Shenandoah: An ultra-low pause time garbage collector for OpenJDK

Christine H. Flood
Principal Software Engineer
Red Hat

Benefits of Garbage Collection

- Omniscient housecleaning
- No premature freeing of objects
- Safety (no programming with pointers)
- Fewer memory leaks
- Better cache/page locality
- Less memory fragmentation
- Fast allocation (pointer bump)

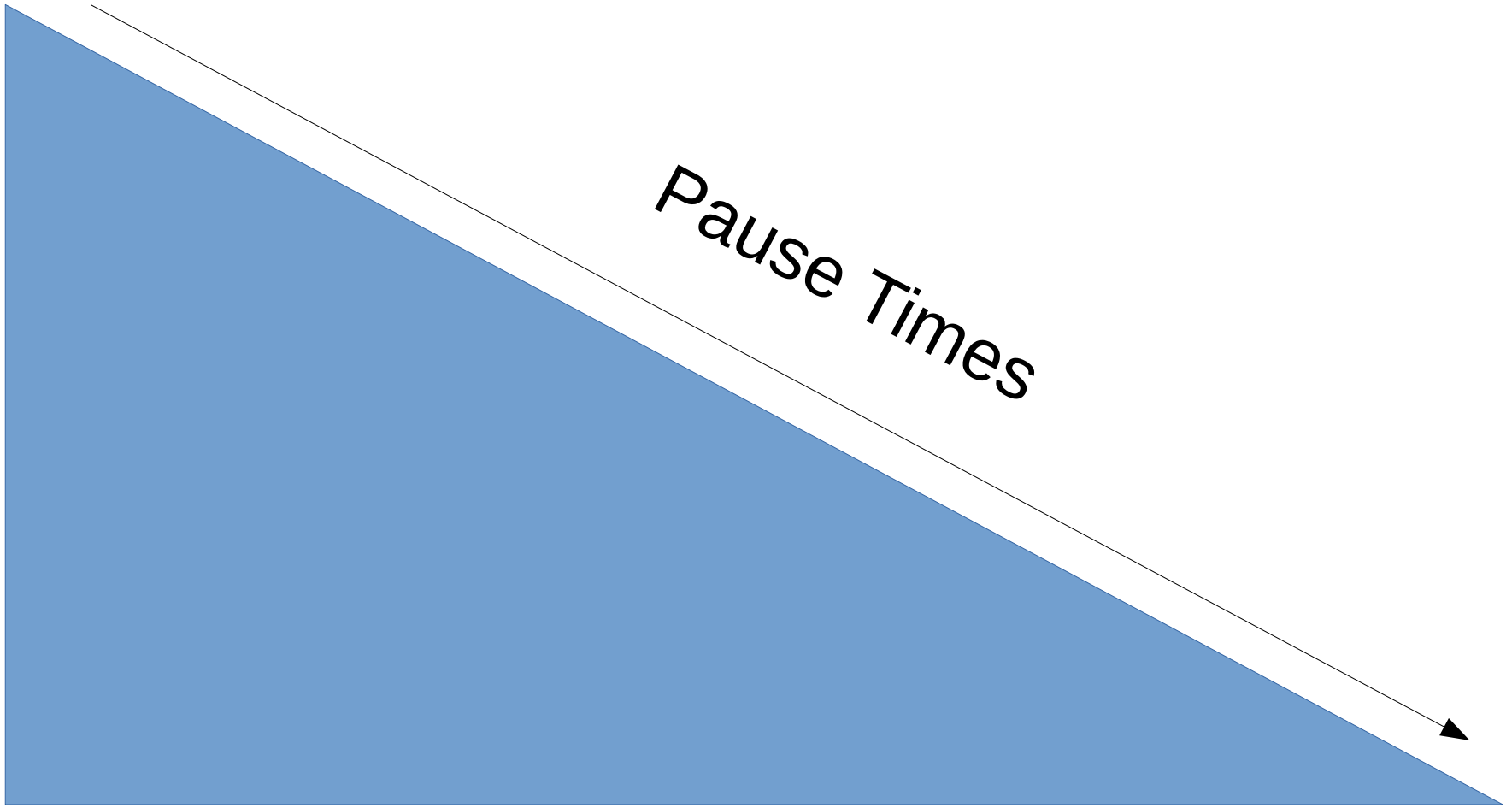
Downside to Garbage Collection

- Pauses

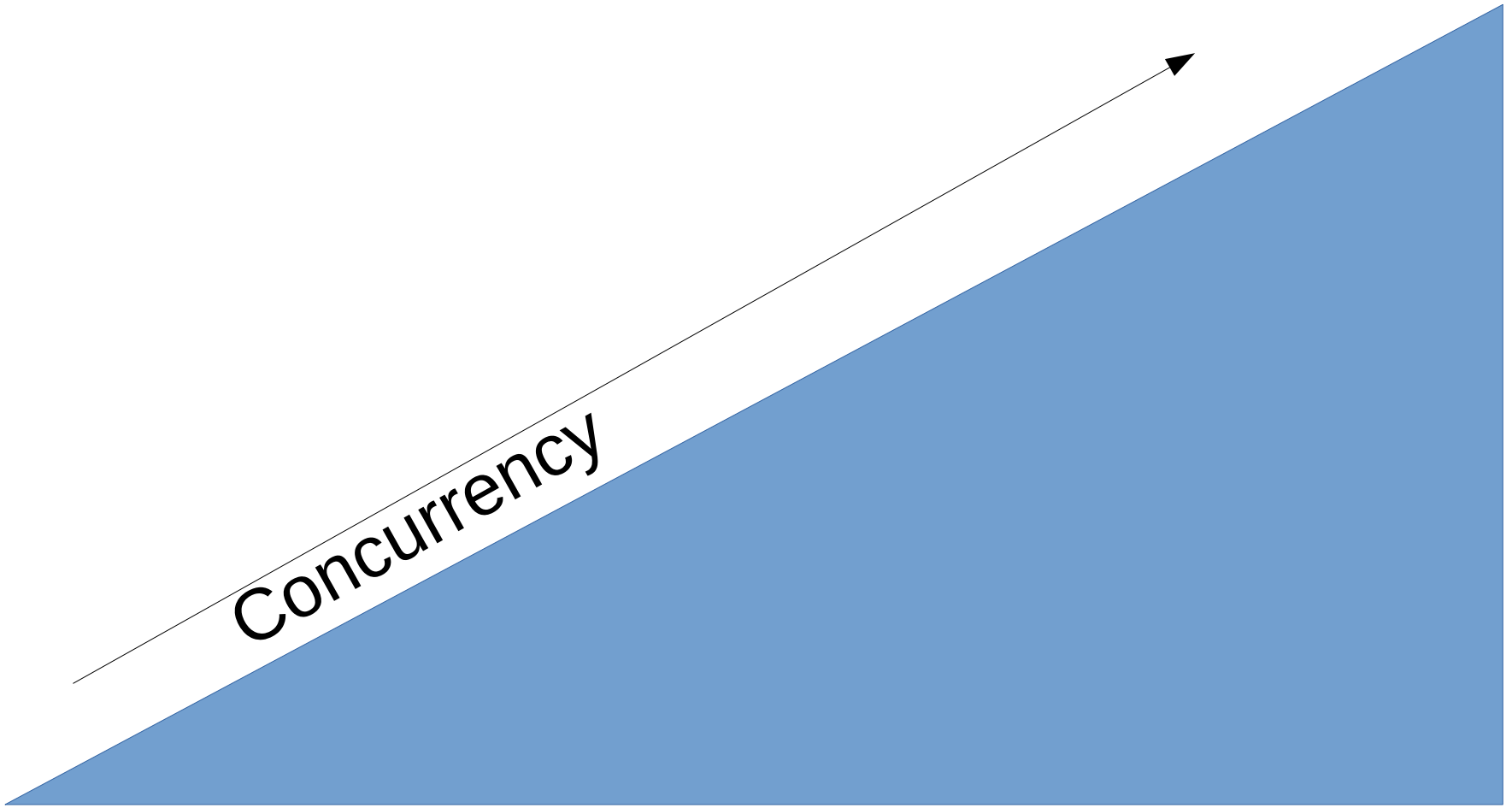
Application requirements have changed.

- 2000
  - Telco applications used 200mb heaps and required <50ms pause times
- 2015
  - E-Commerce applications use 100gb+ heaps and require <10ms pause times

What can we do to shrink pause times?



Increase concurrency



Flood's first law

- Never slow down the mutator (Java) threads unless you absolutely have to.

How is Shenandoah Different?

- We compact objects while the program is running.

Currently Available OpenJDK GC's

- Serial GC
  - Small Footprint
  - Minimal overhead
- Parallel GC
  - High Throughput
- ParNew/CMS
  - Minimal Pause Times
- G1
  - Managed Pause Times
  - Compaction

G1 Managed Pause Times

Choose the size of your collection set so that the copying work will not exceed the allotted pause time.

Shenandoah

- Perform the copying work while the Java threads are running.
- Pause times proportional to root set, basically number of threads.

Shenandoah

Goal: < 10ms GC pauses for 100gb+ heaps.

Specjbb 2015 results

- AMD-Opteron box with 32 threads
- 140gb heap
- Shenandoah
  - RUN RESULT: hbIR (max attempted) = 19881, hbIR (settled) = 16584, max-jOPS = 18092, critical-jOPS = 6290
- G1
  - RUN RESULT: hbIR (max attempted) = 23838, hbIR (settled) = 20773, max-jOPS = 19547, critical-jOPS = 3187

Specjbb2015 pauses

- G1
- Total = 3291
  - Young=3242
  - Mixed=49
- Full=11

- Shenandoah
- Total=892
  - Init Mark 436
  - Final Mark 436
- Full 8 (user requested)

SpecJBB2015 Pause Times not counting full_gc

- G1
  - 3291 pauses
  - 713.56s total
  - 216.82ms avg
  - 2361.48ms max
- Shenandoah
  - 882 pauses
  - 29.89s total
  - 34.28ms avg
  - 379.14ms max

We have a fix planned for the large max value.

Specjbb 2015 results

- Intel Xeon box with 144 threads
- 350gb heap
- Shenandoah
  - RESULT: hbIR (max attempted) = 47576, hbIR (settled) = 43619, max-jOPS = 50431, critical-jOPS = 34169
- G1
  - RUN RESULT: hbIR (max attempted) = 102168, hbIR (settled) = 85156, max-jOPS = 79691, critical-jOPS = 22950

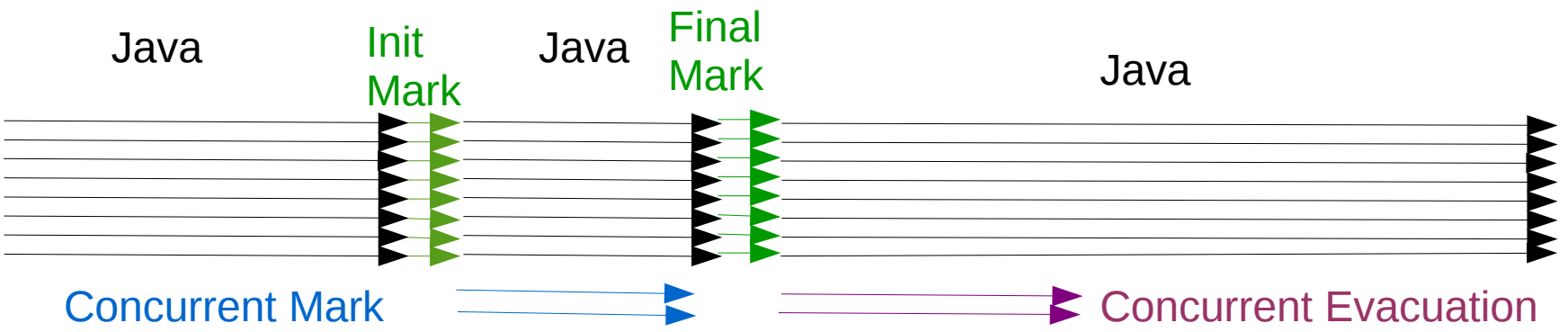
SpecJBB2015 Pause Times not counting full_gc (144 threads)

- G1
  - 2017 pauses
  - 332.59s total
  - 164.89ms avg
  - 1481.96ms max
- Shenandoah
  - 789 pauses
  - 23.25s total
  - 29.47ms avg
  - 323.87ms max

We have a fix planned for the large max value.
Odd pauses due to OOM due to bad heuristics.

How did we do it?

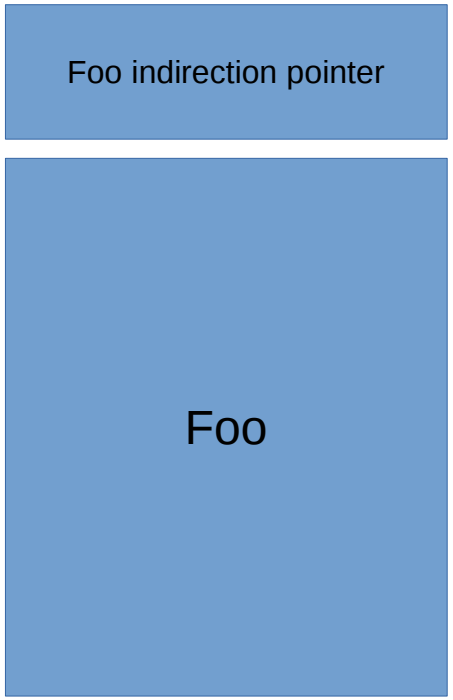
Shenandoah: Current implementation



We use as many threads as are available to do concurrent phases.

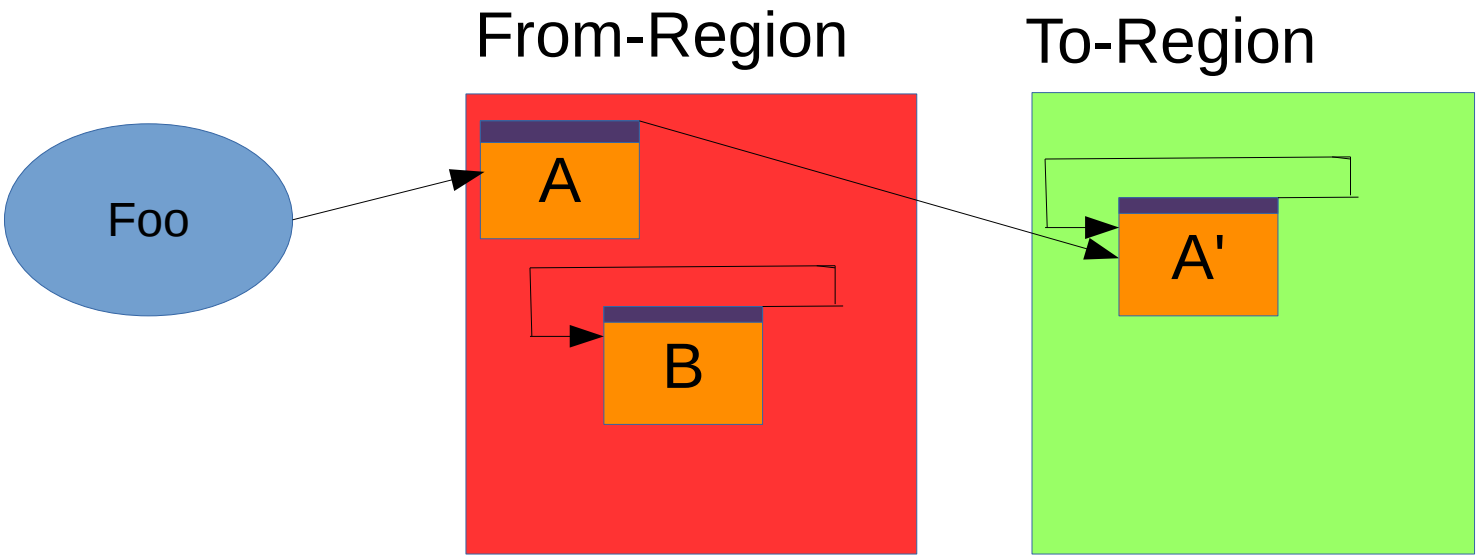
Basic Idea

- Forwarding Pointers

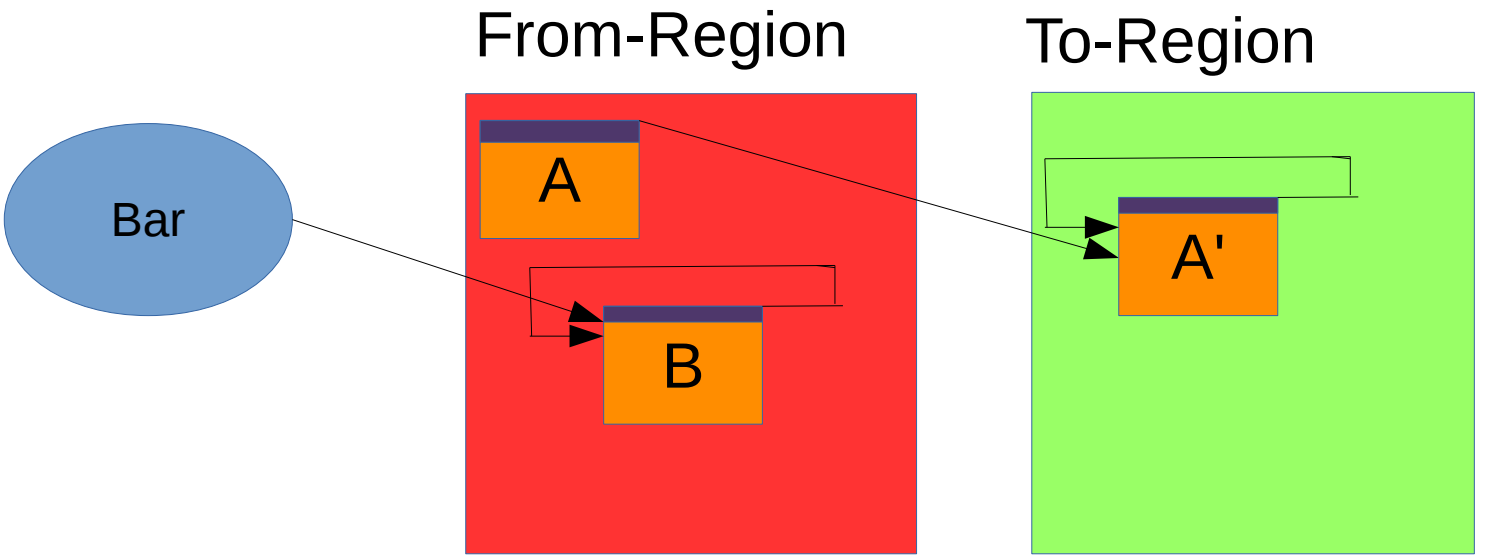


Accesses go through a forwarding pointer to find the real location of the Object.

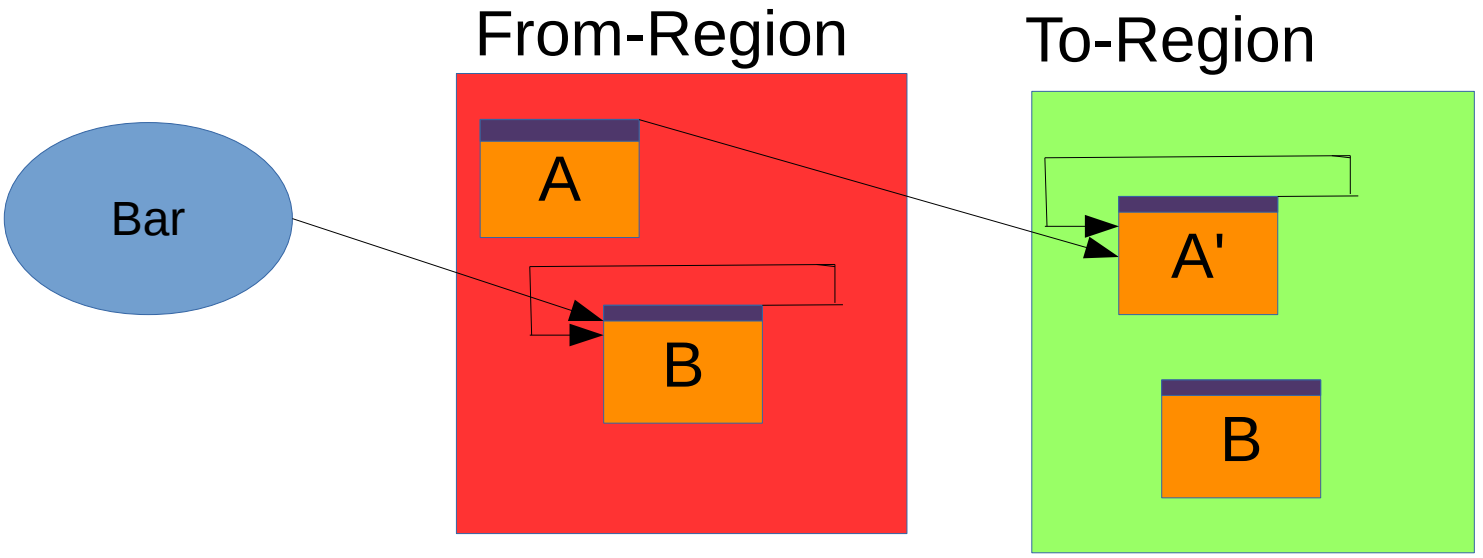
Reads



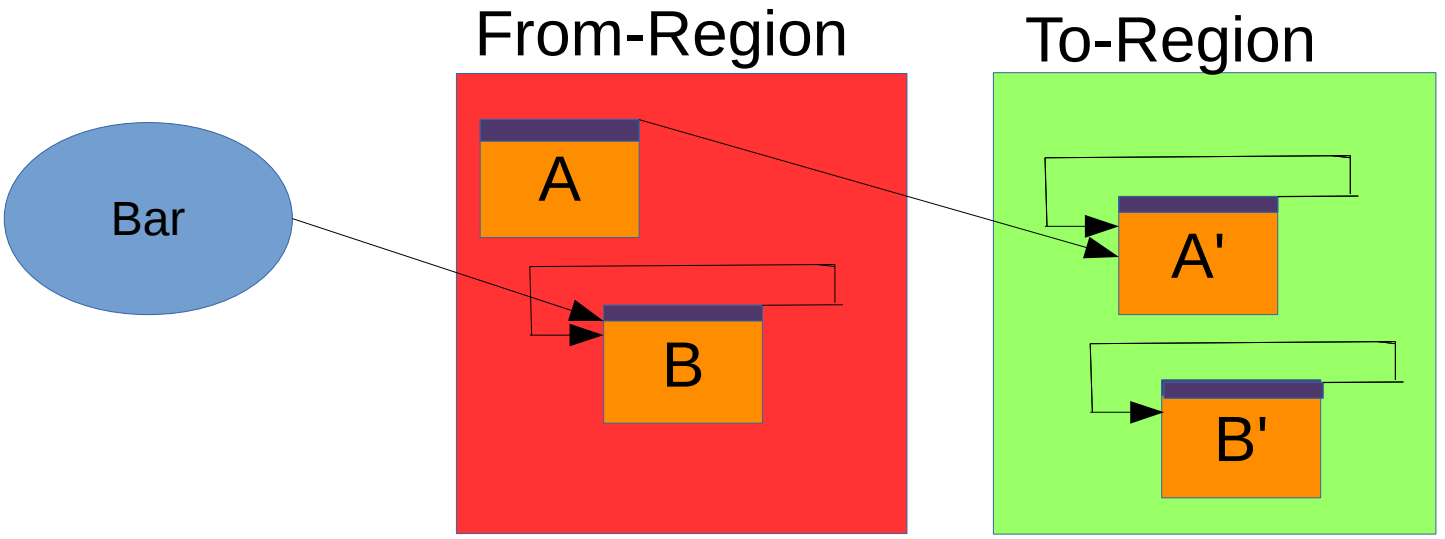
Writes force a copy



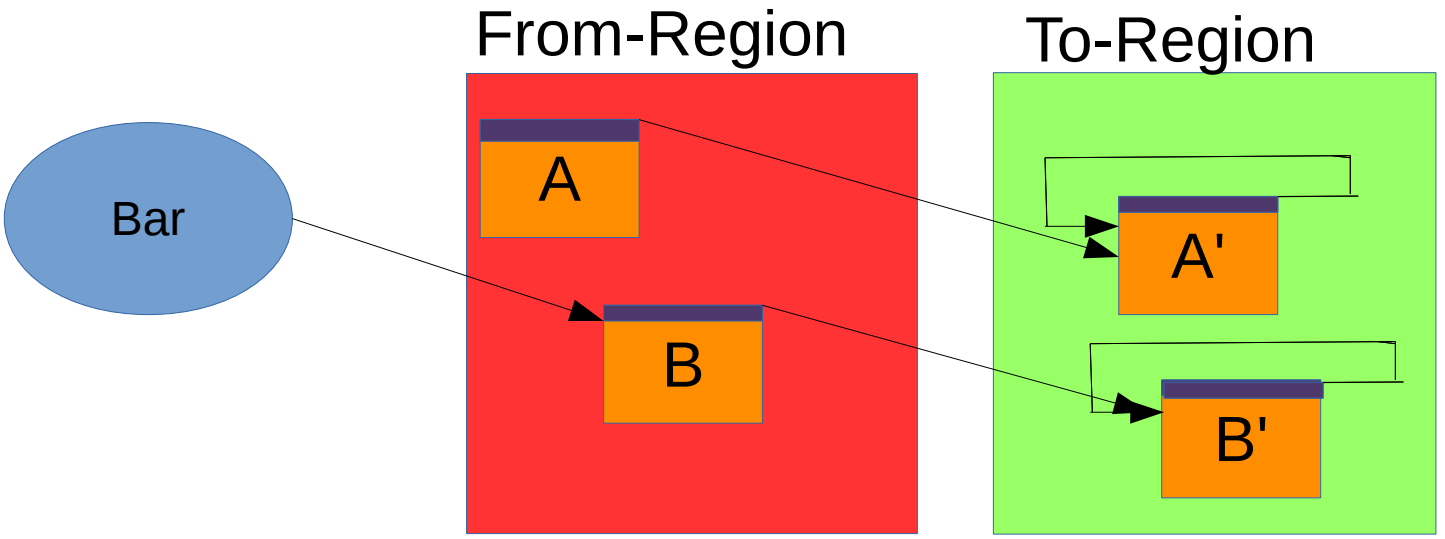
Writes force a copy



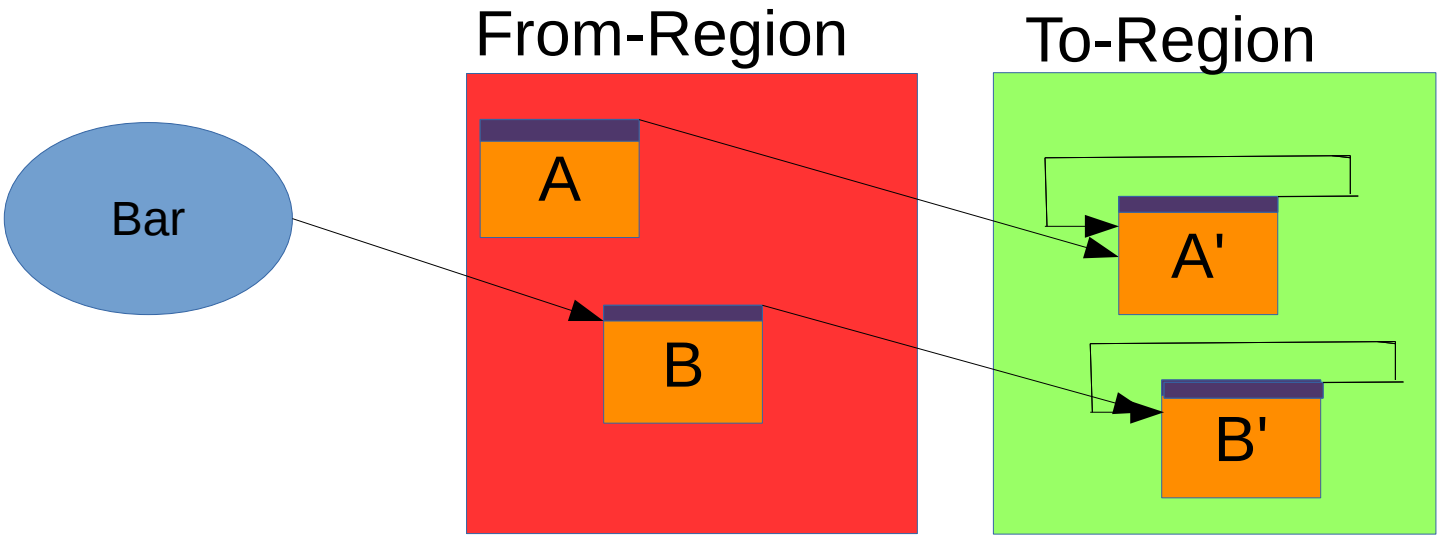
Writes force a copy



Writes force a copy

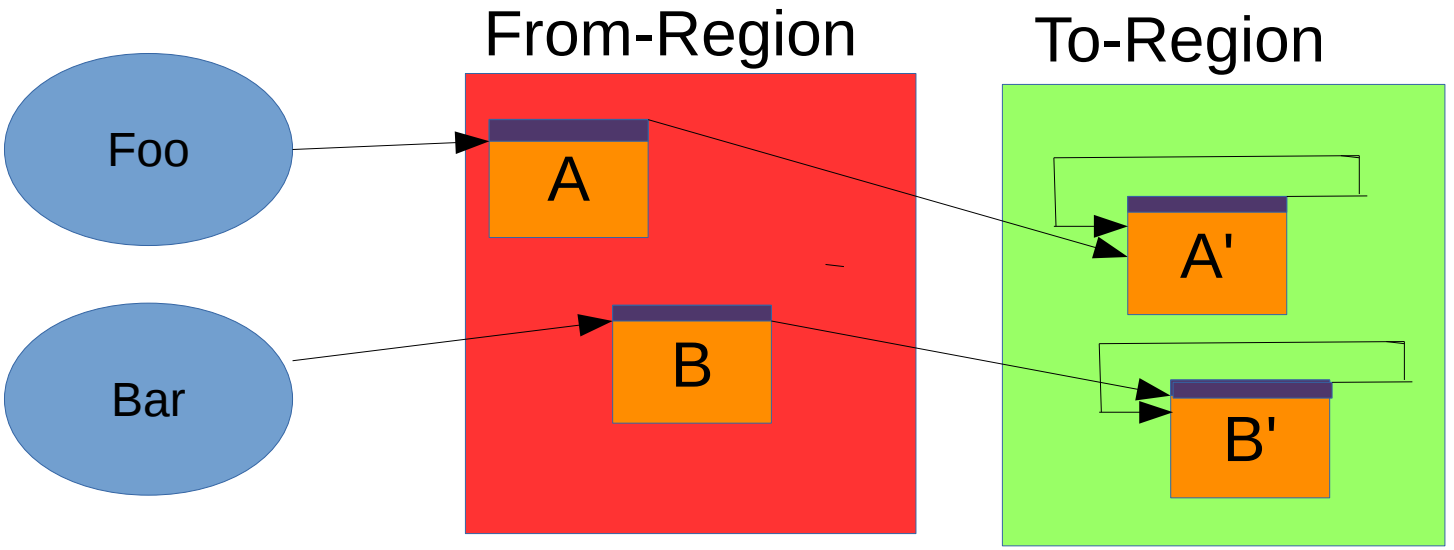


Writes force a copy



Then perform the write on the to-space copy.

Acmp may force two copies



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
  - ...

Shenandoah Barriers

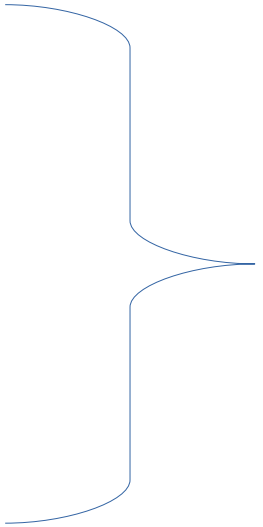
- Write barriers:
  - putfield
  - Xastore
  - Intrinsics
  - ...

Implementing Barriers

- Interpreter
- C1 (client compiler)
- C2 (server compiler)

Dragon Book Compiler

- Source Program
- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Intermediate Code Generator
- Code Optimizer
- Code Generator
- Peephole Optimizer
- Target Program



Front End

Middle End

Back End

Intermediate Representation(IR)

Source Program:

```
Int x = 1
Do {
   Cond = { x != 1}
   If (cond) {
      X = 2;
   }
} while (read());
Return x;
```

SSA:

```
X0: 1;
Do {
   X1: phi(x0,x3);
   Cond: (x1 != 1)
   If (cond) {
      X2: 2;
   }
   X3:phi (x2, x1);
} while (read());
Return x3;
```

Sea of Nodes

- Just another Intermediate representation
- Explicit control and memory dependencies
- Designed to make code generation and generated code fast.

What we tried...

- Parse time barrier insertion
  - Optimization opportunities were missed
- Late barrier insertion
  - Getting this right was tricky
- Macro node barriers
  - Write barriers inhibited important loop optimizations
- So...

We added new nodes.

- ShenandoahReadBarrierNode
- ShenandoahWriteBarrierNode

Shenandoah specific nodes

- Compiler sees through them.
- Inserted at Parse time
- Enabled Shenandoah specific optimizations

Shenandoah specific optimizations

- Don't need read or write barriers on newly allocated objects.
- Don't need read barriers after write barriers.
- Don't need multiple read barriers when reading different fields of the same object.
- Hoist barriers out of hot paths (like loops).

Current status

- We are an official OpenJDK project.
  - http://openjdk.java.net/projects/shenandoah/
- We are stable and getting performance numbers we are proud of.
- Working towards next Fedora release (Spring 2016)
- Always been our goal to put this back upstream in OpenJDK when it's ready.

What's left to do?

- Performance
  - Investigate Anomalies
  - Lock free heap region sets
  - More C2 optimizations?
  - Big application testing.
  - Try out differing architectures.
  - Heuristic Tuning.