and only if evacuation is currently in progress
- … otherwise it's a simple read barrier

Acmp barriers

- If we compare a == a', we can get false negatives
- Therefore, if an object comparison fails, we resolve both operands through a read barrier, then try again.
-

CmpXChg Barriers

- compareAndSwapObject() combines all three, because it loads, compares and writes an object field
- We insert a somewhat complex barrier that
  - Resolves the written value (read-barrier)
  - Ensures to-space copy (write-barrier)
  - Prevents false negative (acmp-barrier)

How are barriers implemented?

- Need two types of barriers:
  - Read barrier - read brooks pointer
  - Write barrier – maybe copy obj & update brooks ptr

- `oop read_barrier(oop obj)`
- `oop write_barrier(oop obj)`

Shenandoah barriers

```
oop read_barrier(oop obj) {
  return *(obj-0x8);
}
```

Shenandoah barriers

```
oop write_barrier(oop obj) {
  if (evacuation_in_progress) {
    return runtime_wbarrier(obj);
  }
  return obj;
}
```

Shenandoah barriers

- Read barriers:
  - getfield
  - Xaload
  - Intrinsics
  - Some esoteric stuff

Shenandoah barriers

- Write barriers:
  - putfield
  - Xastore
  - Intrinsics
  - Some esoteric stuff

Shenandoah barrier example

```
// Method without barriers
void doStuff(TypeA a, TypeA b) {
  for (..) {
    a.x = 3;                 // putfield
    System.out.println(b.x); // getfield
  }
}

// Same method with Shenandoah barriers
void doStuff(TypeA a, TypeA b) {
  for (..) {
    a = write_barrier(a);
    a.x = 3;                 // putfield
    b = read_barrier(b);
    System.out.println(b.x); // getfield
  }
}
```

Shenandoah barriers

- Barriers are inserted by:
  - The interpreter
  - The C1 compiler
  - The C2 compiler
  - By us, hardcoded in the runtime

Shenandoah barriers

- Initial implementation showed disheartening performance: more than 50% slower than with other Gcs
- So how did we make it fast?

Shenandoah barriers

- How to optimize barriers?
  - Make barrier more efficient
  - Eliminate barriers
  - Optimize barrier placement

Shenandoah barriers

- Making barriers more efficient
  - Eliminate null-checks
  - Inline null-checks
  - Inline evacuation-in-progress checks
  - Inline in-collection-set checks

→ Only call runtime when really necessary

Shenandoah barriers

- Eliminate barriers
- We don't need barriers:
  - For known NULL objects
  - For inlined constants
  - For newly allocated objects
  - After write barriers
- Since we can only figure most of this out after parsing, this isn't possible to do with parse-time barriers

Eliminate barriers on null objects

```
bool isNull(Type a) {
  Type b = null;
  a' = read_barrier(a);
  b' = read_barrier(b);
  return a' == b';
}
```

Eliminate barriers on null objects

```
bool isNull(Type a) {
  Type b = null;
  a' = read_barrier(a); // Dont care
  b' = read_barrier(b); // Known null
  return a' == b';
}
```

Eliminate barriers on null objects

```
bool isNull(Type a) {
  Type b = null;
  return a == b;
}
```

Eliminate barriers on constants

```
static final Type A = ...;
int getFoo() {
  return A.foo;
}
```

Eliminate barriers on constants

```
static final Type A = ...;
int getFoo() {
  Type A' = read_barrier(A);
  return A'.foo;
}
```

Eliminate barriers on constants

```
static final Type A = ...;
int getFoo() {
  // Constants are always in to-space
  Type A' = read_barrier(A);
  return A'.foo;
}
```

Eliminate barriers on new objects

```
int getFoo() {
  Type a = new Type();
  a' = read_barrier(a);
  return a'.foo;
}
```

Eliminate barriers on new objects

```
int getFoo() {
  Type a = new Type();
  // New objects are always in to-space
  a' = read_barrier(a);
  return a'.foo;
}
```

Eliminate barriers on new objects

```
int getFoo() {
  Type a = new Type();
  return a.foo;
}
```

Eliminate barriers after write barriers

```
int getFoo(Type a) {
  a' = write_barrier(a);
  a'.bar = …;
  a'' = read_barrier(a');
  return a''.foo;
}
```

Eliminate barriers after write barriers

```
int getFoo(Type a) {
  a' = write_barrier(a);
  a'.bar = ...;
  // a' already in to-space
  a'' = read_barrier(a');
  return a''.foo;
}
```

Eliminate barriers after write barriers

```
int getFoo(Type a) {
  a' = write_barrier(a);
  a'.bar = …;
  return a'.foo;
}
```

Optimize barrier placement

- Hoist barriers out of hot loops

Example

```
void doStuff(TypeA a, TypeZ z) {
  for (…) {
    Call(); // Safepoint
    for (…) {
      a = write_barrier(a);
      a.x = foo;
      z = read_barrier(z);
      System.out.println(z.y);
    }
  }
```

Example

```
void doStuff(TypeA a, TypeZ z) {
  a = write_barrier(a);
  for (…) {
    Call(); // Safepoint
    for (…) {
      a.x = foo;
      z = read_barrier(z);
      System.out.println(z.y);
    }
  }
```

Example

```
void doStuff(TypeA a, TypeZ z) {
  a = write_barrier(a);
  z = read_barrier(z);
  for (…) {
    Call(); // Safepoint
    for (…) {
      a.x = foo;
      System.out.println(z.y);
    }
  }
```

Lessons learned

- Basic algorithm pretty easy
- Hard parts:
  - Finding all the right places where to insert barriers
  - Support all JVM peculiarities:
    - Weak references
    - JNI Critical regions
    - System.gc()
  - Compiler support and optimization

Status

- Feature complete
- Stable (beta-quality)
- Good performance (see later…)
- Established OpenJDK project:
  http://openjdk.java.net/projects/shenandoah/
- Got nightly builds:
  - https://adopt-openjdk.ci.cloudbees.com/view/OpenJDK/

    (Thanks Adopt-OpenJDK!!)

Future Work (last year)

- Finish big application testing.
- Move the barriers to right before code generation.
- Barrier-specific C2 opts?
- Exploit Java Memory Model?
- Heuristics tuning!
- Generational Shenandoah?
- Remembered Sets for updating roots and freeing memory sooner?
- Round Robin Thread Stopping?
- NUMA Aware?

Future Work (now)

- Finish big application testing.
- Move the barriers to right before code generation.
- Barrier-specific C2 opts?
- ~~Exploit Java Memory Model?~~
- Heuristics tuning!
- ~~Generational Shenandoah?~~
- ~~Remembered Sets for updating roots and freeing memory sooner?~~
- Round Robin Thread Stopping? (2.0)
- NUMA Aware? (2.0)

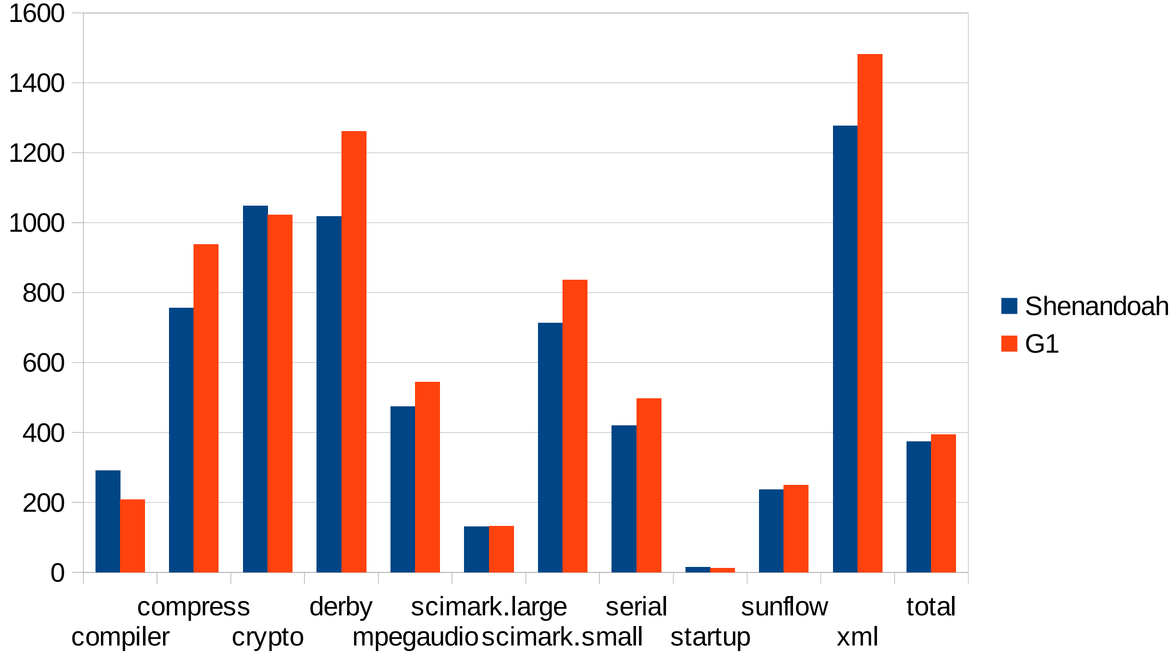
Releases?

- First in Fedora 24
- JDK 10
- JEP 189: http://openjdk.java.net/jeps/189

Performance

- SPECjbb2015

\- 32 cores
\- 160GB RAM, 140GB heap



Throughput: Shenandoah: 374ops/m G1: 393ops/m (95%, min 80%, max 140%)
Pauses: Shenandoah: avg: 41ms, max: 202ms
G1: avg: 240ms, max: 1126ms

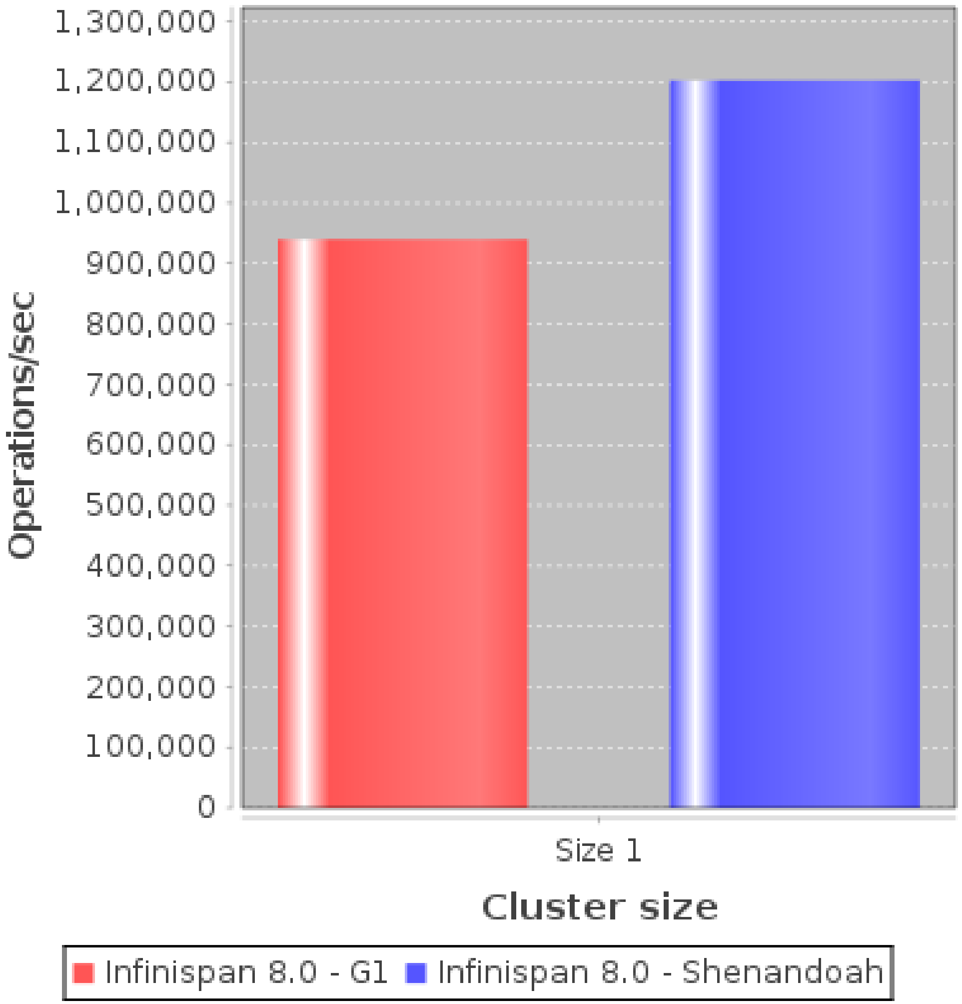
Performance SPECjbb2015

- Max-jops: maximum throughput
- Critical-jops: throughput under response-time-constraints (SLA)

	G1	Shenandoah	
Max-jops	18117	16899	93%
Critical-jops	4294	7990	186%
Pause avg	862ms	24.6ms	
Pause max	2054ms	78.61	

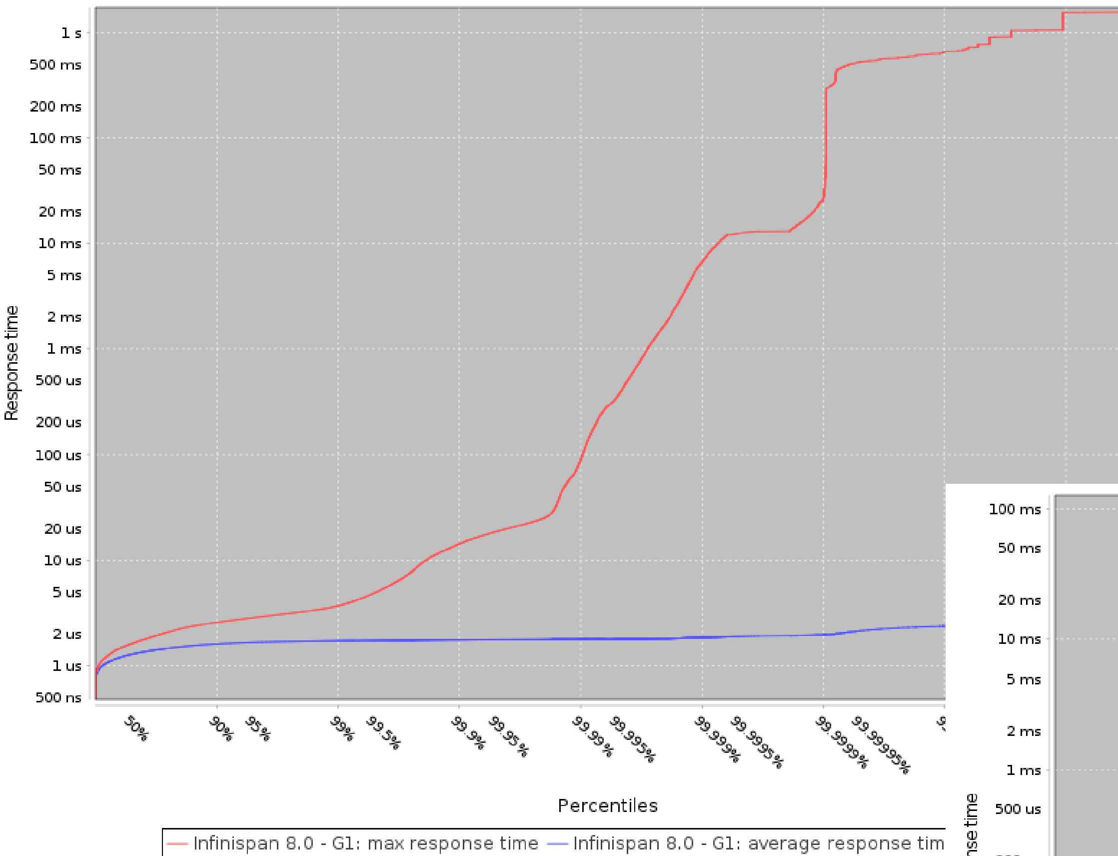
Performance Radargun/Infinispan

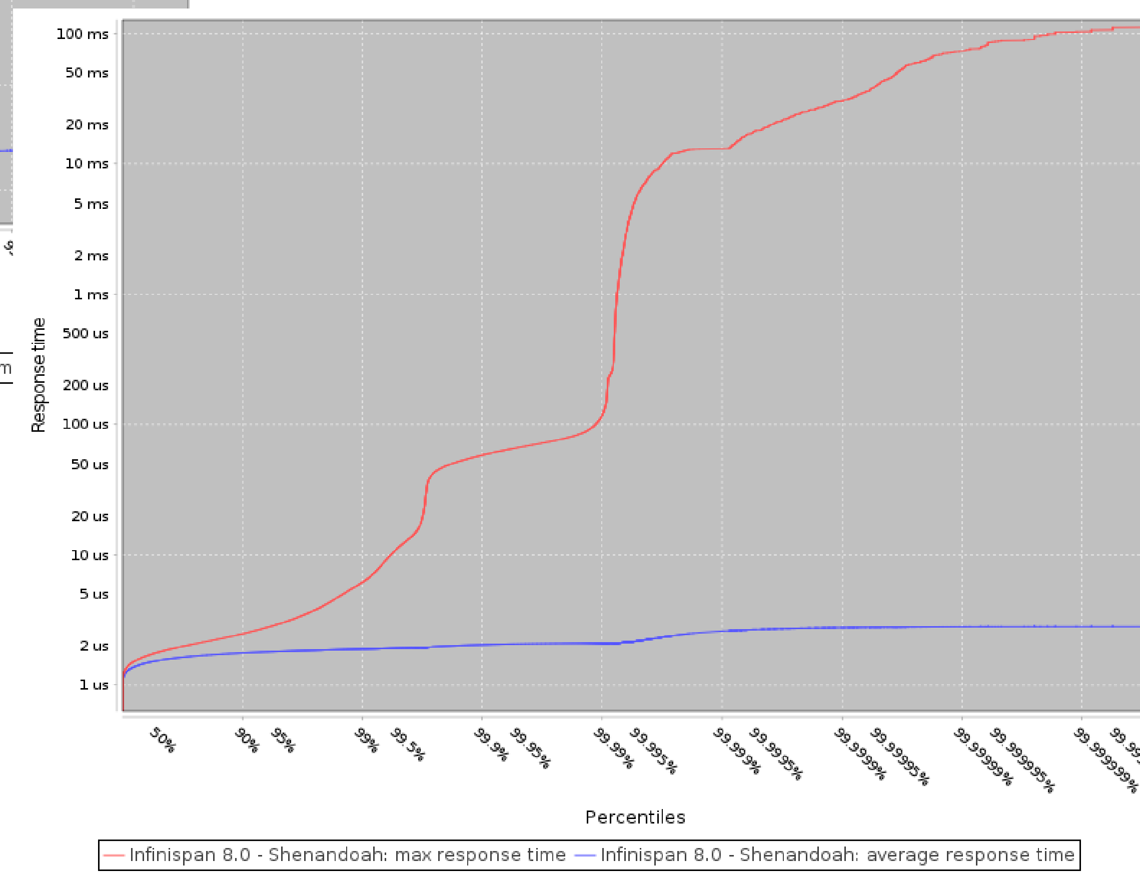
Throughput:



G1: 940,065 reqs/s Shenandoah: 1,202,925 reqs/s

Performance Radargun/Infinispan





Response time percentiles

Beware the scales!

LRU test

- Simple handwritten LRU cache benchmark
- ParallelGC: 116091ms / 100000 ops
- G1: 98598ms / 100000 ops
- Shenandoah: 56698ms / 100000 ops

Please test

- Download and build:
  - http://hg.openjdk.java.net/shenandoah
- Or use nightly builds:
  - https://adopt-openjdk.ci.cloudbees.com/view/OpenJDK/job/project-shenandoah-jdk9/
  - https://adopt-openjdk.ci.cloudbees.com/view/OpenJDK/job/project-shenandoah-jdk8/
- Report issues or success stories to:
  - http://mail.openjdk.java.net/mailman/listinfo/shenandoah-dev

References

- http://openjdk.java.net/projects/shenandoah/